Course Script

INF 5110: Compiler construction

INF5110, spring 2019

Martin Steffen

Contents

Chapter
Introduction

What is it about?

Learning Targets of this Chapter

The chapter gives an overview over different phases of a compiler and their tasks.

Contents

1.1 Introduction

Course info

Sources

Different from some previous semesters, one recommended book the course is [6] besides also, as in previous years, [9]. We will not be able to cover the whole book anyway (neither the full [9] book). In addition the slides will draw on other sources, as well. Especially in the first chapters, for the so-called front-end, the material is so "standard" and established, that it almost does not matter, which book to take.

Course material from:

- Martin Steffen (`msteffen@ifi.uio.no`)
- Stein Krogdahl (`stein@ifi.uio.no`)
- Birger Møller-Pedersen (`birger@ifi.uio.no`)
- Eyvind Wærstad Axelsen (`eyvinda@ifi.uio.no`)

Course's web-page

`http://www.uio.no/studier/emner/matnat/ifi/INF5110`

- overview over the course, pensum (watch for updates)
- various announcements, beskjeder, etc.

Course material and plan

- Material: based largely on [6] and [9], but also other sources will play a role. A classic is "the dragon book" [2], we might use part of code generation from there
- see also *errata* list at `http://www.cs.sjsu.edu/~louden/cmptext/`
- approx. 3 hours teaching per week
- mandatory assignments (= "obligs")
  - O_1 published mid-February, deadline mid-March
  - O_2 published beginning of April, deadline beginning of May
- group work up-to 3 people recommended. Please inform us about such planned group collaboration
- slides: see updates on the net
- **exam**: (if written) *12th June, 14:30*, 4 hours.

Motivation: What is CC good for?

- not everyone is actually building a full-blown compiler, **but**
  - fundamental concepts and techniques in CC
  - most, if not basically all, software reads, processes/transforms and outputs "data"
  - ⇒ often involves techniques central to CC
  - understanding compilers ⇒ deeper understanding of programming language(s)
  - new language (domain specific, graphical, new language paradigms and constructs. . .)
  - ⇒ CC & their principles will *never* be "out-of-fashion".

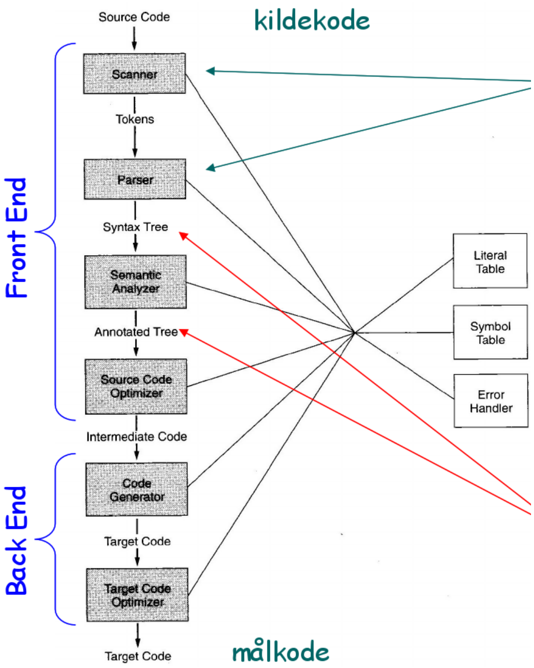
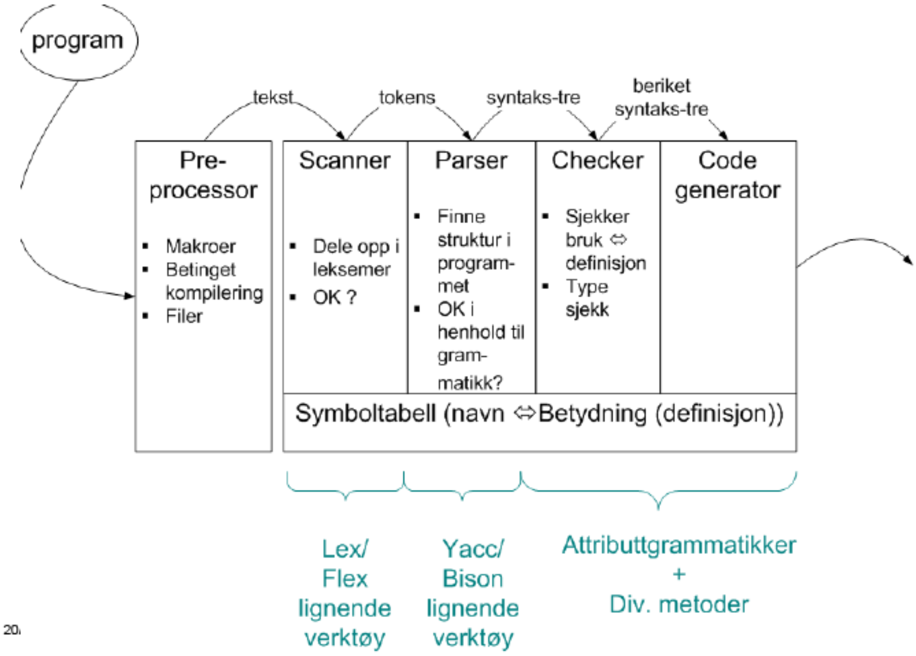


Figure 1.1: Structure of a typical compiler

1.2 Compiler architecture & phases

Architecture of a typical compiler

Anatomy of a compiler


program
tekst
tokens
syntaks-tre
beriket syntaks-tre
Pre-processor
Makroer
Betinget kompilering
Filer
Scanner
Dele opp i leksemer
OK ?
Parser
Finne struktur i program-met
OK i henhold til gram-matikk?
Checker
Sjekker bruk ⇔ definisjon
Type sjekk
Code generator
Symboltabell (navn ⇔Betydning (definisjon))
Lex/ Flex lignende verktøy
Yacc/ Bison lignende verktøy
Attributtgrammatikker + Div. metoder
20

Pre-processor

- either separate program or integrated into compiler
- nowadays: C-style preprocessing mostly seen as "hack" grafted on top of a compiler.[1]
- examples (see next slide):
  - file inclusion[2]
  - macro definition and expansion[3]
  - conditional code/compilation: Note: `#if` is *not* the same as the `if`-programming-language construct.
- problem: often messes up the line numbers

C-style preprocessor examples

```
#include <filename>
```

Listing 1.1: file inclusion

```
#vardef #a = 5; #c = #a+1
...
#if (#a < #b)
     ..
#else
   ...
#endif
```

Listing 1.2: Conditional compilation

Also languages like TeX, LaTeX\ etc. support conditional complication (e.g., `if<condition> ... else ... fi` in TeX). As a side remark: These slides and this script makes quite some use of it: some text shows up only in the handout-version, etc.

C-style preprocessor: macros

```
#macrodef hentdata(#1,#2)
    --- #1----
     #2---(#1)---
#enddef

...
#hentdata(kari,per)
```

Listing 1.3: Macros

[1] C-preprocessing is still considered a *useful* hack, otherwise it would not be around ... But it does not naturally encourage elegant and well-structured code, just quick fixes for some situations.

[2] The single most primitive way of "composing" programs split into separate pieces into one program.

[3] Compare also to the `\newcommand`-mechanism in LaTeX or the analogous `\def`-command in the more primitive TeX-language.

```
 --- kari----
per---(kari)---
```

Note: the code is not really C, it's used to illustrate macros similar to what can be done in C. For real C, see `https://gcc.gnu.org/onlinedocs/cpp/Macros.html`. Comditional compilation is done with

`#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`. and `#endif`. Definitions are done with `#define`.

Scanner (lexer . . .)

- input: "the program text" (= string, char stream, or similar)
- task
  - *divide* and *classify* into *tokens*, and
  - remove blanks, newlines, comments ..
- theory: finite state automata, regular languages

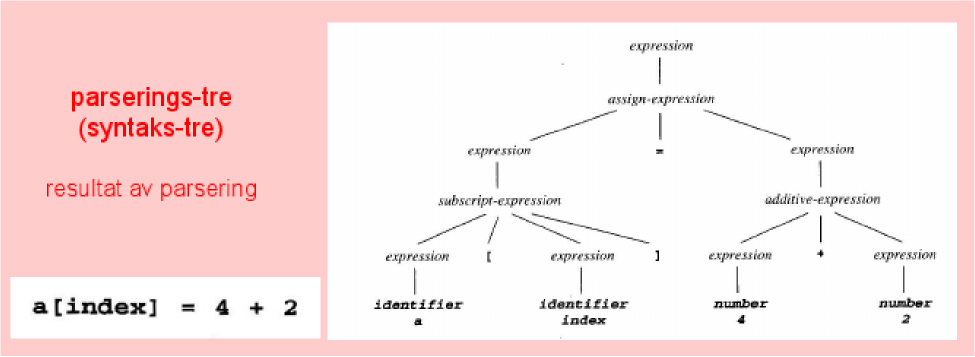
Scanner: illustration

```
a[index]␣=␣4␣+␣2
```

lexeme	token class	value
`a`	*identifier*	`"a"` 2
`[`	*left bracket*	
`index`	*identifier*	`"index"` 21
`]`	*right bracket*	
`=`	*assignment*	
`4`	*number*	`"4"` 4
`+`	*plus sign*	
`2`	*number*	`"2"` 2

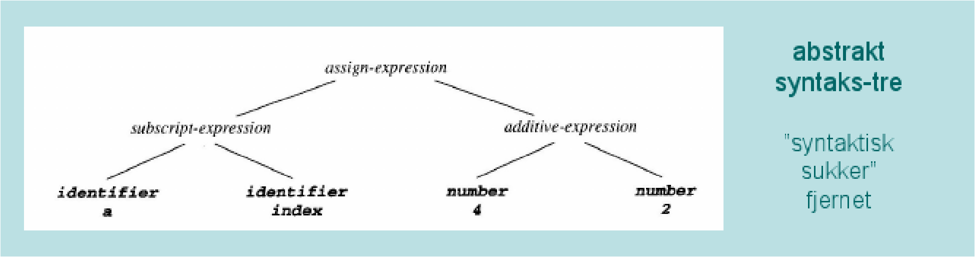
0	
1	
2	`"a"`
	⋮
21	`"index"`
22	
	⋮

Parser


parserings-tre
(syntaks-tre)

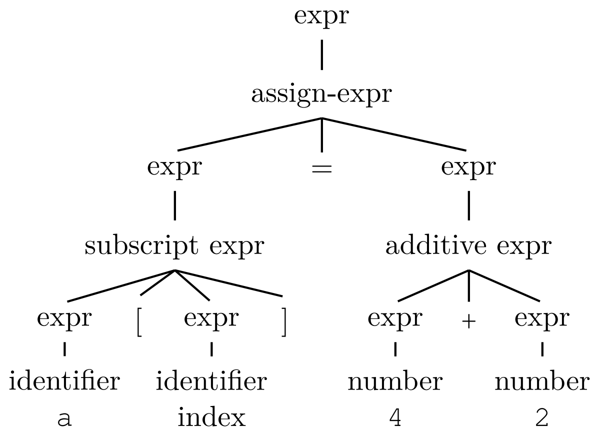
resultat av parsering

`a[index] = 4 + 2`

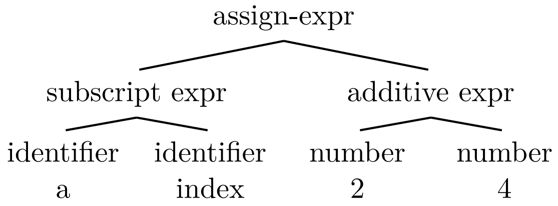

abstrakt
syntaks-tre

"syntaktisk
sukker"
fjernet

`a[index] = 4 + 2`: parse tree/syntax tree



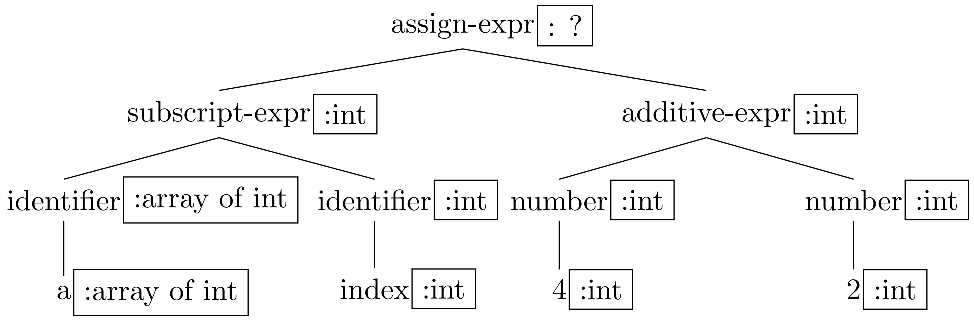
`a[index] = 4 + 2`: abstract syntax tree



The trees here are mainly for illustration. It's not meant as "this is how *the* abstract syntax tree looks like" for the example. In general, abstract syntax trees are less verbose that parse trees. The latter are sometimes also called *concrete* syntax trees. The parse tree(s) for a given word are fixed by the *grammar*. The abstract syntax tree is a bit a matter of design. Of course, the grammar is also a matter of design, but once the grammar is fixed, the parse trees are fixed, as well. What *is* typical in the illustrative example is: an abstract syntax tree would not bother to add nodes representing brackets (or parentheses etc), so those are omitted. In general, ASTs are more compact, ommitting superfluous information without omitting *relevant* information.

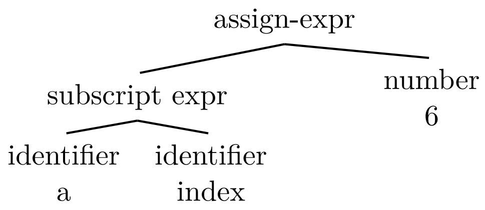
(One typical) Result of semantic analysis

- one standard, general outcome of semantic analysis: "annotated" or "decorated" AST
- additional info (non context-free):
  - *bindings* for declarations
  - (static) *type* information



- here: *identifiers* looked up wrt. declaration
- 4, 2: due to their form, basic types.

Optimization at source-code level



1

```
t = 4+2;
a[index] = t;
```

2

```
t = 6;
a[index] = t;
```

3

```
a[index] = 6;
```

The lecture will not dive too much into optimizations. The ones illustrated here are known as *constant folding* and *constant propagation*. Optimizations can be done (and actually are done) in various phases on the compiler. Here we said, optimization at "source-code level", and what is typically meant by that is optimization on the abstract syntax tree (presumably at the AST after type checking and some semantic analysis). The AST is considered so close to the actual input that one still considers it as "source code" and no one tries serioiusly optimize code a the input-string level. If the compiler "massages" the input, it's mostly not seen as optimization, it's rather (re-)formatting. There are indeed format-tool that assist the user to have the program is a certain "standardized" format (standard indentation, new-lines appropriately, etc.)

Concerning optimization, what is also typical is,that there are many different optimizations building upon each other. First, optimization A is done, then, taking the result, optimization B, etc. Sometimes even doing A again, and then B again, etc.

Code generation & optimization

```
MOV  R0, index ;;  value of index -> R0
MUL  R0, 2     ;;  double value of R0
MOV  R1, &a    ;;  address of a -> R1
ADD  R1, R0    ;;  add R0 to R1
MOV *R1, 6     ;;  const 6 -> address in R1
```

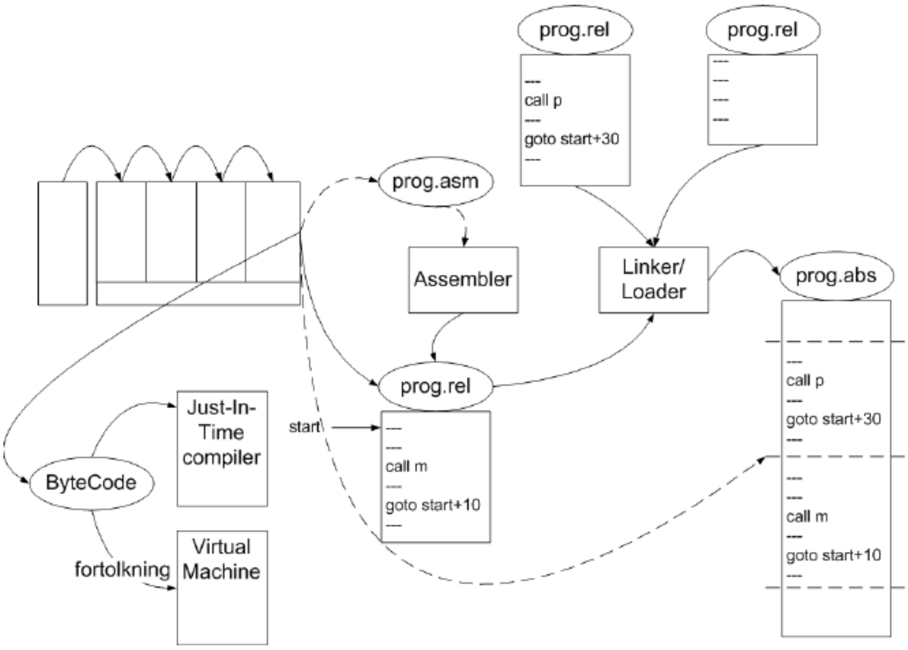
```
MOV R0, index      ;; value of index -> R0
SHL R0             ;; double value in R0
MOV &a[R0], 6      ;; const 6 -> address a+R0
```

- *many* optimizations possible
- potentially difficult to automatize[4], based on a formal description of language and machine
- platform dependent

[4] Not that one has much of a choice. Difficult or not, *no one* wants to optimize generated machine code by hand . . . .

For now it's not too important what the code snippets do. It should be said, though, that it's not a priori always clear in which way a transformation such as the one shown is an *improvement*. One transformation that most probably is an improvement, that's the "shift left" for doubling. Another one is that the program is *shorter*. Program size is something that one might like to "optmize" in itself. Also: ultimately each machine operation needs to be *loaded* to the processor (and that costs time in itself). Note, however, that it's generally not the case that "one assembler line costs one unit of time". Especially, the last line in the second program could costs more than other simpler operations. In general, operations on *registers* are quite faster anyway than those referring to main memory. In order to make a meaningful statement of the effect of a program transformation, one would need to have a *"cost model"* taking register access vs. memory access and other aspects into account.

Anatomy of a compiler (2)



Misc. notions

- front-end vs. back-end, analysis vs. synthesis
- separate compilation
- how to handle *errors*?
- "data" handling and management at run-time (static, stack, heap), garbage collection?
- language can be compiled in *one pass*?
  - E.g. C and Pascal: declarations must *precede* use
  - no longer too crucial, enough memory available
- compiler assisting tools and infrastructure, e.g.
  - debuggers

  - profiling
  - project management, editors
  - build support
  - ...

Compiler vs. interpeter

compilation

- classical: source ⇒ machine code for given machine
- different "forms" of machine code (for 1 machine):
  - executable ⇔ relocatable ⇔ textual assembler code

full interpretation

- directly executed from program code/syntax tree
- often for command languages, interacting with the OS, etc.
- speed typically 10–100 slower than compilation

compilation to intermediate code which is interpreted

- used in e.g. Java, Smalltalk, ...
- intermediate code: designed for efficient execution (byte code in Java)
- executed on a simple interpreter (JVM in Java)
- typically 3–30 times slower than direct compilation
- in Java: byte-code ⇒ machine code in a just-in time manner (JIT)

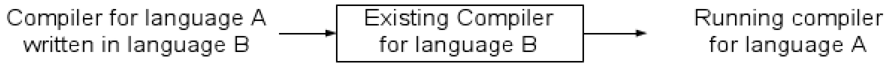
More recent compiler technologies

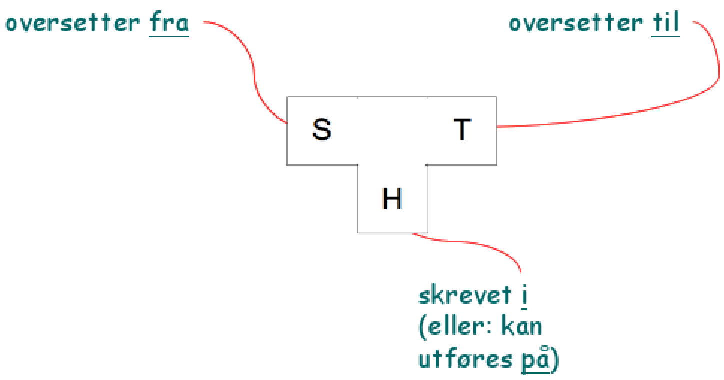
- *Memory* has become cheap (thus comparatively large)
  - keep whole program in main memory, while compiling
- OO has become rather popular
  - special challenges & optimizations
- Java
  - "compiler" generates byte code
  - part of the program can be *dynamically* loaded during run-time
- concurrency, multi-core
- virtualization
- graphical languages (UML, etc), "meta-models" besides grammars

1.3 Bootstrapping and cross-compilation

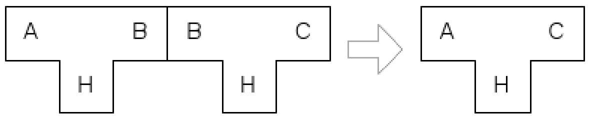
Compiling from source to target on host

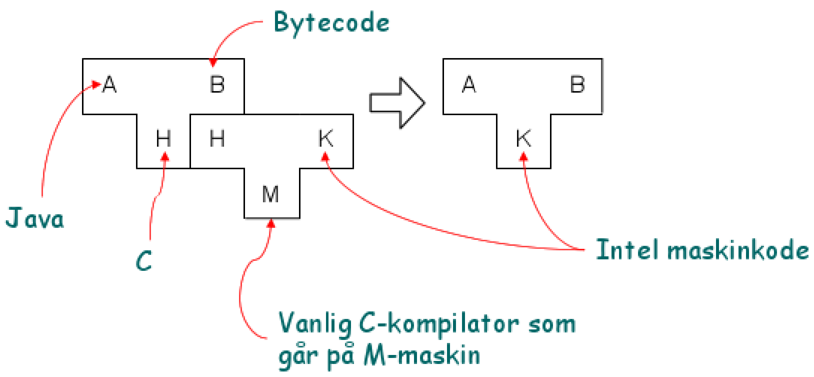
“tombstone diagrams” (or T-diagrams). . . .


Compiler for language A written in language B → Existing Compiler for language B → Running compiler for language A

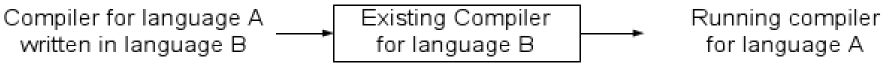


Two ways to compose “T-diagrams”





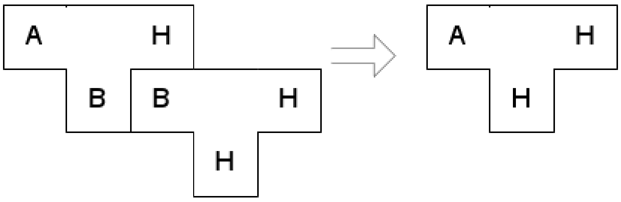
Using an “old” language and its compiler for write a compiler for a “new” one



Compiler for language A written in language B

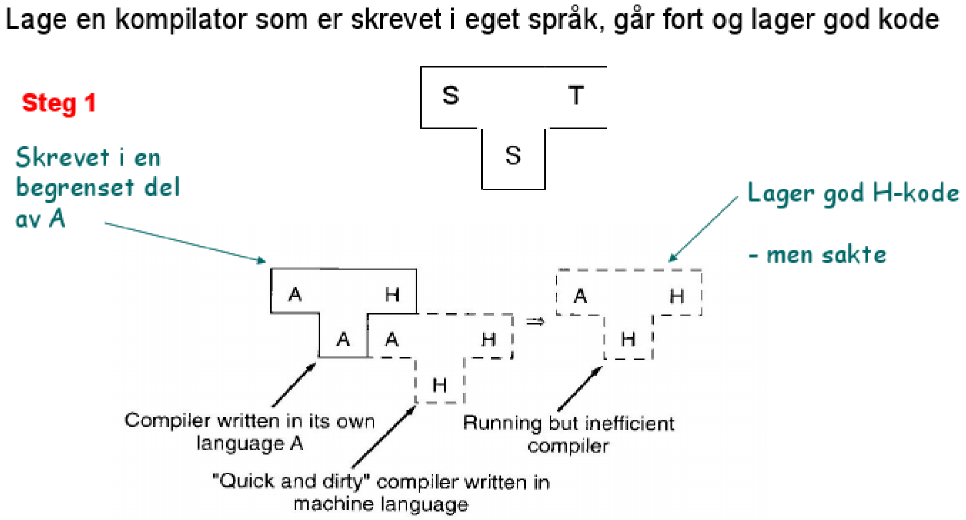
Existing Compiler for language B

Running compiler for language A



Pulling oneself up on one’s own bootstraps

> bootstrap (verb, trans.): to promote or develop . . . with little or no assistance
>
> — Merriam-Webster



Lage en kompilator som er skrevet i eget språk, går fort og lager god kode

Steg 1

Skrevet i en begrenset del av A

Lager god H-kode

- men sakte

Compiler written in its own language A

"Quick and dirty" compiler written in machine language

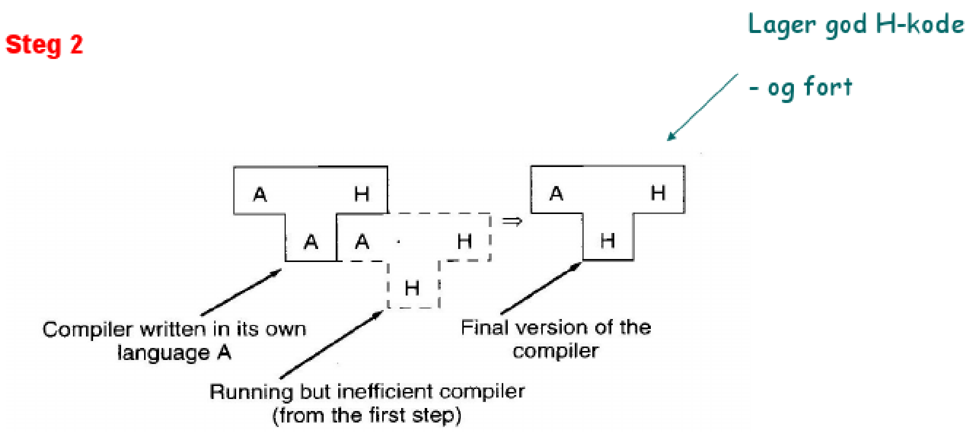
Running but inefficient compiler

Explanation

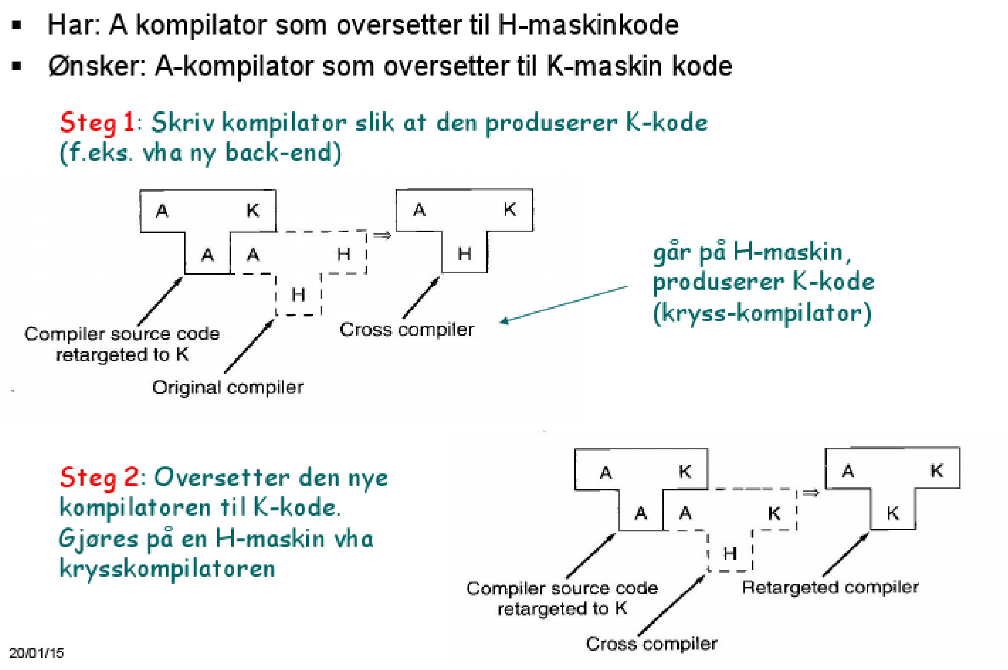
There is no magic here. The first thing is: the “Q&D” compiler in the diagram is said to be in machine code. If we want to run that compiler as executable (as opposed to being interpreted, which is ok too), of course we need machine code, but it does not mean that *we* have to write that Q&D compiler in machine code. Of course we can use the approach explained before that we use an existing language with an existing compiler to create that machine-code version of the Q&D compiler.

Furthermore: when talking about *efficiency* of a compiler, we mean (at least here) exactly that: it’s the compilation process itself which is inefficent! As far as efficency goes, one the one hand the compilation process can be efficient or not, and on the other the generated code can be (on average and given competen programmers) be efficent not. Both aspects are not independent, though: to generate very efficient code, a compiler might use many and aggressive optimizations. Those may produce efficient code but cost time to do. At the first stage, we don’t care how long it takes to compile, and *also* not how efficient is the *code it produces!* Note the that code that it produces is a compiler, it’s actually a second version of “same” compiler, namely for the new language A to H and on H. We don’t care how efficient the generated code, i.e., the compiler is, because we use it just in the next step, to generate the final version of compiler (or perhaps one step further to the final compiler).

Bootstrapping 2


Steg 2
Lager god H-kode
- og fort
A
H
A
A
H
H
⇒
A
H
H
Compiler written in its own language A
Running but inefficient compiler (from the first step)
Final version of the compiler

Porting & cross compilation



- Har: A kompilator som oversetter til H-maskinkode
- Ønsker: A-kompilator som oversetter til K-maskin kode

Steg 1: Skriv kompilator slik at den produserer K-kode (f.eks. vha ny back-end)

går på H-maskin, produserer K-kode (kryss-kompilator)

Steg 2: Oversetter den nye kompilatoren til K-kode. Gjøres på en H-maskin vha krysskompilatoren

20/01/15

The situation is that K is a new "platform" and we want to get a compiler for our new language A for K (assuming we have one already for the old platform H). It means that not only we want to compile *onto* K, but also, of course, that it has to run on K. These are two requirements: (1) a compiler *to* K and (2) a compiler to run *on* K. That leads to two stages.

In a first stage, we "rewrite" our compiler for A, targeted towards H, to the new platform K. If structured properly, it will "only" require to *port* or *re-target* the so-called back-end from the old platform to the new platform. If we have done that, we can use our executable compiler on H to generate code for the new platform K. That's known as *cross-compilation*: use platform H to generate code for platform K.

But now, that we have a (so-called cross-)compiler from A to K, running on the old platform H, we use it to compile the retargeted compiler *again*!

Part
Front end

Chapter
Scanning

What is it about?

Learning Targets of this Chapter

1. alphabets, languages,
2. regular expressions
3. finite state automata / recognizers
4. connection between the two concepts
5. minimization

The material corresponds roughly to [6, Section 2.1–2.5] or ar large part of [9, Chapter 2]. The material is pretty canonical, anyway.

Contents

2.1 Introduction

Scanner section overview

What's a scanner?

- Input: source code.[1]
- Output: sequential stream of **tokens**

- *regular expressions* to describe various token classes
- (deterministic/non-determinstic) finite-state automata (FSA, DFA, NFA)
- implementation of FSA
- regular expressions → NFA
- NFA ↔ DFA

[1] The argument of a scanner is often a *file name* or an *input stream* or similar.

What's a scanner?

- other names: lexical scanner, **lexer**, tokenizer

A scanner's functionality

Part of a compiler that takes the source code as input and translates this stream of characters into a stream of **tokens**.

More info

- char's typically language independent.[2]
- *tokens* already language-specific.[3]
- works always "left-to-right", producing one *single token* after the other, as it scans the input[4]
- it "segments" char stream into "chunks" while at the same time "classifying" those pieces ⇒ **tokens**

Typical responsibilities of a scanner

- segment & classify char stream into tokens
- typically described by "rules" (and **regular expressions**)
- typical language aspects covered by the scanner
  - describing *reserved words* or *key words*
  - describing format of *identifiers* (= "strings" representing variables, classes . . .)
  - comments (for instance, between `//` and `NEWLINE`)
  - *white space*
    - to segment into tokens, a scanner typically "jumps over" white spaces and afterwards starts to determine a new token
    - not only "blank" character, also `TAB`, `NEWLINE`, etc.
- lexical rules: often (explicit or implicit) *priorities*
  - *identifier* or *keyword*? ⇒ keyword
  - take the *longest* possible scan that yields a valid token.

"Scanner = regular expressions (+ priorities)"

Rule of thumb

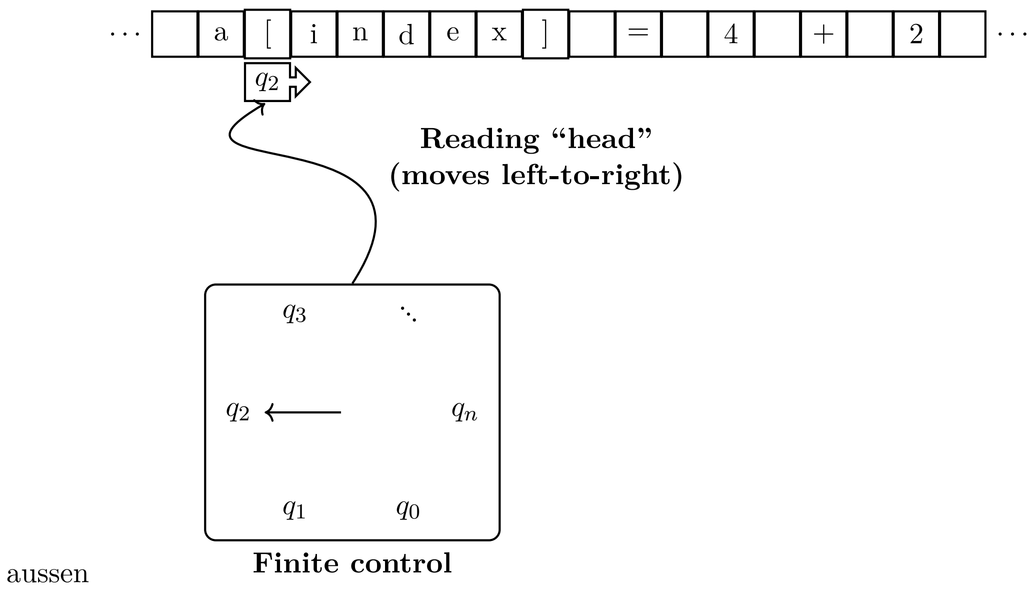
Everything about the source code which is so simple that it can be captured by **reg. expressions** belongs into the scanner.

[2] Characters are language-independent, but perhaps the encoding (or its interpretation) may vary, like ASCII, UTF-8, also Windows-vs.-Unix-vs.-Mac newlines etc.

[3] There are large commonalities across many languages, though.

[4] No theoretical necessity, but that's how also humans consume or "scan" a source-code text. At least those humans trained in e.g. Western languages.

How does scanning roughly work?



```
a[index] = 4 + 2
```

How does scanning roughly work?

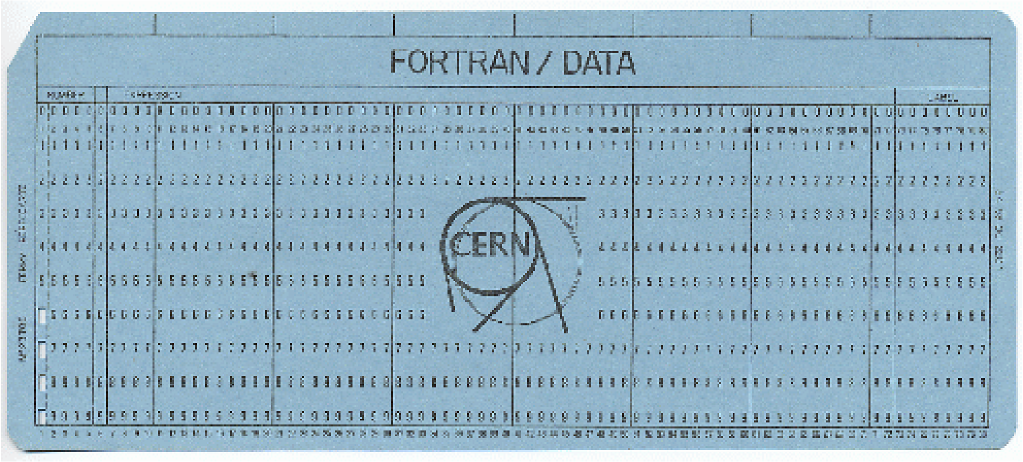
- usual invariant in such pictures (by convention): arrow or head points to the *first* character to be *read next* (and thus *after* the last character having been scanned/read last)
- in the scanner *program* or procedure:
  - analogous invariant, the arrow corresponds to a *specific variable*
  - contains/points to the next character to be read
  - name of the variable depends on the scanner/scanner tool
- the *head* in the pic: for illustration, the scanner does not really have a "reading head"
  - remembrance of Turing machines, or
  - the old times when perhaps the program data was stored on a tape.[5]

The bad(?) old times: Fortran

- in the days of the pioneers
- main memory was *smaaaaaaaaaall*
- compiler technology was not well-developed (or not at all)
- programming was for *very* few "experts".[6]
- Fortran was considered high-level (wow, a language so complex that you had to compile it . . .)

[5] Very deep down, if one still has a magnetic disk (as opposed to SSD) the secondary storage still has "magnetic heads", only that one typically does not parse *directly* char by char from disk. . .

[6] There was no computer science as profession or university curriculum.



(Slightly weird) lexical ascpects of Fortran

Lexical aspects = those dealt with by a scanner

- **whitespace** *without* "meaning":
  `I F( X 2.  EQ. 0) TH E N` vs. `IF ( X2.  EQ.0 ) THEN`
- no *reserved* words!
  `IF (IF.EQ.0) THEN THEN=1.0`
- general *obscurity* tolerated:
  `DO99I=1,10` vs. `DO99I=1.10`

```
DO 99 I=1,10
 -
 -
99 CONTINUE
```

Fortran scanning: remarks

- Fortran (of course) has evolved from the pioneer days ...
- no keywords: nowadays mostly seen as *bad* idea[7]
- treatment of white-space as in Fortran: not done anymore: `THEN` and `TH EN` *are* different things in all languages
- however:[8] both considered "the same":

Ifthen

```
if b then ..
```

[7] It's mostly a question of language *pragmatics*. Lexers/parsers would have no problems using `while` as variable, but humans tend to.

[8] Sometimes, the part of a lexer / parser which removes whitespace (and comments) is considered as separate and then called *screener*. Not very common, though.

Ifthen2

```
if␣␣␣b␣␣␣␣then␣..
```

- since concepts/tools (and much memory) were missing, Fortran scanner and parser (and compiler) were
  - quite simplistic
  - syntax: designed to "help" the lexer (and other phases)

A scanner classifies

- "good" classification: depends also on later phases, may not be clear till later

Rule of thumb

Things being treated equal in the syntactic analysis (= parser, i.e., subsequent phase) should be put into the same category.

- terminology not 100% uniform, but most would agree:

Lexemes and tokens

Lexemes are the "chunks" (pieces) the scanner produces from segmenting the input source code (and typically dropping whitespace). **Tokens** are the result of *classifying* those lexemes.

- token = token name × token value

A scanner classifies & does a bit more

- token data structure in *OO* settings
  - token themselves defined by classes (i.e., as instance of a class representing a specific token)
  - token values: as attribute (instance variable) in its values
- often: scanner does slightly *more* than just classification
  - store names in some *table* and store a corresponding index as attribute
  - store text constants in some *table*, and store corresponding index as attribute
  - even: *calculate* numeric constants and store value as attribute

One possible classification

name/identifier	`abc123`
integer constant	`42`
real number constant	`3.14E3`
text constant, string literal	`"this is a text constant"`
arithmetic op's	`+ - * /`
boolean/logical op's	`and or not` (alternatively `/\ \/`)
relational symbols	`<= < >= > = == !=`
all other tokens:	`{ } ( ) [ ] , ; := .` etc.
every one it its own group	

- this classification: not the only possible (and not necessarily complete)
- note: *overlap*:
  - `"."` is here a token, but also part of real number constant
  - `"<"` is part of `"<="`

One way to represent tokens in C

```
typedef struct {
  TokenType tokenval;
  char * stringval;
  int numval;
} TokenRecord;
```

If one only wants to store one attribute:

```
typedef struct {
  Tokentype tokenval;
  union
  { char * stringval;
    int numval
  } attribute;
} TokenRecord;
```

How to define lexical analysis and implement a scanner?

- even for complex languages: lexical analysis (in principle) not hard to do
- "manual" implementation straightforwardly possible
- *specification* (e.g., of different token classes) may be given in "prosa"
- however: there are straightforward formalisms and efficient, rock-solid tools available:
  - easier to specify unambigously
  - easier to communicate the lexical definitions to others
  - easier to change and maintain
- often called **parser generators** typically not just generate a scanner, but code for the next phase (parser), as well.

Prosa specification

A precise prosa specification is not so easy to achieve as one might think. For ASCII source code or input, things are basically under control. But what if dealing with unicode? Checking "legality" of user input to avoid SQL injections or similar format string attacks can involve lexical analysis/scanning. If you "specify" in English: " *Backlash is a control character and forbidden as user input* ", which characters (besides `char 92` in ASCII) in Chinese Unicode represents actually other versions of backslash? Note: unclarities about "what's a backslash" have been used for security attacks. Remember that "the" backslash-character in OSs often has a special status, like it *cannot* be part of a file-name but used as separator between file names, denoting a *path* in the file system. If one can "smuggle in" an inofficial ("chinese") backslash into a file-name, one can potentially access parts of the file directory tree, which are supposed to be inaccessible.

Parser generator

The most famous pair of lexer+parser tools is called "compiler compiler" (lex/yacc = "yet another compiler compiler") since it generates (or "compiles") an important part of the front end of a compiler, the lexer+parser. Those kinds of tools are seldomly called compiler compilers any longer.

2.2 Regular expressions

General concept: How to generate a scanner?

1. **regular expressions** to describe language's *lexical* aspects
   - like whitespaces, comments, keywords, format of identifiers etc.
   - often: more "user friendly" variants of reg-exprs are supported to specify that phase
2. *classify* the lexemes to tokens
3. translate the reg-expressions ⇒ NFA.
4. turn the NFA into a *deterministic* FSA (= DFA)
5. the DFA can straightforwardly be implementated

- step done automatically by a "lexer generator"
- lexer generators help also in other user-friendly ways of specifying the lexer: defining *priorities*, assuring that the longest possible token is given back, repeat the processs to generate a sequence of tokens[9]

The classification in step 2 is actually *not* directly covered by the classical Reg-expr = DFA = NFA results, it's something extra. The classical constructions presented here are used to *recognise* (or reject) words. As a "side effect", in an actual implementation, the "class" of the word needs to be given back as well, i.e., the corresponding *token* needs to be concstructed and handed over (step by step) to the next compiler phase, the parser.

[9] Maybe even prepare useful error messages if scanning (not scanner generation) fails.

Use of regular expressions

- **regular languages**: fundamental class of "languages"
- **regular expressions**: standard way to describe regular languages
- not just used in compilers
- often used for flexible " *searching* ": simple form of pattern matching
- e.g. input to search engine interfaces
- also supported by many editors and text processing or scripting languages (starting from classical ones like `awk` or `sed`)
- but also tools like `grep` or `find` (or general "globbing" in shells)

```
find . -name "*.tex"
```

- often *extended* regular expressions, for user-friendliness, not theoretical expressiveness

As for the origin of regular expressions: one starting point is Kleene [8] and there had been earlier works *outside* "computer science".

Kleene was a famous mathematician and influence on theoretical computer science. Funnily enough, regular languages came up in the context of neuro/brain science. See the following link for the origin of the terminology. Perhaps in the early years, people liked to draw connections between between biology and machines and used metaphors like "electronic brain", etc.

Alphabets and languages

Definition 2.2.1 (Alphabet Σ)**.** Finite set of elements called "letters" or "symbols" or "characters".

Definition 2.2.2 (Words and languages over Σ)**.** Given alphabet Σ, a **word** over Σ is a finite sequence of letters from Σ. A **language** over alphabet Σ is a *set* of finite *words* over Σ.

- practical examples of alphabets: ASCII, Norwegian letters (capital and non-capitals) etc.

In this lecture: we avoid terminology "symbols" for now, as later we deal with e.g. symbol tables, where symbols means something slighly different (at least: at a different level). Sometimes, the Σ is left "implicit" (as assumed to be understood from the context).

Remark: Symbols in a symbol table (see later)

In a certain way, symbols in a symbol table can be seen similar to symbols in the way we are handled by automata or regular expressions now. They are simply "atomic" (not further dividable) members of what one calls an alphabet. On the other hand, in practical terms inside a compiler, the symbols here in the scanner chapter live on a different level compared to symbols encountered in later sections, for instance when discussing symbol tables. Typically here, they are *characters*, i.e., the alphabet is a so-called character set, like for instance, ASCII. The lexer, as stated, segments and classifies the sequence of characters and hands over the result of that process to the parser. The results is a sequence of *tokens*, which is what the parser has to deal with later. It's on that parser-level, that the pieces (notably the identifiers) can be treated as atomic pieces of some language, and what is known as the symbol table typcially operates on symbols at that level, not at the level of individual characters.

Languages

- note: Σ is finite, and words are of *finite* length
- languages: in general infinite sets of words
- simple examples: Assume $\Sigma = \{a, b\}$
- *words* as finite "sequences" of letters
  - ϵ: the empty word (= empty sequence)
  - ab means " first a then b "
- sample languages over Σ are
  1. $\{\}$ (also written as $\emptyset$) the empty set
  2. $\{a, b, ab\}$: language with 3 finite words
  3. $\{\epsilon\}$ ($\neq \emptyset$)
  4. $\{\epsilon, a, aa, aaa, \ldots\}$: infinite languages, all words using only a 's.
  5. $\{\epsilon, a, ab, aba, abab, \ldots\}$: alternating a's and b's
  6. $\{ab, bbab, aaaaa, bbabbabab, aabb, \ldots\}$: ?????

Remark 1 (Words and strings)**.** *In terms of a real implementation: often, the letters are of type character (like type `char` or `char32` ...) words then are "sequences" (say arrays) of characters, which may or may not be identical to elements of type `string`, depending on the language for implementing the compiler. In a more conceptual part like here we do not write words in "string notation" (like "ab"), since we are dealing abstractly with sequences of letters, which, as said, may not actually be strings in the implementation. Also in the more conceptual parts, it's often good enough when handling alphabets with 2 letters, only, like $\Sigma = \{a, b\}$ (with one letter, it gets unrealistically trivial and results may not carry over to the many-letter alphabets). But 2 letters are often enough to illustrate some concepts, after all, computers are using 2 bits only, as well ....*

Finite and infinite words

There are important applications dealing with *infinite* words, as well, or also even infinite alphabets. For traditional scanners, one mostly is happy with finite Σ 's and especially sees no use in scanning infinite "words". Of course, some character sets, while not actually infinite, are large (like Unicode or UTF-8)

Sample alphabets

Often we operate for illustration on alphabets of size 2, like $\{a, b\}$. One-letter alphabets are uninteresting, let alone 0-letter alphabets. 3 letter alphabets may not add much as far as "theoretical" questions are concerned. That may be compared with the fact that computers ultimately operate in words over two different "bits" .

How to describe languages

- language mostly here in the abstract sense just defined.
- the "dot-dot-dot" (...) is not a good way to describe to a computer (and to many humans) what is meant (what was meant in the last example?)
- enumerating explicitly all allowed words for an infinite language does not work either

Needed

A **finite** way of describing infinite languages (which is hopefully efficiently implementable & easily readable)

Is it apriori to be expected that *all* infinite languages can even be captured in a finite manner?

- small metaphor

$$2.727272727\ldots \qquad 3.1415926\ldots \tag{2.1}$$

Remark 2 (Programming languages as "languages")**.** *Well, Java etc., seen syntactically as all possible strings that can be compiled to well-formed byte-code, also is a* language *in the sense we are currently discussing, namely a set of words over unicode. But when speaking of the "Java-language" or other programming languages, one typically has also other aspects in mind (like what a program does when it is executed), which is not covered by thinking of Java as an infinite set of strings.*

Remark 3 (Rational and irrational numbes)**.** *The illustration on the slides with the two numbers is partly meant as that: an illustration drawn from a field you may know. The first number from equation (2.1) is a* rational number*. It corresponds to the fraction*

$$\frac{30}{11} . \tag{2.2}$$

That fraction is actually an acceptable finite representation for the "endless" notation 2.72727272... *using "... " As one may remember, it may pass as a decent definition of rational numbers that they are exactly those which can be represented finitely as fractions of two integers, like the one from equation (2.2). We may also remember that it is characteristic for the "endless" notation as the one from equation (2.1), that for rational numbers, it's* periodic*. Some may have learnt the notation*

$$2.\overline{72} \tag{2.3}$$

for finitely representing *numbers with a* periodic *digit expansion (which are exactly the rationals). The second number, of course, is π, one of the most famous numbers which do* not *belong to the rationals, but to the "rest" of the reals which are not rational (and hence called irrational). Thus it's one example of a "number" which cannot represented by a fraction, resp. in the periodic way as in (2.3).*

Well, fractions may not work out for π (and other irrationals), but still, one may ask, whether π can otherwise be represented finitely. That, however, depends on what actually one accepts as a "finite representation". If one accepts a finite description that describes how to construct *ever closer approximations to π, then there is a finite representation of π. That construction basically is very old (Archimedes), it corresponds to the limits one learns in analysis, and there are computer algorithms, that spit out digits of π as long as you want (of course they can spit them out* all *only if you had infinite time). But the* code *of the algo who does that is finite.*

The bottom line is: it's possible to describe infinite "constructions" in a finite manner, but what exactly *can be captured depends on what precisely is allowed in the description formalism. If only fractions of natural numbers are allowed, one can describe the rationals but not more.*

A final word on the analogy to regular languages. The set of rationals (in, let's say, decimal notation) can be seen as language over the alphabet $\{0, 1, \ldots, 9\ .\}$*, i.e., the decimals and the "decimal point". It's however, a language containing* infinite *words, such as* 2.727272727.... *The syntax* $2.\overline{72}$ *is a* finite *expression but denotes the mentioned infinite word (which is a decimal representation of a rational number). Thus, coming back to the regular languages resp. regular expressions,* $2.\overline{72}$ *is similar to the* Kleene*-star, but* not the same*. If we write* $2.(72)^*$*, we mean the* language *of* finite words

$$\{2, 2.72, 2.727272, \ldots\} .$$

In the same way as one may conveniently define *rational number (when represented in the alphabet of the decimals) as those which can be written using periodic expressions (using for instance* $\overline{\text{overline}}$*),* regular *languages over an alphabet are simply those sets of finite words that can be written by* regular expressions *(see later). Actually, there are deeper connections between regular languages and rational numbers, but it's not the topic of compiler constructions. Suffice to say that it's not a coincidence that* regular *languages are also called* rational *languages (but not in this course).*

Regular expressions

Definition 2.2.3 (Regular expressions)**.** A *regular expression* is one of the following

1. a *basic* regular expression of the form $\mathbf{a}$ (with $a \in \Sigma$), or $\boldsymbol{\epsilon}$, or $\varnothing$
2. an expression of the form $r \mid s$, where r and s are regular expressions.
3. an expression of the form $r\,s$, where r and s are regular expressions.
4. an expression of the form r^*, where r is a regular expression.

Precedence (from high to low): $*$, concatenation, | By "concatenation", the third point in the enumeration is meant. It is written or represented without explicit concatenation operator, just as juxtaposition, like **ab** is the concatenation of the characters **a** and **b**, and also for concatenating whole words: $w_1\, w_2$.

Regular expressions

In [6], $\emptyset$ is not part of the regular expressions. For completeness sake it's included here even if it does not play a practically important role.

In other textbooks, also the notation + instead of | for "alternative" or "choice" is a known convention. The | seems more popular in texts concentrating on *grammars*. Later, we will encounter *context-free* grammars (which can be understood as a generalization of regular expressions) and the |-symbol is consistent with the notation of alternatives in the definition of rules or productions in such grammars. One motivation for using + elsewhere is that one might wish to express "parallel" composition of languages, and a conventional symbol for parallel is |. We will not encounter parallel composition of languages in this course. Also, regular expressions using lot of parentheses and | seems slightly less readable for humans than using +.

Regular expressions are a language in itself, so they have a syntax and a semantics. One could write a lexer (and parser) to parse a regular language. Obviously, tools like parser generators *do* have such a lexer/parser, because their input language are regular expression (and context free grammars, besides syntax to describe further things). One can see regular languages as a domain-specific language for tools like (f)lex (and other purposes).

A "grammatical" definition

Later introduced as (notation for) context-free grammars:

$$\begin{array}{rcl} r & \rightarrow & \mathbf{a} \\ r & \rightarrow & \boldsymbol{\epsilon} \\ r & \rightarrow & \emptyset \\ r & \rightarrow & r \mid r \\ r & \rightarrow & r\,r \\ r & \rightarrow & r^* \end{array} \tag{2.4}$$

Same again

Notational conventions

Later, for CF grammars, we use capital letters to denote "variables" of the grammars (then called *non-terminals*). If we like to be consistent with that convention, the definition looks as follows:

Grammar

$$
\begin{array}{lcl}
R & \rightarrow & \mathbf{a} \\
R & \rightarrow & \boldsymbol{\epsilon} \\
R & \rightarrow & \boldsymbol{\emptyset} \\
R & \rightarrow & R \mid R \\
R & \rightarrow & R\,R \\
R & \rightarrow & R^{*}
\end{array}
\tag{2.5}
$$

Symbols, meta-symbols, meta-meta-symbols . . .

- regexprs: notation or "language" to describe "languages" over a given alphabet Σ (i.e. subsets of Σ^*)
- language being described ⇔ language used to describe the language

⇒ language ⇔ meta-language

- here:
  - regular expressions: notation to describe regular languages
  - English resp. context-free notation: notation to describe regular expressions (a notation itself)
- for now: carefully use *notational convention* for precision

To be careful: we will *later* (when dealing with parsers) distinguish between context-free languages on the one hand and *notations* to denote context-free languages on the other.

In the same manner here: we *now* don't want to confuse regular languages as concept from particular notations (specifically, regular expressions) to write them down.

Notational conventions

- notational conventions by *typographic* means (i.e., different fonts etc.)
- you need good eyes, but: difference between
  - $\boldsymbol{a}$ and a
  - $\boldsymbol{\epsilon}$ and ϵ
  - $\boldsymbol{\emptyset}$ and $\emptyset$
  - $\boldsymbol{|}$ and $|$ (especially hard to see :-))
  - . . .
- later (when gotten used to it) we may take a more "relaxed" attitude towards it, assuming things are clear, as do many textbooks.

Remark 4 (Regular expression syntax)**.** *We are rather careful with notations and meta-notations, especially at the beginning. Note: in compiler* implementations, *the distinction between language and meta-language etc. is very real (even if not done by typographic means as in the script here or textbooks . . .): the programming language being implemented need not be the programming language used to implent that language (the latter would be the "meta-language"). For example in the oblig: the language to implement is called "Compila", and the language used in the implementation will (for most) be Java. Both languages have concepts like "types", "expressions", "statements", which are often quite* *similar. For instance, both languages support an integer type at the user level. But one is an integer type in Compila, the other integers at the meta-level.*

Later, there will be a number of examples using regular expressions. There is a slight "ambiguity" about the way regular expressions are described (in this slides, and elsewhere). It may remain unnoticed (so it's unclear if I should point it out here). On the other had, the lecture is, among other things, about scanning and parsing of syntax, therefore it may be a good idea to reflect on the syntax *of regular expressions itself.*

In the examples shown later, we will use regular expressions using parentheses, like for instance in $\mathbf{b(ab)^*}$. *One question is: are the parentheses* **(** *and* **)** *part of the definition of regular expressions or not? That depends a bit. In the presentation here typically one would not care, one tells the readers that parentheses will be used for disambiguation, and leaves it at that (in the same way one would not bother to tell the reader that it's fine to use "space" between different expressions (like* $a\,|\,b$ *is the same expression as* $a \quad | \quad b$*). Another way of saying that is that textbooks, intended for human readers, give the definition of regular expressions as* abstract syntax *as opposed to* concrete syntax. *Those two concepts will play a prominent role later in the grammar and parsing sections and will become clearer then. Anyway, it's thereby assumed that the reader can interpret parentheses as grouping mechanism, as is common elsewhere, as well, and they are left out from the definition not to clutter it.*

Of course, computers *and programs (i.e., in particular scanners or lexers), are not as good as humans to be educated in "commonly understood" conventions (such as the instruction for the reader that "paretheses are not really part of the regular expressions but can be added for disambiguation".)* Abstract *syntax corresponds to describing the* output *of a parser (which are* abstract syntax trees*). In that view, regular expressions (as all notation represented by abstract syntax) denote* trees. *Since trees in texts are more difficult (and space-consuming) to write, one simply use the usual* linear notation *like the* $\mathbf{b(ab)^*}$ *from above, with parentheses and "conventions" like precedences, to disambiguate the expression. Note that a tree representation represents the grouping of sub-expressions in its structure, so for grouping purposes, parentheses are not needed in abstract syntax.*

Of course, if one wants to implement a lexer or to use one of the available ones, one has to deal with the particular concrete *syntax of the particular scanner. There, of course, characters like* '(' *and* ')' *(or tokens like* `LPAREN` *or* `RPAREN`*) will typically occur.*

To sum up the discussion: Using concepts which will be discussed in more depth later, one may say: whether paretheses are considered as part of the syntax of regular expressions or not depends on the fact whether the definition is wished to be understood as describing concrete *syntax trees or* abstract *syntax trees!*

See also Remark 5 later, which discusses further "ambiguities" in this context.

Same again once more

$$\begin{array}{rcll} R & \rightarrow & \mathbf{a} \mid \boldsymbol{\epsilon} \mid \boldsymbol{\emptyset} & \text{basic reg. expr.} \\ & \mid & R\,|\,R \mid R\,R \mid R^* \mid (R) & \text{compound reg. expr.} \end{array} \tag{2.6}$$

Note:

- symbol $\boldsymbol{|}$: (bold) as symbol of regular expressions
- symbol $|$: (normal, non-bold) meta-symbol of the CF grammar notation
- the meta-notation used here for CF grammars will be the subject of later chapters
- this time: parentheses “added” to the syntax.

Semantics (meaning) of regular expressions

Definition 2.2.4 (Regular expression)**.** Given an alphabet Σ. The meaning of a regexp r (written $\mathcal{L}(r)$) over Σ is given by equation (2.7).

$$\begin{array}{rcll}
\mathcal{L}(\boldsymbol{\varnothing}) & = & \{\} & \text{empty language} \\
\mathcal{L}(\boldsymbol{\epsilon}) & = & \{\epsilon\} & \text{empty word} \\
\mathcal{L}(\boldsymbol{a}) & = & \{a\} & \text{single “letter” from } \Sigma \\
\mathcal{L}(\boldsymbol{r}\boldsymbol{s}) & = & \{w_1 w_2 \mid w_1 \in \mathcal{L}(r), w_2 \in \mathcal{L}(s)\} & \text{concatenation} \\
\mathcal{L}(\boldsymbol{r} \boldsymbol{|} \boldsymbol{s}) & = & \mathcal{L}(r) \cup \mathcal{L}(s) & \text{alternative} \\
\mathcal{L}(\boldsymbol{r}^{\boldsymbol{*}}) & = & \mathcal{L}(r)^* & \text{iteration}
\end{array} \tag{2.7}$$

- conventional *precedences*: $\boldsymbol{*}$, concatenation, $\boldsymbol{|}$.
- Note: left of “=”: reg-expr *syntax*, right of “=”: semantics/meaning/math [10]

Examples

In the following:

- $\Sigma = \{a, b, c\}$.
- we don’t bother to “boldface” the syntax

words with exactly one b	$(a \mid c)^* b (a \mid c)^*$
words with max. one b	$((a \mid c)^*) \mid ((a \mid c)^* b (a \mid c)^*)$ $(a \mid c)^* \ (b \mid \boldsymbol{\epsilon}) \ (a \mid c)^*$
words of the form $a^n b a^n$, i.e., equal number of a’s before and after 1 b	

[10] Sometimes confusingly “the same” notation.

Another regexpr example

words that do not contain two b's in a row.

$$
\begin{array}{lcl}
(b\ (a \mid c))^* & & \text{not quite there yet} \\
((a \mid c)^* \mid (b\ (a \mid c))^*)^* & & \text{better, but still not there} \\
 & = & \text{(simplify)} \\
((a \mid c) \mid (b\ (a \mid c)))^* & = & \text{(simplifiy even more)} \\
(a \mid c \mid ba \mid bc)^* & & \\
(a \mid c \mid ba \mid bc)^*\ (b \mid \epsilon) & & \text{potential } b \text{ at the end} \\
(notb \mid b\ notb)^* (b \mid \epsilon) & & \text{where } notb \triangleq a \mid c
\end{array}
$$

Remark 5 (Regular expressions, disambiguation, and associativity)**.** *Note that in the equations in the example, we silently allowed ourselves some "sloppyness" (at least for the nitpicking mind). The slight ambiguity depends on how we* exactly *interpret definitions of regular expressions. Remember also Remark 4 on page 28, discussing the (non-)status of parentheses in regular expressions. If we think of Definition 2.2.3 on page 26 as describing abstract syntax and a concrete regular expression as representing an abstract syntax tree, then the constructor* $\mid$ *for alternatives is a* binary *constructor. Thus, the regular expression*

$$a \mid c \mid ba \mid bc \tag{2.8}$$

which occurs in the previous example is ambiguous. *What is meant would be one of the following*

$$a \mid (c \mid (ba \mid bc)) \tag{2.9}$$

$$(a \mid c) \mid (ba \mid bc) \tag{2.10}$$

$$((a \mid c) \mid ba) \mid bc\ , \tag{2.11}$$

corresponding to 3 different trees, where occurences of $\mid$ *are inner nodes with two children each, i.e., sub-trees representing subexpressions. In textbooks, one generally does not want to be bothered by writing all the parentheses. There are typically two ways to disambiguate the situation. One is to state (in the text) that the operator, in this case* $\mid$, associates to the left *(alternatively it* associates to the right*). That would mean that the "sloppy" expression without parentheses is meant to represent either (2.9) or (2.11), but not (2.10). If one really wants (2.10), one needs to indicate that using parentheses. Another way of finding an excuse for the sloppyness is to realize that it (in the context of regular expressions)* does not matter*, which of the three trees (2.9) – (2.11) is actually meant. This is specific for the setting here, where the symbol* $\mid$ *is* semantically *represented by* set union $\cup$ *(cf. Definition 2.2.4 on the facing page) which* is *an associative operation on sets. Note that, in principle, one may choose the first option —disambiguation via fixing an associativity— also in situations, where the operator is* not *semantically associative. As illustration, use the '–' symbol with the usal intended meaning of "subtraction" or "one number minus another". Obviously, the expression*

$$5 - 3 - 1 \tag{2.12}$$

now can be interpreted in two semantically different ways, one representing the result 1*, and the other* 3*. As said, one* could *introduce the convention (for instance) that the binary minus-operator associates to the left. In this case, (2.12) represents* $(5-3)-1$.

Whether or not in such a situation one wants symbols to be associative or not is a judgement call (a matter of language pragmatics). On the one hand, disambiguating may make expressions more readable by allowing to omit parenthesis or other syntactic markers which may make the expression or program look cumbersome. On the other, the "light-weight" and "easy-on-the-eye" syntax may trick the unsuspecting programmer into misconceptions about what the program means, if unaware of the rules of associativity and priorities. Disambiguation via associativity rules and priorities is therefore a double-edged sword and should be used carefully. A situation where most would agree associativity is useful and completely unproblematic is the one illustrated for $\mid$ in regular expression: it does not matter anyhow semantically. Decisions concerning when to use ambiguous syntax plus rules how to disambiguate them (or forbid them, or warn the user) occur in many situations in the scannning and parsing phases of a compiler.

Now, the discussion concerning the "ambiguity" of the expression $(a \mid c \mid ba \mid bc)$ from equation (2.8) concentrated on the $\mid$-construct. A similar discussion could obviously be made concerning concatenation (which actually here is not represented by a readable concatenation operator, but just by juxtaposition (= writing expressions side by side)). In the concrete example from (2.8), no ambiguity wrt. concatenation actually occurs, since expressions like ba are not ambiguous, but for longer sequences of concatenation like abc, the question of whether it means $a(bc)$ or $a(bc)$ arises (and again, it's not critical, since concatenation is semantically associative).

Note also that one might think that the expression suffering from an ambiguity concerning combinations of operators, for instance, combinations of $\mid$ and concatenation. For instance, one may wonder if $\boldsymbol{ba} \mid \boldsymbol{bc}$ *could be interpreted as $(ba) \mid (bc)$ and $b(a \mid (bc))$ and $b(a \mid b)c$. However, in Definition 2.2.4 on page 30, we stated* precedences *or priorities, stating that concatenation has a* higher *precedence over $\mid$, meaning that the correct interpretation is $(ba) \mid (bc)$. In a text-book the interpretation is "suggested" to the reader by the typesetting $ba \mid bc$ (and the notation it would be slightly less "helpful" if one would write $ba|bc$... and what about the programmer's version* a␣b|a␣c*?). The situation with precedence is one where difference precedences lead to semantically different interpretations. Even if there's a danger therefore that programmers/readers mis-interpret the real meaning (being unaware of precedences or mixing them up in their head), using precedences in the case of regular expressions certainly is helpful, The alternative of being forced to write, for instance*

$$((a(b(cd))) \mid (b(a(ad)))) \quad \textit{for} \quad abcd \mid baad$$

is not even appealing to hard-core Lisp-programmers (but who knows ...).

A final note: all this discussion about the status of parentheses or left or right assocativity in the interpretation of (for instance mathematical) notation is mostly is over-the-top for most mathematics or other fields where some kind of formal notations or languages are used. There, notation is introduced, perhaps accompanied by sentences like "parentheses or similar will be used when helpful" or "we will allow ourselves to omit parentheses if no confusion may arise", which means, the educated reader is expected to figure it out. Typically, thus, one glosses over too detailed syntactic conventions to proceed to the more interesting and challenging aspects of the subject matter. In such fields one is furthermore sometimes so used to notational traditions ("multiplication binds stronger than addition"), perhaps established since decades or even centuries, that one does not even think about *them consciously. For scanner and parser designers, the situation is different; they are requested to come up with the notational (lexical and syntactical) conventions of perhaps a* new *language, specify them precisely and implement them efficiently. Not only that: at the same time, one aims at a good balance between expliciteness ("Let's just force the programmer to write all the parentheses and grouping explicitly, then he will get less misconceptions of what the program means (and the lexer/parser will be easy to write for me...)") and economy in syntax, leaving many conventions, priorities, etc. implicit without confusing the target programmer.* □

Additional "user-friendly" notations

$$\begin{aligned} r^+ &= rr^* \\ r? &= r \mid \epsilon \end{aligned}$$

Special notations for *sets* of letters:

$$\begin{array}{rl} [0-9] & \text{range (for ordered alphabets)} \\ \sim a & \text{not } a \text{ (everything except } a\text{)} \\ . & \text{all of } \Sigma \end{array}$$

naming regular expressions ("regular definitions")

$$\begin{aligned} \mathit{digit} &= [0-9] \\ \mathit{nat} &= \mathit{digit}^+ \\ \mathit{signedNat} &= (+|-)\mathit{nat} \\ \mathit{number} &= \mathit{signedNat}(\text{"."}\mathit{nat})?(\mathtt{E}\ \mathit{signedNat})? \end{aligned}$$

2.3 DFA

Finite-state automata

- simple "computational" machine
- (variations of) FSA's exist in many flavors and under different names
- other well-known names include finite-state machines, finite labelled transition systems, ...
- "state-and-transition" representations of programs or behaviors (finite state or else) are wide-spread as well
  - state diagrams
  - Kripke-structures
  - I/O automata
  - Moore & Mealy machines
- the logical behavior of certain classes of electronic circuitry with internal memory ("flip-flops") is described by finite-state automata.

Historically, the design of electronic circuitry (not yet chip-based, though) was one of the early very important applications of finite-state machines.

Remark 6 (Finite states)**.** *The distinguishing feature of FSA (as opposed to more powerful automata models such as push-down automata, or Turing-machines), is that they have "* finitely many states *". That sounds clear enough at first sight. But one has too be a bit more careful. First of all, the set of states of the automaton, here called Q, is finite and fixed for a given automaton, all right. But actually, the same is true for pushdown automata and Turing machines! The trick is: if we look at the illustration of the finite-state automaton earlier, where the automaton had a* head*. The picture corresponds to an* accepting *use of an automaton, namely one that is fed by letters on the tape, moving internally from one state to another, as controlled by the different letters (and the automaton's internal "logic", i.e., transitions). Compared to the full power of Turing machines, there are* two *restrictions, things that a finite state automaton cannot do*

- *it moves on one direction only (left-to-right)*
- *it is* read-only.

All non-finite state machines have some additional *memory they can use (besides $q_0, \ldots, q_n \in Q$). Push-down automata for example have additionally a stack, a Turing machine is allowed to* write *freely (= moving not only to the right, but back to the left as well) on the tape, thus using it as external memory.*

FSA

Definition 2.3.1 (FSA)**.** A FSA $\mathcal{A}$ over an alphabet Σ is a tuple $(\Sigma, Q, I, F, \delta)$

- Q: finite set of states
- $I \subseteq Q$, $F \subseteq Q$: initial and final states.
- $\delta \subseteq Q \times \Sigma \times Q$ transition relation

- final states: also called *accepting* states
- transition relation: can *equivalently* be seen as function $\delta : Q \times \Sigma \to 2^Q$: for each state and for each letter, give back the set of sucessor states (which may be empty)
- more suggestive notation: $q_1 \xrightarrow{a} q_2$ for $(q_1, a, q_2) \in \delta$
- we also use freely —self-evident, we hope— things like

$$q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3$$

FSA as scanning machine?

- FSA have slightly unpleasant properties when considering them as decribing an actual program (i.e., a scanner procedure/lexer)
- given the "theoretical definition" of acceptance:

The automaton eats one character after the other, and, when reading a letter, it moves to **a** successor state, if any, of the current state, depending on the character at hand.

- 2 problematic aspects of FSA
  - **non-determinism**: what if there is more than one possible successor state?
  - **undefinedness**: what happens if there's no next state for a given input
- the 2nd one is *easily* repaired, the 1st one requires more thought

- [6]: **recogniser** corresponds to DFA

Non-determinism

Sure, one could try **backtracking**, but, trust us, you don't want that in a scanner. And even if you think it's worth a shot: how do you scan a program directly from magnetic tape, as done in the bad old days? Magnetic tapes can be rewound, of course, but winding them back and forth all the time destroys hardware quickly. How should one scan network traffic, packets etc. on the fly? The network definitely cannot be rewound. Of course, buffering the traffic would be an option and doing then backtracking using the buffered traffic, but maybe the packet-scanning-and-filtering should be done in hardware/firmware, to keep up with today's enormous traffic bandwith. Hardware-only solutions have no dynamic memory, and therefore actually *are* ultimately finite-state machine with no extra memory.

DFA: deterministic automata

Definition 2.3.2 (DFA)**.** A *deterministic, finite automaton* $\mathcal{A}$ (DFA for short) over an alphabet Σ is a tuple $(\Sigma, Q, I, F, \delta)$

- Q: finite set of states
- $I = \{i\} \subseteq Q$, $F \subseteq Q$: initial and final states.
- $\delta : Q \times \Sigma \to Q$ transition function

- transition function: special case of transition relation:
  - deterministic
  - left-total ("complete")

For a relation, being *left-total* means, for each pair q, a from $Q \times \Sigma$, $\delta(q, a)$ is defined. When talking about functions (not relations), it simply means, the function is *total*, not partial.

Some people call an automaton where δ is not a left-total but a determinstic relation (or, equivalently, the function δ is not total, but partial) still a deterministic automaton. In that terminology, the DFA as defined here would be determinstic *and* total.

Meaning of an FSA

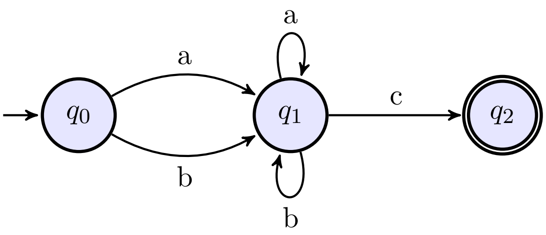
The intended **meaning** of an FSA over an alphabet Σ is the set of all the finite words, the automaton **accepts**.

Definition 2.3.3 (Accepted words and language of an automaton)**.** A word $c_1 c_2 \dots c_n$ with $c_i \in \Sigma$ is *accepted* by automaton $\mathcal{A}$ over Σ, if there exists states $q_0, q_2, \dots, q_n$ from Q such that

$$q_0 \xrightarrow{c_1} q_1 \xrightarrow{c_2} q_2 \xrightarrow{c_3} \quad \dots \quad q_{n-1} \xrightarrow{c_n} q_n \ ,$$

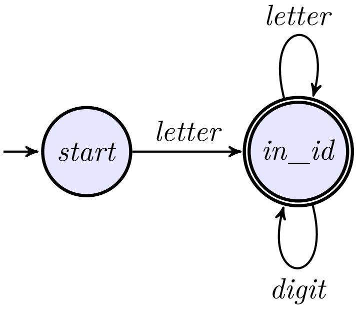
and were $q_0 \in I$ and $q_n \in F$. The *language* of an FSA $\mathcal{A}$, written $\mathcal{L}(\mathcal{A})$, is the set of all words that $\mathcal{A}$ accepts.

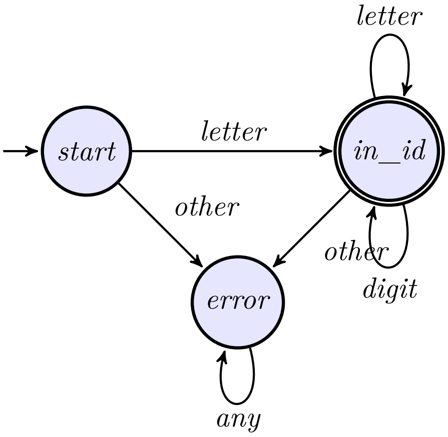
FSA example



Example: identifiers

$$identifier = letter(letter \mid digit)^* \tag{2.13}$$

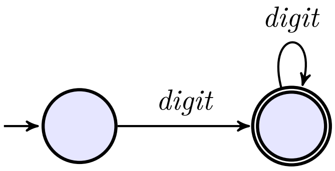




- transition *function*/relation δ *not* completely defined (= *partial* function)

Automata for numbers: natural numbers

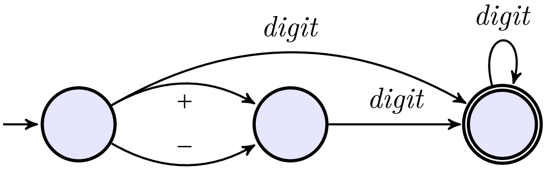
$$\begin{aligned} \mathit{digit} &= [0-9] \\ \mathit{nat} &= \mathit{digit}^{+} \end{aligned} \tag{2.14}$$



One might say, it's not really the natural numbers, it's about a decimal *notation* of natural numbers (as opposed to other notations, for example Roman numeral notation). Note also that initial zeroes are allowed here. It would be easy to disallow that.

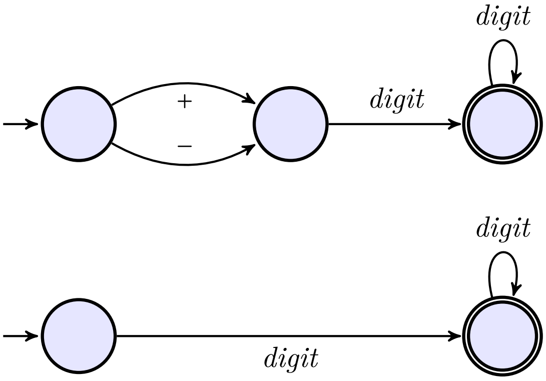
Signed natural numbers

$$\mathit{signednat} = (+\mid-)\mathit{nat} \mid \mathit{nat} \tag{2.15}$$



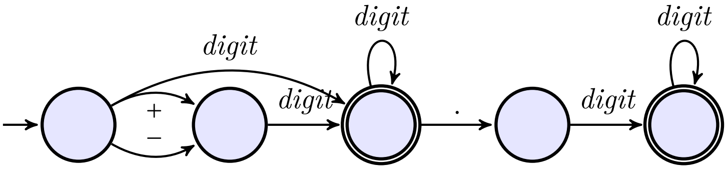
Again, the automaton is *deterministic.* It's easy enough to come up with this automaton, but the *non-deterministic* one is probably more straightforward to come by with. Basically, one informally does two "constructions", the "alternative" which is simply writing "two automata", i.e., one automaton which consists of the union of the two automata, basically. In this example, it therefore has two initial states (which is disallowed obviously for deterministic automata). Another implicit construction is the *"sequential composition"*.

Signed natural numbers: non-deterministic



Fractional numbers

$$frac = signednat(\text{"."}nat)? \tag{2.16}$$

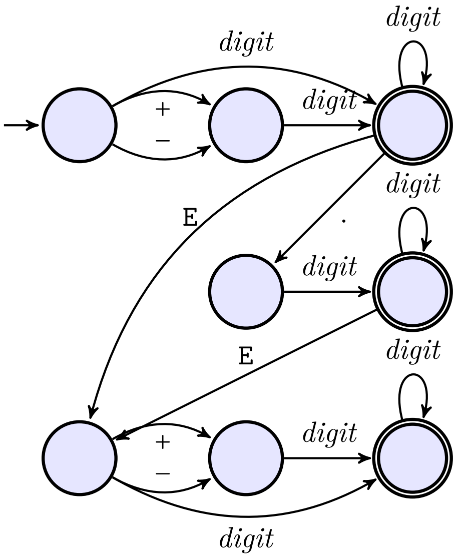


Floats

$$\begin{aligned} digit &= [0-9] \\ nat &= digit^{+} \\ signednat &= (+ \mid -)nat \mid nat \\ frac &= signednat(\text{"."}nat)? \\ float &= frac(\texttt{E}\ signednat)? \end{aligned} \tag{2.17}$$

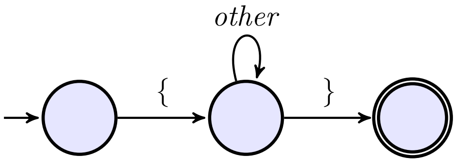
- Note: no (explicit) recursion in the definitions
- note also the treatment of *digit* in the automata.

DFA for floats

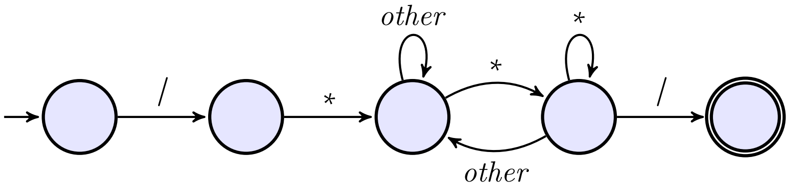


DFAs for comments

Pascal-style



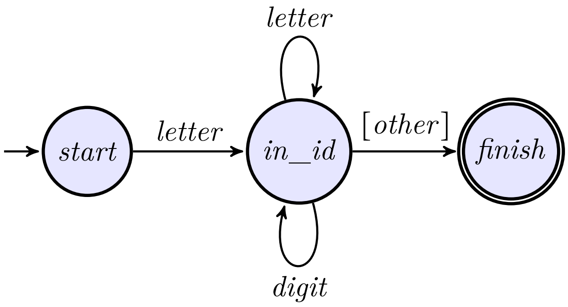
C, C^{++}, Java



2.4 Implementation of DFA

Repeat frame: Example: identifiers

Implementation of DFA (1)



Unlike the previous automaton, this one is *deterministic*, but it's *not* total. The transition function is only *partial.* The "missing" transitions are often not shown (to make the pictures more compact). It is then implicitly assumed, that encountering a character not covered by a transition leads to some extra "error" state (which also is not shown).

The [] around the transition *other* at the end means that the scanner does *not move forward* on the input there (but the automaton proceeds to the accepting state). That is something that is *not* 100% in the "mathematical theory" of FSA, but is how the *implementation* in the scanner will behave. Note also that the accepting state has changed: we have an extra state what we move to by the special kind of transition [*other*]. As the name implies, "other" means all symbols different from the ones already covered by the other outgoing edges. This (in this examples) is used to realized the *longest prefix*: The shown DFA not just accepts "some" identifier it spots on the input, i.e., an arbitrary sequence of letters and digits (starting with a letter). No, it takes as many letters and digits as possible until it encounters a character *not* fitting the specification but not earlier. Only at that point does the automaton accepts (but without advancing the input, in that this character will have to be scanned and classified as the "next chunk" and this "the next automaton".

Implementation of DFA (1): "code"

```
{ starting state }

if  the next character is a letter
then
  advance the input;
  { now in state 2 }
  while the next character is a letter or digit
  do
     advance the input;
     { stay in state 2 }
  end while;
  { go to state 3, without advancing input }
  accept;
else
  { error or other cases }
end
```

Explicit state representation

```
state := 1  { start }
while state = 1 or 2
do
  case state of
  1:  case input character of
      letter: advance the input;
              state := 2
      else state := .... { error or other };
      end case;
  2: case input character of
     letter,digit: advance the input;
                   state := 2; { actually unessessary }
     else          state := 3;
     end case;
  end case;
end while;
if state = 3 then accept else error;
```

Table representation of a DFA

state \ input char	letter	digit	other
1	2		
2	2	2	3
3			

Better table rep. of the DFA

state \ input char	letter	digit	other	accepting
1	2			no
2	2	2	[3]	no
3				yes

added info for

- accepting or not
- " *non-advancing* " transitions
  - here: 3 can be reached from 2 via such a transition

Table-based implementation

```
state := 1  { start }
ch := next input character;
while not Accept[state] and not error(state)
do

while state = 1 or 2
do
  newstate := T[state,ch];
  {if Advance[state,ch]
   then ch:=next input character};
  state := newstate
end while;
if Accept [state] then accept;
```

2.5 NFA

Non-deterministic FSA

Definition 2.5.1 (NFA (with ϵ transitions)). A *non-deterministic* finite-state automaton (NFA for short) $\mathcal{A}$ over an alphabet Σ is a tuple $(\Sigma, Q, I, F, \delta)$, where

- Q: finite set of states
- $I \subseteq Q$, $F \subseteq Q$: initial and final states.
- $\delta : Q \times \Sigma \to 2^Q$ transition function

In case, one uses the alphabet $\Sigma + \{\epsilon\}$, one speaks about an NFA with ϵ-transitions.

- in the following: NFA mostly means, allowing ϵ transitions[11]
- ϵ: treated *differently* than the "normal" letters from Σ.
- δ can *equivalently* be interpreted as *relation*: $\delta \subseteq Q \times \Sigma \times Q$ (transition relation labelled by elements from Σ).

[11] It does not matter much anyhow, as we will see.

Finite state machines

Remark 7 (Terminology (finite state automata))**.** *There are slight variations in the definition of (deterministic resp. non-deterministic) finite-state automata. For instance, some definitions for non-deterministic automata might not use ϵ-transitions, i.e., defined over Σ, not over $\Sigma + \{\epsilon\}$. Another word for FSAs are finite-state machines. Chapter 2 in [9] builds in ϵ-transitions into the definition of NFA, whereas in Definition 2.5.1, we mention that the NFA is not just non-deterministic, but "also" allows those specific transitions. Of course, ϵ-transitions lead to non-determinism, as well, in that they correspond to "spontaneous" transitions, not triggered and determined by input. Thus, in the presence of ϵ-transition, and starting at a given state, a fixed input may* not determine *in which state the automaton ends up in.*

Deterministic or non-deterministic FSA (and many, many variations and extensions thereof) are widely used, *not only for scanning. When discussing scanning, ϵ-transitions come in handy, when translating regular expressions to FSA, that's why [9] directly builds them in.*

Language of an NFA

- remember $\mathcal{L}(\mathcal{A})$ (Definition 2.3.3 on page 35)
- applying definition directly to $\Sigma + \{\epsilon\}$: accepting words "containing" letters ϵ
- as said: *special* treatment for ϵ-transitions/ϵ-"letters". ϵ rather represents absence of input character/letter.

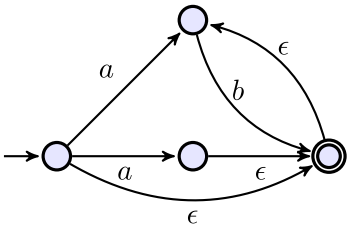
Definition 2.5.2 (Acceptance with ϵ-transitions)**.** A word w over alphabet Σ is *accepted* by an NFA with ϵ-transitions, if there exists a word w' which is accepted by the NFA with alphabet $\Sigma + \{\epsilon\}$ according to Definition 2.3.3 and where w is w' with all occurrences of ϵ removed.

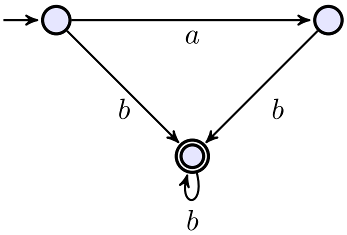
Alternative (but equivalent) intuition

$\mathcal{A}$ reads one character after the other (following its transition relation). If in a state with an outgoing ϵ-transition, $\mathcal{A}$ can move to a corresponding successor state *without* reading an input symbol.

NFA vs. DFA

- *NFA*: often easier (and smaller) to write down, esp. starting from a regular expression
- non-determinism: not *immediately* transferable to an *algo*

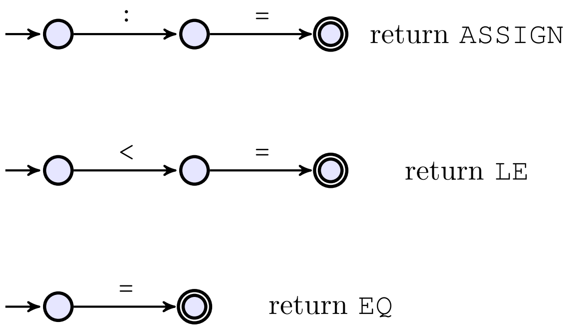




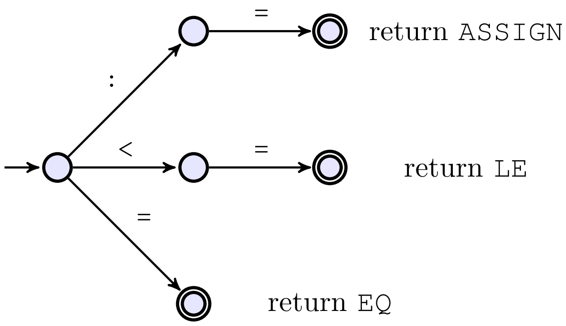
2.6 From regular expressions to NFAs (Thompson's construction)

Why non-deterministic FSA?

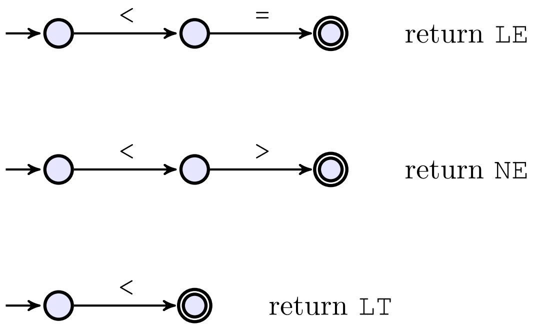
Task: recognize :=, <=, and = as three different tokens:



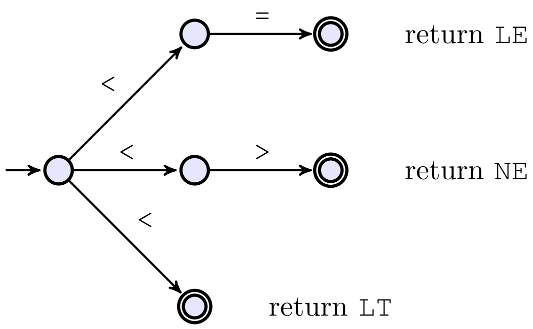
FSA (1-2)



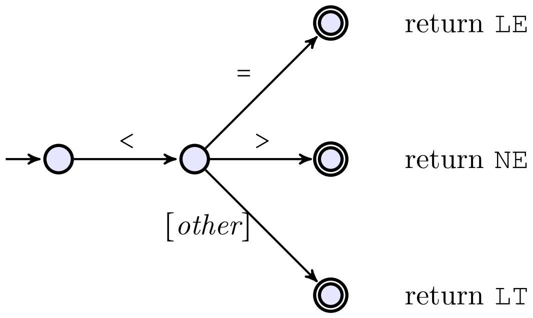
What about the following 3 tokens?



Non-det FSA (2-2)



Non-det FSA (2-3)



Regular expressions → NFA

- needed: a *systematic* translation (= algo, best an efficient one)
- conceptually easiest: translate to NFA (with ϵ-transitions)
  - postpone determinization for a second step
  - (postpone minimization for later, as well)

Compositional construction [11]

Design goal: The NFA of a compound regular expression is given by taking the NFA of the immediate subexpressions and connecting them appropriately.

Compositionality

- construction slightly[12] simpler, if one uses automata with one start and one accepting state

⇒ ample use of ϵ-transitions

Compositionality

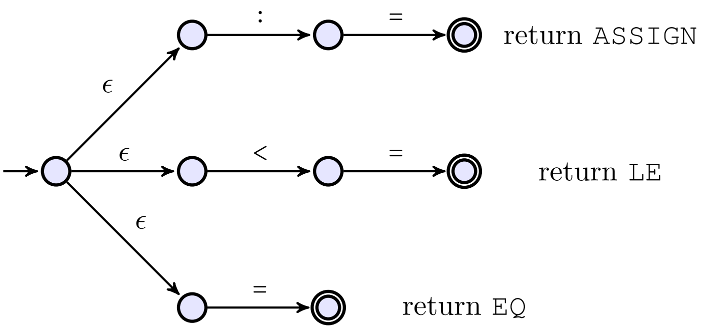
Remark 8 (Compositionality)**.** *Compositional concepts (definitions, constructions, analyses, translations, . . .) are* immensely *important and pervasive in compiler techniques (and beyond). One example already encountered was the definition of the language of a regular expression (see Definition 2.2.4 on page 30). The design goal of a compositional translation here is the underlying reason why to base the construction on* non-deterministic *machines.*

Compositionality is also of practical importance ("component-based software"). In connection with compilers, separate compilation *and (static / dynamic) linking (i.e. "composing") of separately compiled "units" of code is a crucial feature of modern programming languages/compilers. Separately compilable units may vary, sometimes they are called modules or similarly. Part of the success of C was its support for separate compilation (and tools like* `make` *that helps organizing the (re-)compilation process). For fairness sake, C was by far not the first major language supporting separate compilation, for instance FORTRAN II allowed that, as well, back in 1958.*

Btw., Ken Thompson, the guy who first described the regexpr-to-NFA construction discussed here, is one of the key figures behind the UNIX operating system and thus also the C language (both went hand in hand). Not suprisingly, considering the material of this section, he is also the author of the grep -tool ("globally search a regular expression and print"). He got the Turing-award (and many other honors) for his contributions. □

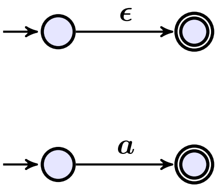
[12] It does not matter much, though.

Illustration for ϵ-transitions



Thompson's construction: basic expressions

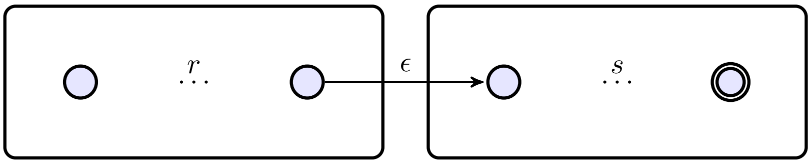
basic (= non-composed) regular expressions: $\boldsymbol{\epsilon}$, $\varnothing$, $\boldsymbol{a}$ (for all $a \in \Sigma$)

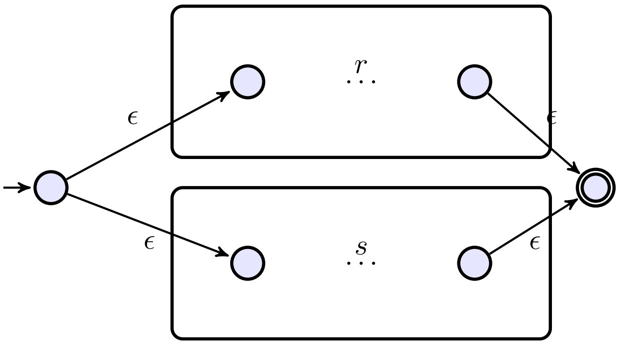


The $\varnothing$ is slightly odd: it's sometimes not part of regular expressions. We can see it as represented as the empty automaton (which has no states and which therefore was not drawn pictorially). If it's lacking, then one cannot express the empty language, obviously. That's not nice, because then the regular languages are not closed under complement. Also: obviously, there exists an automaton with an empty language. Therefore, $\varnothing$ should be part of the regular expressions, even if practically it does not play much of a role.

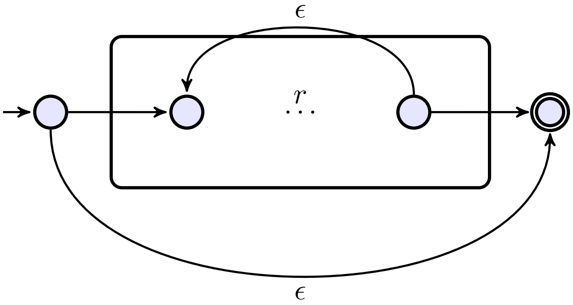
The representation of $\varnothing$ as empty automaton is ok. If we do that, however, it's not the case that in Thompson's construction all automata have one start and one final state, the empty automaton would be an exception.

Thompson's construction: compound expressions

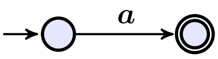


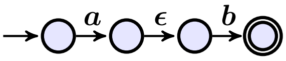


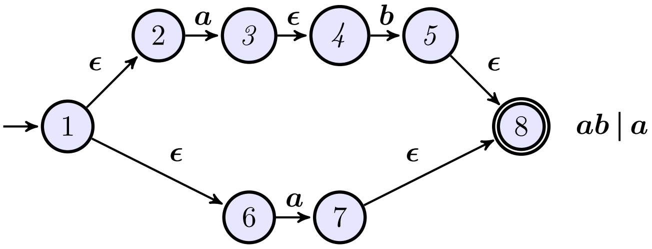
Thompson's construction: compound expressions: iteration



Example: $ab \mid a$







2.7 Determinization

Determinization: the subset construction

Main idea

- Given a non-det. automaton $\mathcal{A}$. To construct a DFA $\overline{\mathcal{A}}$: instead of *backtracking*: explore all successors "at the same time" ⇒
- each state q' in $\overline{\mathcal{A}}$: represents a *subset* of states from $\mathcal{A}$
- Given a word w: "feeding" that to $\overline{\mathcal{A}}$ leads to *the* state representing *all* states of $\mathcal{A}$ *reachable* via w

- side remark: this construction, known also as *powerset* construction, seems straightforward enough, but: analogous constructions works for some other kinds of automata, as well, but for others, the approach does *not* work.[13]
- origin of the construction: Rabin and Scott [10]

Some notation/definitions

Definition 2.7.1 (ϵ-closure, a-successors)**.** Given a state q, the *ϵ-closure* of q, written $close_{\epsilon}(a)$, is the set of states reachable via zero, one, or more ϵ-transitions. We write q_a for the set of states, reachable from q with one a-transition. Both definitions are used analogously for sets of states.

[13] For some forms of automata, non-deterministic versions are strictly more expressive than the deterministic one.

ϵ-closure

Remark 9 (ϵ-closure)**.** *[9] does not sketch an algorithm but it should be clear that the ϵ-closure is easily implementable for a given state, resp. a given finite set of states. Some textbooks also write λ instead of ϵ, and consequently speak of λ-closure. And in still other contexts (mainly not in language theory and recognizers), silent transitions are marked with τ.*

It may be obvious but: the set of states in the ϵ-closure of a given state are not "language-equivalent". However, the union of languages for all states from the ϵ-closure corresponds to the language accepted with the given state as initial one. However, the language being accepted is not the property which is relevant here in the determinization. The ϵ-closure is needed to capture the set of all states reachable by a given word. But again, the exact characterization of the set need to be done carefully. The states in the set are also not equivalent *wrt. their reachability information: Obviously, states in the ϵ-closure of a given state may be reached by* more *words. The set of reaching words for a given state, however, is* not *in general the* intersection *of the sets of corresponding words of the states in the closure.*

□

It may also be worth remarking: later, when it comes to parsing, there will be similarly the phenomenon that some derivation steps done in a grammar (not in an automaton) will be done "eating symbols" (in the context, those symbols will be called "terminals" or "terminal symbols". That may pose problems for parsing (for some forms of parsing more than for others). Such a situation can be compared with the treatment of "ϵs" in the construction of a parser-automaton (there also called ϵ-closure).

Transformation process: sketch of the algo

Input: NFA $\mathcal{A}$ over a given Σ

Output: DFA $\overline{\mathcal{A}}$

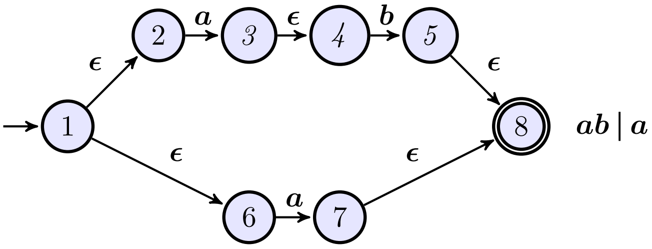
1. the *initial* state: $close_\epsilon(I)$, where I are the initial states of $\mathcal{A}$
2. for a state Q in $\overline{\mathcal{A}}$: the *a-successor* of Q is given by $close_\epsilon(Q_a)$, i.e.,

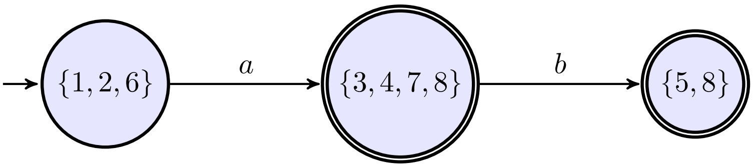
$$Q \xrightarrow{a} close_\epsilon(Q_a) \tag{2.18}$$

3. repeat step 2 for all states in $\overline{\mathcal{A}}$ and all $a \in \Sigma$, until no more states are being added
4. the *accepting* states in $\overline{\mathcal{A}}$: those containing *at least* one accepting state of $\mathcal{A}$

Note: Cooper and Torczon [6]: slightly more "concrete" formulation using a work-list.

Example $ab \mid a$

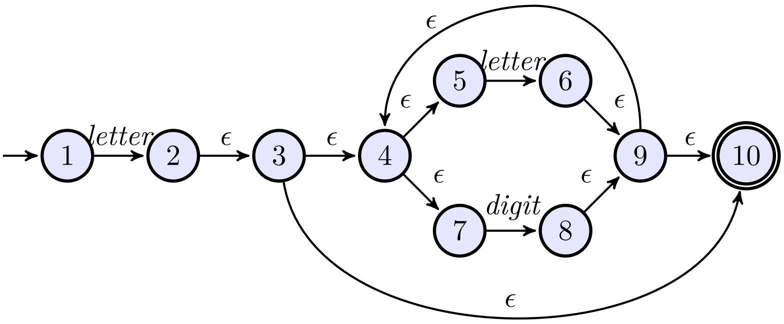




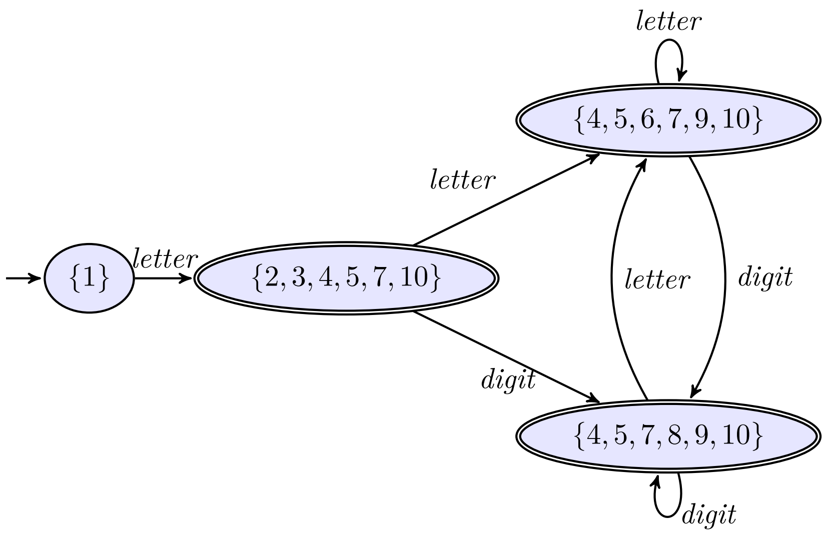
$ab \mid a$

Example: identifiers

Remember: regexpr for identifies from equation (2.13)



Identifiers: DFA



2.8 Minimization

Minimization

- automatic construction of DFA (via e.g. Thompson): often many superfluous states
- goal: "combine" states of a DFA without changing the accepted language

1. Properties of the minimization algo

   Canonicity: all DFA for the same language are transformed to the *same* DFA

   Minimality: resulting DFA has *minimal* number of states
2. Remarks
   - "side effects": answers to *equivalence* problems
     - given 2 DFA: do they accept the same language?
     - given 2 regular expressions, do they describe the same language?
   - modern version: Hopcroft [7].

Hopcroft's partition refinement algo for minimization

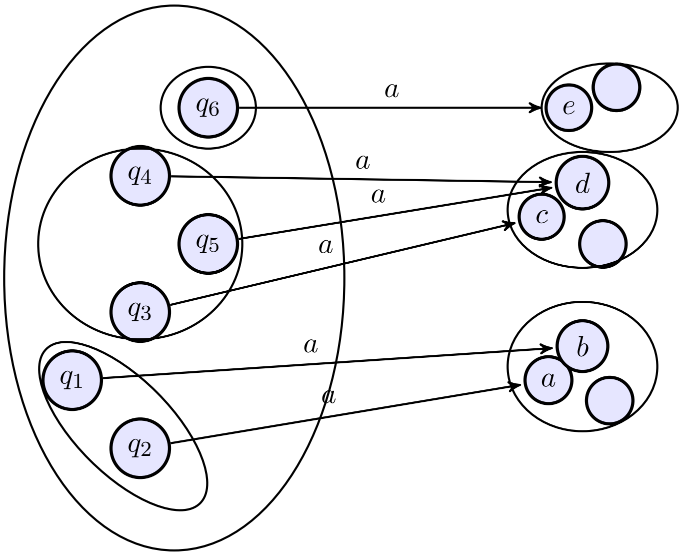
- starting point: *complete* DFA (i.e., *error*-state possibly needed)
- first idea: *equivalent* states in the given DFA may be *identified*
- **equivalent**: when used as starting point, accepting the same language
- **partition refinement:**
  - works "the other way around"
  - instead of collapsing equivalent states:
    - ∗ start by "collapsing as much as possible" and then,
    - ∗ iteratively, detect *non-equivalent* states, and then *split* a "collapsed" state
    - ∗ stop when no violations of "equivalence" are detected

- *partitioning* of a set (of states):
- *worklist*: data structure of to keep non-treated classes, termination if worklist is empty

Partition refinement: a bit more concrete

- **Initial** partitioning: 2 partitions: set containing all *accepting* states F, set containing all *non-accepting* states $Q \backslash F$
- **Loop** do the following: pick a current equivalence class Q_i and a symbol a
  - if for all $q \in Q_i$, $\delta(q, a)$ is member of the *same* class Q_j ⇒ consider Q_i as done (for now)
  - else:
    - ∗ **split** Q_i into $Q_i^1, \ldots Q_i^k$ s.t. the above situation is repaired for each Q_i^l (but don't split more than necessary).
    - ∗ be aware: a split may have a "cascading effect": other classes being fine before the split of Q_i need to be reconsidered ⇒ *worklist* algo
- **stop** if the situation stabilizes, i.e., no more split happens (= worklist empty, at latest if back to the original DFA)

Split in partition refinement: basic step

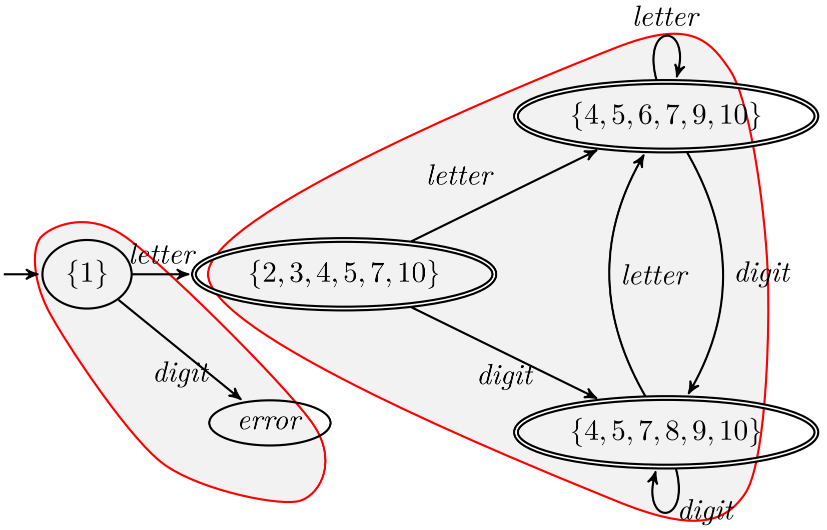


- before the split $\{q_1, q_2, \ldots, q_6\}$
- after the split on a: $\{q_1, q_2\}, \{q_3, q_4, q_5\}, \{q_6\}$

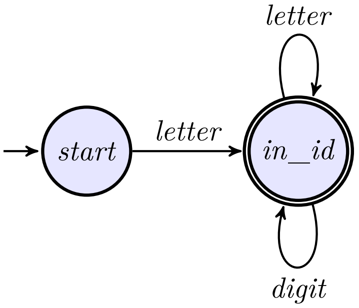
1. Note
   The pic shows only one letter a, in general one has to do the same construction for all letters of the alphabet.

Again: DFA for identifiers

Completed automaton

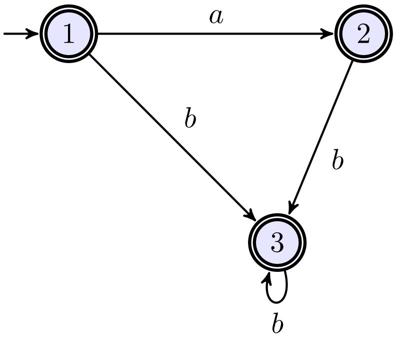


Minimized automaton (error state omitted)



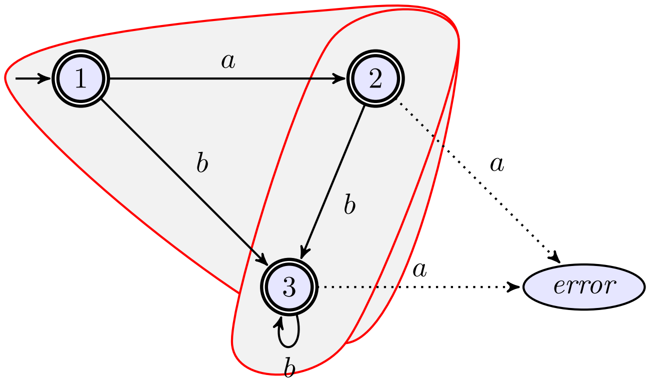
Another example: partition refinement & error state

$$(\boldsymbol{a} \mid \boldsymbol{\epsilon})\boldsymbol{b}^* \tag{2.19}$$

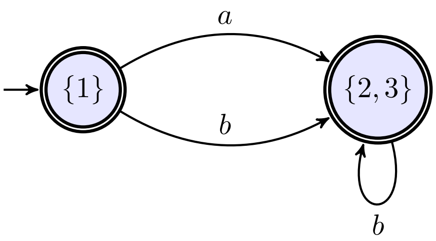


Partition refinement

error state added initial partitioning split after a



End result (error state omitted again)



2.9 Scanner implementations and scanner generation tools

This last section contains only rather superficial remarks concerning how to implement as scanner or lexer. A few more details can be found in [6, Section 2.5]. The oblig will include the implementation of a lexer/scanner.

Tools for generating scanners

- scanners: simple and well-understood part of compiler
- hand-coding possible
- mostly better off with: generated scanner
- standard tools **lex** / **flex** (also in combination with *parser* generators, like **yacc** / **bison**
- variants exist for many implementing languages
- based on the results of this section

Main idea of (f)lex and similar

- output of lexer/scanner = input for parser
- programmer specifies regular expressions for each token-class and corresponding actions[14] (and whitespace, comments etc.)
- the spec. language offers some conveniences (extended regexpr with priorities, associativities etc) to ease the task
- automatically translated to NFA (e.g. Thompson)
- then made into a deterministic DFA ("subset construction")
- minimized (with a little care to keep the token classes separate)
- implement the DFA (usually with the help of a *table* representation)

Sample flex file (excerpt)

```

DIGIT      [0-9]
ID         [a-z][a-z0-9]*

%%

{DIGIT}+    {
            printf( "An integer: %s (%d)\n", yytext,
                    atoi( yytext ) );
            }

{DIGIT}+"."{DIGIT}*          {
            printf( "A float: %s (%g)\n", yytext,
                    atof( yytext ) );
            }

if|then|begin|end|procedure|function         {
            printf( "A keyword: %s\n", yytext );
            }
```

[14] Tokens and actions of a parser will be covered later. For example, identifiers and digits as described but the reg. expressions, would end up in two different token classes, where the actual string of characters (also known as *lexeme*) being the value of the token attribute.

Chapter
Grammars

What is it about?

Learning Targets of this Chapter

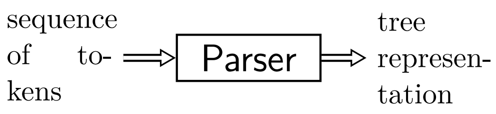
1. (context-free) grammars + BNF
2. ambiguity and other properties
3. terminology: tokens, lexemes
4. different trees connected to grammars/parsing
5. derivations, sentential forms

The chapter corresponds to [6, Section 3.1–3.2] (or [9, Chapter 3]).

Contents

3.1 Introduction

Bird’s eye view of a parser



- *check* that the token sequence correspond to a *syntactically correct* program
  - if yes: yield *tree* as intermediate representation for subsequent phases
  - if not: give *understandable* error message(s)
- we will encounter various kinds of trees
  - derivation trees (derivation in a (context-free) grammar)
  - *parse* tree, *concrete syntax tree*
  - *abstract syntax trees*
- mentioned tree forms hang together, dividing line a bit fuzzy
- result of a parser: typically AST

(Context-free) grammars

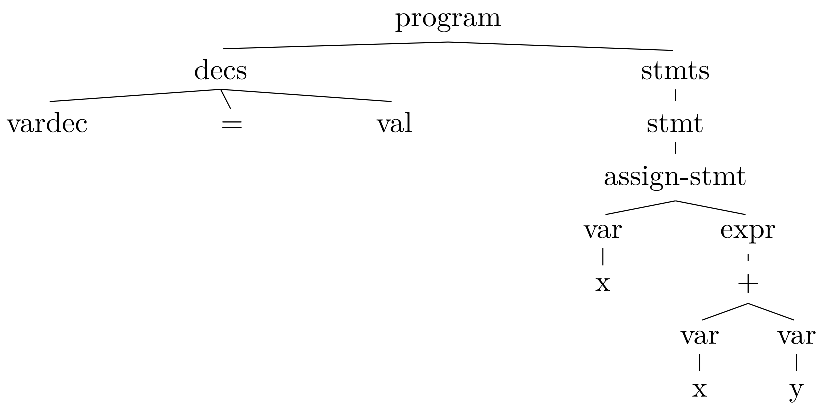
- specifies the *syntactic structure* of a language
- here: grammar means CFG
- G **derives** word w

Parsing

Given a stream of "symbols" w and a grammar G, find a *derivation* from G that produces w

The slide talks about deriving "words". In general, words are finite sequences of symbols from a given alphabet (as was the case for regular languages). In the concrete picture of a parser, the words are sequences of *tokens*, which are the elements that come out of the scanner. A successful derivation leads to tree-like representations. There a various slightly different forms of trees connected with grammars and parsing, which we will later see in more detail; for a start now, we will just illustrate such tree-like structures, without distinguishing between (abstract) syntax trees and parse trees.

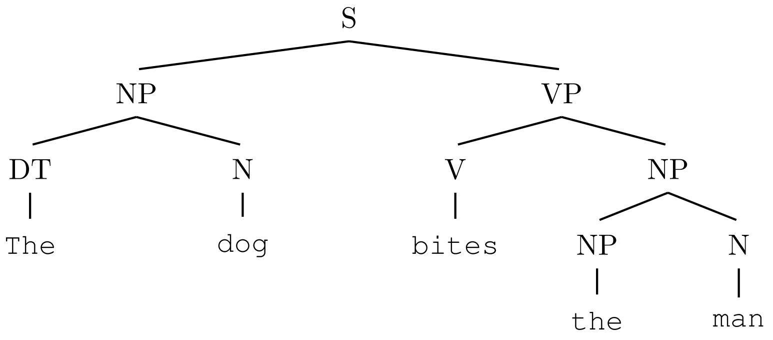
Sample syntax tree



Syntax tree

The displayed syntax tree is meant "impressionistic" rather then formal. Neither is it a sample syntax tree of a real programming language, nor do we want to illustrate for instance special features of an *abstract* syntax tree vs.\ a *concrete* syntax tree (or a parse tree). Those notions are closely related and corresponding trees might all looks similar to the tree shown. There might, however, be subtle conceptual and representational differences in the various classes of trees. Those are not relevant yet, at the beginning of the section.

Natural-language parse tree



"Interface" between scanner and parser

- remember: task of scanner = "chopping up" the input char stream (throw away white space, etc.) and *classify* the pieces (1 piece = *lexeme*)
- classified lexeme = **token**
- sometimes we use ⟨`integer`, "42"⟩
  - `integer`: "class" or "type" of the token, also called *token name*
  - "42" : *value of the token attribute* (or just value). Here: directly the *lexeme* (a string or sequence of chars)
- a note on (sloppyness/ease of) terminology: often: the token name is simply just called the token
- for (context-free) grammars: the *token (symbol)* corrresponds there to **terminal symbols** (or terminals, for short)

Token names and terminals

Remark 10 (Token (names) and terminals)**.** *We said, that sometimes one uses the name "token" just to mean token symbol, ignoring its value (like "42" from above). Especially, in the conceptual discussion and treatment of context-free grammars, which form the core of the specifications of a parser, the token value* is basically irrelevant*. Therefore, one simply identifies "tokens = terminals of the grammar" and silently ignores the presence of the values. In an implementation, and in lexer/parser generators, the value* "42" *of an integer-representing token must obviously* not *be forgotten, though . . . The grammar may be the core of the specification of the syntactical analysis, but the result of the scanner, which resulted in the lexeme* "42" *must nevertheless not be thrown away, it's only not really part of the parser's tasks.*

Notations

Remark 11. *Writing a compiler, especially a compiler front-end comprising a scanner and a parser, but to a lesser extent also for later phases, is about implementing representation of syntactic structures. The slides here don't* implement *a lexer or a parser or similar, but* describe *in a hopefully unambiguous way the principles of how a compiler front end works and is implemented. To describe that, one needs "language" as well, such as English language (mostly for intuitions) but also "mathematical" notations such as regular expressions, or in this section, context-free grammars. Those mathematical definitions have* themselves *a particular* syntax*. One can see them as formal* domain-specific languages *to describe (other) languages. One faces therefore the (unavoidable) fact that one deals with two levels of languages: the language that is described (or at least whose* syntax *is described) and the language used to descibe that language. The situation is, of course, when writing a book teaching a human language: there is a language being taught, and a language used for teaching (both may be different). More closely, it's analogous when implementing a general purpose programming language: there is the language used to implement the compiler on the one hand, and the language for which the compiler is written for. For instance, one may choose to implement a C^{++}-compiler in C. It may increase the confusion, if one chooses to write a C compiler in C .... Anyhow, the language for describing (or implementing) the language of interest is called the* meta-language*, and the other one described therefore just "the language".*

When writing texts or slides about such syntactic issues, typically one wants to make clear to the reader what is meant. One standard way are typographic *conventions, i.e., using specific typographic fonts. I am stressing "nowadays" because in classic texts in compiler construction, sometimes the typographic choices were limited (maybe written as "typoscript", i.e., as "manuscript" on a type writer).*

3.2 Context-free grammars and BNF notation

Grammars

- in this chapter(s): focus on **context-free grammars**
- thus here: grammar = CFG
- as in the context of regular expressions/languages: *language* = (typically infinite) set of words
- **grammar** = formalism to unambiguously specify a language
- intended language: all **syntactically correct** programs of a given progamming language

Slogan

A CFG describes the syntax of a programming language. [1]

[1] And some say, regular expressions describe its microsyntax.

Note: a compiler might reject some syntactically correct programs, whose violations *cannot* be captured by CFGs. That is done by *subsequent* phases. For instance, the type checker may reject syntactically correct programs that are ill-typed. The type checker is an important part from the *semantic* phase (or *static analysis* phase). A typing discipline is *not* a syntactic property of a language (in that it cannot captured most commonly by a context-free grammar), it's therefore a "semantics" property.

Remarks on grammars

Sometimes, the word "grammar" is synonymously for context-free grammars, as CFGs are so central. However, the concept of grammars is more general; there exists context-sensitive and Turing-expressive grammars, both more expressive than CFGs. Also a restricted class of CFG correspond to regular expressions/languages. Seen as a grammar, regular expressions correspond so-called left-linear grammars (or alternativelty, right-linear grammars), which are a special form of context-free grammars.

Context-free grammar

Definition 3.2.1 (CFG)**.** A *context-free grammar* G is a 4-tuple $G = (\Sigma_T, \Sigma_N, S, P)$:

1. 2 disjoint finite alphabets of terminals Σ_T and
2. non-terminals Σ_N
3. 1 start-symbol $S \in \Sigma_N$ (a non-terminal)
4. productions P = finite subset of $\Sigma_N \times (\Sigma_N + \Sigma_T)^*$

- terminal symbols: corresponds to tokens in parser = basic building blocks of syntax
- non-terminals: (e.g. "expression", "while-loop", "method-definition" ...)
- grammar: generating (via "derivations") languages
- **parsing**: the *inverse* problem

⇒ CFG = specification

Further notions

- sentence and sentential form
- productions (or rules)
- derivation
- *language* of a grammar $\mathcal{L}(G)$
- parse tree

Those notions will be explained with the help of examples.

BNF notation

- popular & common format to write CFGs, i.e., describe context-free languages
- named after *pioneering* (seriously) work on Algol 60
- notation to write productions/rules + some extra meta-symbols for convenience and grouping

Slogan: Backus-Naur form

What regular expressions are for regular languages is BNF for context-free languages.

"Expressions" in BNF

$$
\begin{array}{rcl}
\mathit{exp} & \rightarrow & \mathit{exp}\ \mathit{op}\ \mathit{exp} \mid \mathbf{(}\,\mathit{exp}\,\mathbf{)} \mid \mathbf{number} \\
\mathit{op} & \rightarrow & \mathtt{+} \mid \mathtt{-} \mid \mathtt{*}
\end{array}
\tag{3.1}
$$

- "→" indicating productions and " | " indicating alternatives [2]
- convention: terminals written **boldface**, non-terminals *italic*
- also simple math symbols like "+" and "(″ are meant above as terminals
- start symbol here: *exp*
- remember: terminals like **number** correspond to tokens, resp.\ token classes. The attributes/token values are not relevant here.

Terminals

Conventions are not always 100% followed, often bold fonts for symbols such as + or (are unavailable or not easily visible. The alternative using, for instance, boldface "identifiers" like **PLUS** and **LPAREN** looks ugly. Some books would write '+' and '('.

In a concrete parser implementation, in an object-oriented setting, one might choose to implement terminals as classes (resp. concrete terminals as instances of classes). In that case, a class name + is typically not available and the class might be named `Plus`. Later we will have a look at how to systematically implement terminals and non-terminals, and having a class `Plus` for a non-terminal '+' etc. is a systematic way of doing it (maybe not the most efficient one available though).

Most texts don't follow conventions so slavishly and hope for an intuitive understanding by the educated reader, that + is a terminal in a grammar, as it's not a non-terminal, which are written here in *italics*.

[2] The grammar consists of 6 productions/rules, 3 for *expr* and 3 for *op*, the | is just for convenience. Side remark: Often also ::= is used for →.

Different notations

- BNF: notationally not 100% "standardized" across books/tools
- "classic" way (Algol 60):

```
<exp> ::=  <exp> <op> <exp>
        |  ( <exp> )
        | NUMBER
<op>  ::=  + | − | ∗
```

- Extended BNF (EBNF) and yet another style

$$\begin{array}{rcl} exp & \rightarrow & exp \; (\; "+" \mid "-" \mid "*" \;) \; exp \\ & \mid & "(" \, exp \, ")" \mid "number" \end{array} \tag{3.2}$$

- note: parentheses as terminals vs. as *metasymbols*

"Standard" BNF

Specific and unambiguous notation is important, in particular if you *implement* a concrete language on a computer. On the other hand: understanding the underlying concepts by *humans* is equally important. In that way, bureaucratically fixed notations may distract from the core, which is *understanding* the principles. XML, anyone? Most textbooks (and we) rely on simple typographic conventions (boldfaces, italics). For "implementations" of BNF specification (as in tools like yacc), the notations, based mostly on ASCII, cannot rely on such typographic conventions.

Syntax of BNF

BNF and its variations is a notation to describe "languages", more precisely the "syntax" of context-free languages. Of course, BNF notation, when exactly defined, is a language in itself, namely a domain-specific language to describe context-free languages. It may be instructive to write a grammar for BNF in BNF, i.e., using BNF as meta-language to describe BNF notation (or regular expressions). Is it possible to use regular expressions as meta-language to describe regular expression?

Different ways of writing the same grammar

- directly written as 6 pairs (6 rules, 6 productions) from $\Sigma_N \times (\Sigma_N \cup \Sigma_T)^*$, with "→" as nice looking "separator":

$$\begin{array}{rcl} exp & \rightarrow & exp \; op \; exp \\ exp & \rightarrow & \mathbf{(} \; exp \; \mathbf{)} \\ exp & \rightarrow & \mathbf{number} \\ op & \rightarrow & \mathbf{+} \\ op & \rightarrow & \mathbf{-} \\ op & \rightarrow & \mathbf{*} \end{array} \tag{3.3}$$

- choice of non-terminals: irrelevant (except for human readability):

$$\begin{array}{lcl} E & \rightarrow & E\,O\,E \mid \mathbf{(}\,E\,\mathbf{)} \mid \textbf{number} \\ O & \rightarrow & + \mid - \mid * \end{array} \tag{3.4}$$

- still: we count 6 productions

Grammars as language generators

Deriving a word:

Start from start symbol. Pick a "matching" rule to rewrite the current word to a new one; repeat until *terminal* symbols, only.

- *non-deterministic* process
- rewrite relation for derivations:
  - one step rewriting: $w_1 \Rightarrow w_2$
  - one step using rule n: $w_1 \Rightarrow_n w_2$
  - many steps: $\Rightarrow^*$, etc.

Non-determinism means, that the process of derivation allows choices to be made, when applying a production. One can distinguish 2 forms of non-determinism here: 1) a sentential form contains (most often) more than one non-terminal. In that situation, one has the choice of expanding one non-terminal or the other. 2) Besides that, there may be more than one production or rule for a given non-terminal. Again, one has a choice.

As far as 1) is concerned. whether one expands one symbol or the other leads to different derivations, but won't lead to different *derivation trees* or *parse trees* in the end. Below, we impose a fixed discipline on *where* to expand. That leads to *left-most* or *right-most* derivations.

Language of grammar G

$$\mathcal{L}(G) = \{s \mid \mathit{start} \Rightarrow^* s \text{ and } s \in \Sigma_T^*\}$$

Example derivation for $(\textbf{number}-\textbf{number})*\textbf{number}$

$$\begin{array}{rcl} \underline{\mathit{exp}} & \Rightarrow & \underline{\mathit{exp}}\ \mathit{op}\ \mathit{exp} \\ & \Rightarrow & (\underline{\mathit{exp}})\ \mathit{op}\ \mathit{exp} \\ & \Rightarrow & (\underline{\mathit{exp}}\ \mathit{op}\ \mathit{exp})\ \mathit{op}\ \mathit{exp} \\ & \Rightarrow & (\mathbf{n}\ \underline{\mathit{op}}\ \mathit{exp})\ \mathit{op}\ \mathit{exp} \\ & \Rightarrow & (\mathbf{n}-\underline{\mathit{exp}})\ \mathit{op}\ \mathit{exp} \\ & \Rightarrow & (\mathbf{n}-\mathbf{n})\underline{\mathit{op}}\ \mathit{exp} \\ & \Rightarrow & (\mathbf{n}-\mathbf{n})*\underline{\mathit{exp}} \\ & \Rightarrow & (\mathbf{n}-\mathbf{n})*\mathbf{n} \end{array}$$

- underline the "place" where a rule is used, i.e., an *occurrence* of the non-terminal symbol is being rewritten/expanded
- here: *leftmost* derivation[3]

Rightmost derivation

$$
\begin{array}{rcl}
\underline{exp} & \Rightarrow & exp\ op\ \underline{exp} \\
 & \Rightarrow & exp\ \underline{op}\ \mathbf{n} \\
 & \Rightarrow & \underline{exp}{*}\mathbf{n} \\
 & \Rightarrow & (exp\ op\ \underline{exp}){*}\mathbf{n} \\
 & \Rightarrow & (exp\ \underline{op}\ \mathbf{n}){*}\mathbf{n} \\
 & \Rightarrow & (\underline{exp}{-}\mathbf{n}){*}\mathbf{n} \\
 & \Rightarrow & (\mathbf{n}{-}\mathbf{n}){*}\mathbf{n}
\end{array}
$$

- other ("mixed") derivations for the same word possible

Some easy requirements for reasonable grammars

- all symbols (terminals and non-terminals): should occur in a some word derivable from the start symbol
- words containing only non-terminals should be derivable
- an example of a silly grammar G (start-symbol A)

$$
\begin{array}{rcl}
A & \to & B\mathbf{x} \\
B & \to & A\mathbf{y} \\
C & \to & \mathbf{z}
\end{array}
$$

- $\mathcal{L}(G) = \varnothing$
- those "sanitary conditions": minimal "common sense" requirements

Remark 12. *There can be further conditions one would like to impose on grammars besides the one sketched. A CFG that derives ultimately only 1 word of terminals (or a finite set of those) does not make much sense either. There are further conditions on grammar characterizing their usefulness for* parsing*. So far, we mentioned just some obvious conditions of "useless" grammars or "defects" in a grammer (like superfluous symbols). "Usefulness conditions" may refer to the use of ϵ-productions and other situations. Those conditions will be discussed when the lecture covers* parsing *(not just grammars).*

Remark 13 ("Easy" sanitary conditions for CFGs)**.** *We stated a few conditions to avoid grammars which technically qualify as CFGs but don't make much sense, for instance to avoid that the grammar is obviously empty; there are easier ways to describe an empty set ...*

There's a catch, though: it might not immediately be obvious that, for a given G, the question $\mathcal{L}(G) \overset{?}{=} \varnothing$ is decidable!

[3] We'll come back to that later, it will be important.

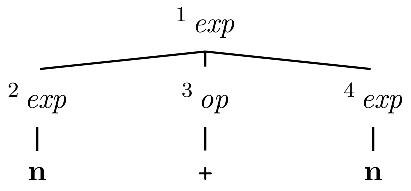
Whether a regular expression describes the empty language is trivially decidable. Whether or not a finite state automaton descibes the empty language or not is, if not trivial, then at least a very easily decidable question. For context-sensitive *grammars (which are more expressive than CFG but not yet Turing complete), the emptyness question turns out to be undecidable. Also, other interesting questions concerning CFGs are, in fact, undecidable, like: given two CFGs, do they describe the same language? Or: given a CFG, does it actually describe a regular language? Most disturbingly perhaps: given a grammar, it's undecidable whether the grammar is ambiguous or not. So there are interesting and relevant properties concerning CFGs which are undecidable. Why that is, is not part of the pensum of this lecture (but we will at least have to deal with the important concept of grammatical ambiguity later). Coming back for the initial question: fortunately, the emptyness problem for CFGs* is *decidable.*

Questions concerning decidability may seem not too relevant at first sight. Even if some grammars can be constructed to demonstrate difficult questions, for instance related to decidability or worst-case complexity, the designer of a language will not intentionally try to achieve an obscure set of rules whose status is unclear, but hopefully strive to capture in a clear manner the syntactic principles of an equally hopefully clearly structured language. Nonetheless: grammars for real languages may become large and complex, and, even if conceptually clear, may contain unexpected bugs which makes them behave unexpectedly (for instance caused by a simple typo in one of the many rules).

In general, the implementor of a parser will often rely on automatic tools ("parser generators") which take as an input a CFG and turns it in into an implementation of a recognizer, which does the syntactic analysis. Such tools obviously can reliably and accurately help the implementor of the parser automatically only for problems which are decidable. *For undecidable problems, one could still achieve things automatically, provided one would compromise by not insisting that the parser always terminates (but that's generally is seen as unacceptable), or at the price of* approximative *answers. It should also be mentioned that parser generators typcially won't tackle CFGs in their* full generality *but are tailor-made for well-defined and well-understood subclasses thereof, where efficient recognizers are automaticlly generatable. In the part about parsing, we will cover some such classes.*

Parse tree

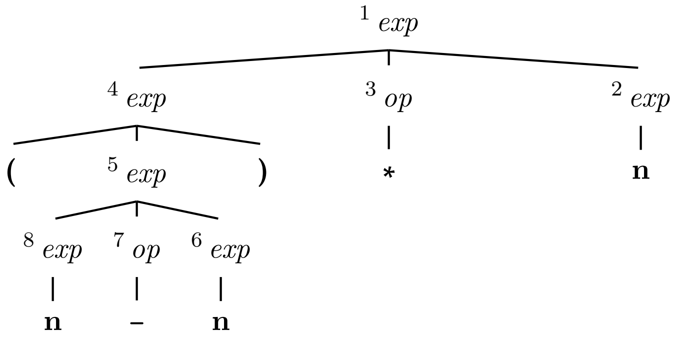
- derivation: if viewed as sequence of steps ⇒ linear "structure"
- order of individual steps: irrelevant
- ⇒ order not needed for subsequent steps
- **parse tree**: structure for the *essence* of derivation
- also called *concrete* syntax tree.[4]



[4] There will be *abstract* syntax trees, as well.

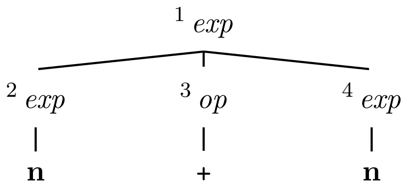
- numbers in the tree
  - *not* part of the parse tree, indicate order of derivation, only
  - here: leftmost derivation

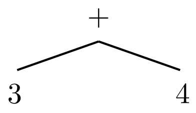
Another parse tree (numbers for rightmost derivation)



Abstract syntax tree

- parse tree: contains still unnecessary details
- specifically: *parentheses* or similar, used for grouping
- tree-structure: can express the intended grouping already
- remember: tokens contain also attribute values (e.g.: full token for token class **n** may contain lexeme like "42" ...)





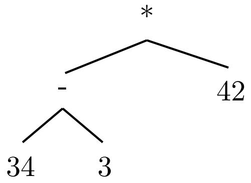
AST vs. CST

- **parse tree**
  - important *conceptual* structure, to talk about grammars and derivations
  - most likely *not explicitly implemented* in a parser
- **AST** is a *concrete* data structure
  - important IR of the syntax (for the language being implemented)
  - written in the meta-language
  - therefore: nodes like + and 3 *are no longer (necessarily and directly) tokens or lexemes*
  - concrete data stuctures in the meta-language (C-structs, instances of Java classes, or what suits best)
  - the figure is meant schematic, only
  - produced by the parser, used by later phases

- note also: we use 3 in the AST, where lexeme was `"3"`

⇒ at some point, the lexeme *string* (for numbers) is translated to a *number* in the meta-language (typically already by the lexer)

Plausible schematic AST (for the other parse tree)



- this AST: rather "simplified" version of the CST
- an AST closer to the CST (just dropping the parentheses): in principle nothing "wrong" with it either

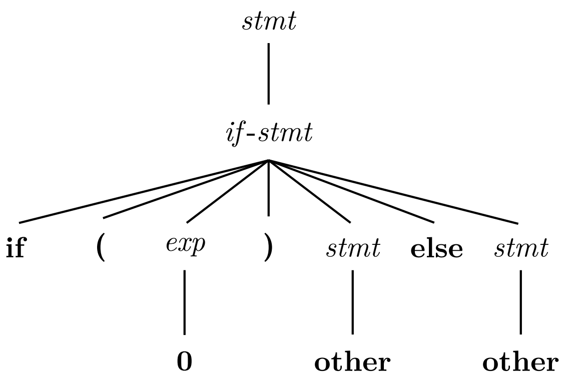
Conditionals

Conditionals G_1

$$\begin{aligned} \textit{stmt} &\rightarrow \textit{if-stmt} \mid \textbf{other} \\ \textit{if-stmt} &\rightarrow \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt} \\ &\mid\ \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\ \textit{exp} &\rightarrow \textbf{0} \mid \textbf{1} \end{aligned} \tag{3.5}$$

Parse tree

if (0) other else other



Another grammar for conditionals

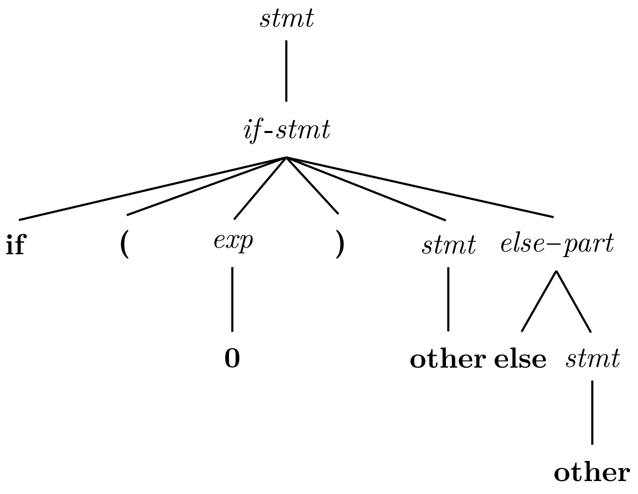
Conditionals G_2

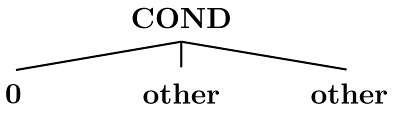
$$\begin{aligned}
\mathit{stmt} &\rightarrow \mathit{if\text{-}stmt} \mid \textbf{other} \\
\mathit{if\text{-}stmt} &\rightarrow \textbf{if}\ \textbf{(}\ \mathit{exp}\ \textbf{)}\ \mathit{stmt}\ \mathit{else{-}part} \\
\mathit{else{-}part} &\rightarrow \textbf{else}\ \mathit{stmt} \mid \epsilon \\
\mathit{exp} &\rightarrow \textbf{0} \mid \textbf{1}
\end{aligned} \tag{3.6}$$

Abbreviation

ϵ = empty word

A further parse tree + an AST





A potentially missing else part may be represented by null-"pointers" in languages like Java

3.3 Ambiguity

Before we mentioned some "easy" conditions to avoid "silly" grammars, without going into it. *Ambiguity* is more important and complex. Roughly speaking, a grammar is ambiguous, if there exist *sentences for which there are two different parse trees.* That's in general highly undesirable, as it means there are sentences with different syntactic interpretations (which therefore may ultimately interpreted differently). That is generally a no-no, but even *if* one would accept such a language definition, parsing would be problematic, as it would involve *backtracking* trying out different possible interpretations during parsing (which would also be a no-no for reasons of efficiency) In fact, later, when dealing with actual concrete parsing procedures, they cover certain *specific* forms of CFG (with names like LL(1), LR(1), etc.), which are in particular non-ambiguous. To say it differently: the fact that a grammar is parseable by some, say, LL(1) top-down parser (which does not do backtracking) implies directly that the grammar is unambiguous. Similar for the other classes we'll cover.

Note also: given an ambiguous grammar, it is often possible to find a *different* "equivalent" grammar that *is* unambiguous. Even if such reformulations are often possible, it's not guaranteed: there are context-free languages which do have an ambiguous grammar, but no unambigous one. In that case, one speaks of an ambiguous context-free language. We concentrate on *ambiguity* of grammars.

Tempus fugit . . .

picture source: wikipedia

Ambiguous grammar

Definition 3.3.1 (Ambiguous grammar)**.** A grammar is *ambiguous* if there exists a word with *two different* parse trees.

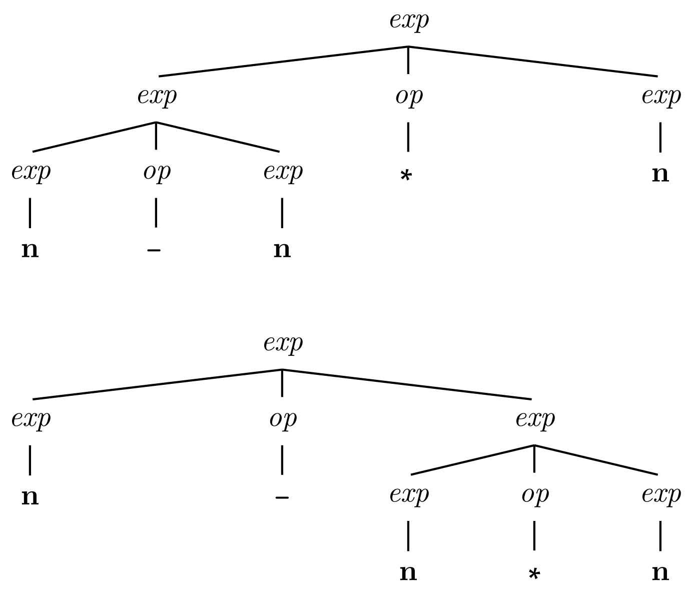
Remember grammar from equation (3.1):

$$\begin{array}{rcl} exp & \rightarrow & exp\ \ op\ \ exp \mid \mathbf{(}\,exp\,\mathbf{)} \mid \mathbf{number} \\ op & \rightarrow & \mathbf{+} \mid \mathbf{-} \mid \mathbf{*} \end{array}$$

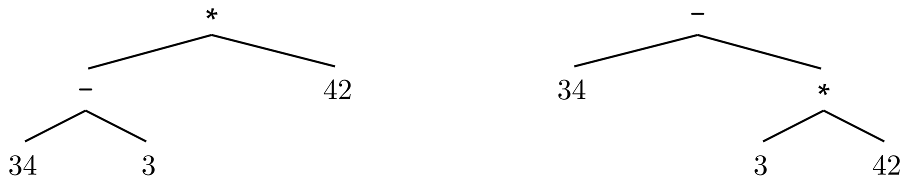
Consider:

$$\mathbf{n - n * n}$$

2 CTS's



2 resulting ASTs



different parse trees ⇒ different[5] ASTs ⇒ different[5] meaning

Side remark: different meaning

The issue of "different meaning" may in practice be subtle: is $(x + y) - z$ the same as $x + (y - z)$? In principle yes, but what about MAXINT ?

Precendence & associativity

- one way to make a grammar unambiguous (or less ambiguous)
- for instance:

binary op's	precedence	associativity
$+, -$	low	left
$\times, /$	higher	left
$\uparrow$	highest	right

[5] At least in many cases.

- $a \uparrow b$ written in standard math as a^b:

$$\begin{aligned} &5 + 3/5 \times 2 + 4 \uparrow 2 \uparrow 3 && = \\ &5 + 3/5 \times 2 + 4^{2^3} && = \\ &(5 + ((3/5 \times 2)) + (4^{(2^3)})) \ . \end{aligned}$$

- mostly fine for *binary* ops, but usually also for unary ones (postfix or prefix)

Unambiguity without imposing explicit associativity and precedence

- removing ambiguity by reformulating the grammar
- **precedence** for op's: *precedence cascade*
  - some bind stronger than others (`*` more than `+`)
  - introduce separate *non-terminal* for each precedence level (here: terms and factors)

Expressions, revisited

- *associativity*
  - *left*-assoc: write the corresponding rules in *left-recursive* manner, e.g.:

$$exp \to exp\ addop\ term \mid term$$

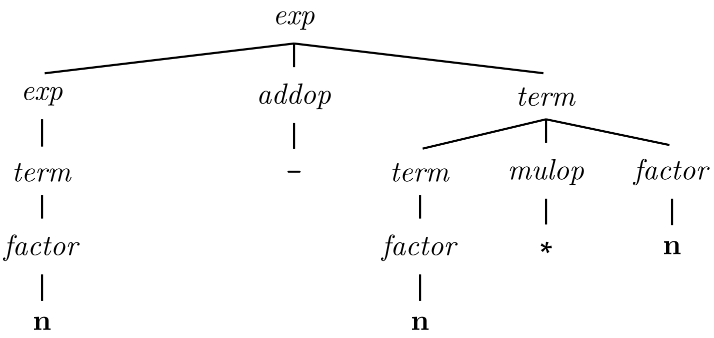
  - *right*-assoc: analogous, but right-recursive
  - *non*-assoc:

$$exp \to term\ addop\ term \mid term$$

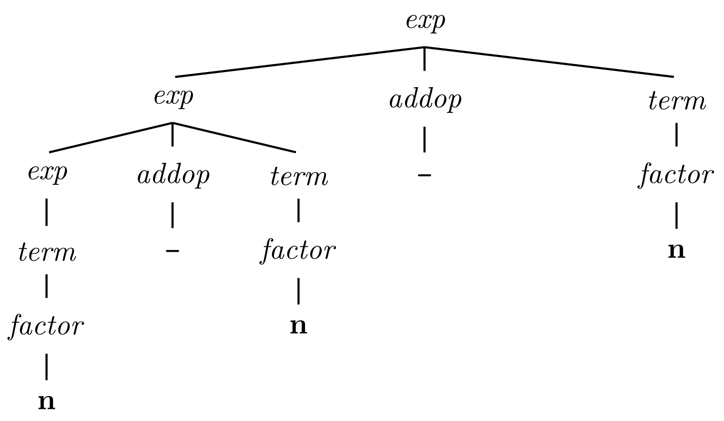
factors and terms

$$\begin{aligned} exp &\to exp\ \ addop\ \ term \mid term \\ addop &\to \texttt{+} \mid \texttt{-} \\ term &\to term\ \ mulop\ \ factor \mid factor \\ mulop &\to \texttt{*} \\ factor &\to \textbf{(}\ exp\ \textbf{)} \mid \textbf{number} \end{aligned} \tag{3.7}$$

$34 - 3 * 42$



34 – 3 – 42



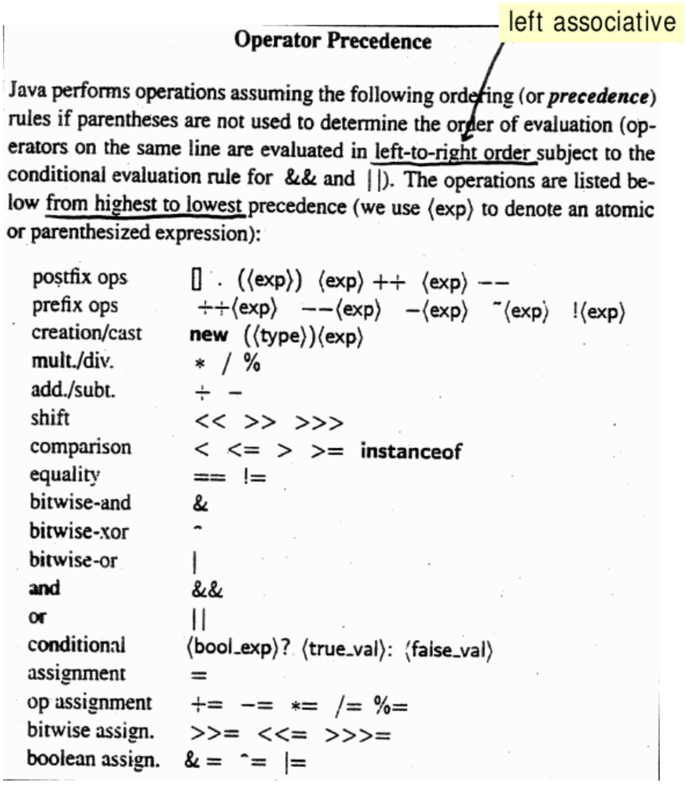
Ambiguity

As mentioned, the question whether a given CFG is ambiguous or not is *undecidable*. Note also: if one uses a parser generator, such as yacc or bison (which cover a practically usefull subset of CFGs), the resulting recognizer is *always* deterministic. In case the construction encounters ambiguous situations, they are "resolved" by making a specific choice. Nonetheless, such ambiguities indicate often that the formulation of the grammar (or even the language it defines) has problematic aspects. Most programmers as "users" of a programming language may not read the full BNF definition, most will try to grasp the language looking at sample code pieces mentioned in the manual, etc. And even if they bother studying the exact specification of the system, i.e., the full grammar, ambiguities are *not* obvious (after all, it's undecidable, at least the problem in general). Hidden ambiguities, "resolved" by the generated parser, may lead to misconceptions as to what a program actually means. It's similar to the situation, when one tries to study a book with arithmetic being unaware that multiplication binds stronger than addition. Without being aware of that, some sections won't make much sense. A parser implementing such grammars may make consistent choices, but the programmer using the compiler may not be aware of them. At least the compiler writer, responsible for designing the language, will be informed about "*conflicts*" in the grammar and a careful designer will try to get rid of them. This may be done by adding associativities and precedences (when appropriate) or reformulating the grammar, or even reconsider the syntax of the language. While ambiguities and conflicts are generally a bad sign, arbitrarily adding a complicated "precedence order" and "associativities" on all kinds of symbols or complicate the grammar adding ever more separate classes of nonterminals just to make the conflicts go away is not a real solution either. Chances are, that those parser-internal "tricks" will be lost on the programmer as user of the language, as well. Sometimes, making the *language* simpler (as opposed to complicate the grammar for the same language) might be the better choice. That can typically be done by making the language more verbose and reducing "overloading" of syntax. Of course, going overboard by making groupings etc.\ of all constructs crystal clear to the parser, may also lead to non-elegant designs. Lisp is a standard example, notoriously known for its extensive use of parentheses. Basically, the programmer directly writes down *syntax trees*, which certainly removes ambiguities, but still, mountains of parentheses are also not the easiest syntax for human consumption (for most humans, at least). So it's a balance (and at least partly a matter of taste, as for most design choices and questions of language pragmatics).

But in general: if it's enormously complex to come up with a reasonably unambigous grammar for an intended language, chances are, that reading programs in that language and intutively grasping what is intended may be hard for humans, too.

Note also: since already the question, whether a given CFG is ambiguous or not is undecidable, it should be clear, that the following question is undecidable, as well: given a grammar, can I reformulate it, still accepting the same language, that it becomes unambiguous?

Real life example



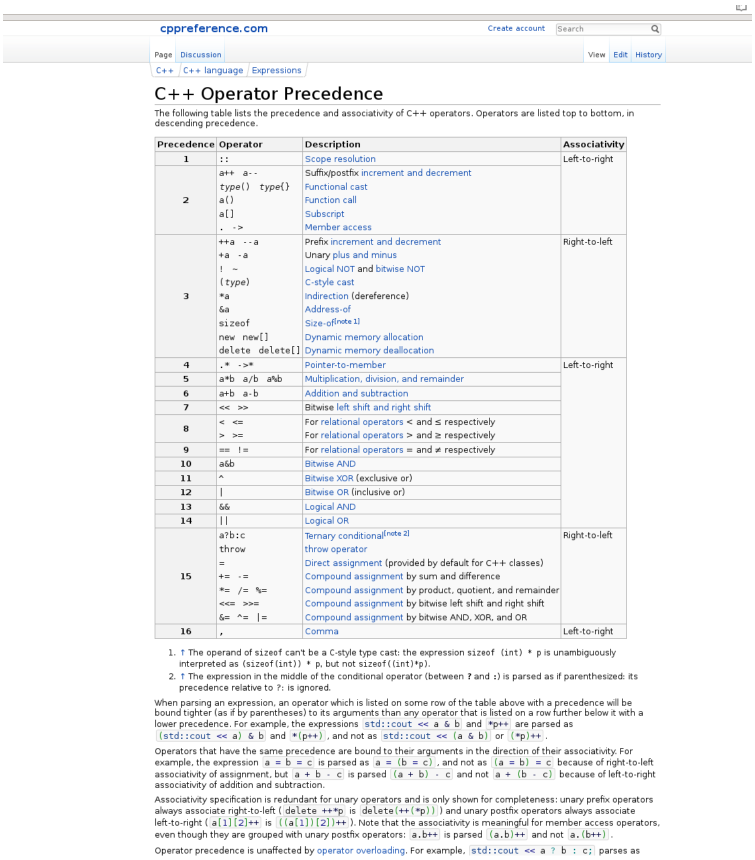
left associative

Operator Precedence

Java performs operations assuming the following ordering (or ***precedence***) rules if parentheses are not used to determine the order of evaluation (operators on the same line are evaluated in left-to-right order subject to the conditional evaluation rule for && and ||). The operations are listed below from highest to lowest precedence (we use ⟨exp⟩ to denote an atomic or parenthesized expression):

postfix ops	[] . (⟨exp⟩) ⟨exp⟩++ ⟨exp⟩−−
prefix ops	++⟨exp⟩ −−⟨exp⟩ −⟨exp⟩ ˜⟨exp⟩ !⟨exp⟩
creation/cast	**new** (⟨type⟩)⟨exp⟩
mult./div.	* / %
add./subt.	+ −
shift	<< >> >>>
comparison	< <= > >= **instanceof**
equality	== !=
bitwise-and	&
bitwise-xor	ˆ
bitwise-or	|
and	&&
or	||
conditional	⟨bool_exp⟩? ⟨true_val⟩: ⟨false_val⟩
assignment	=
op assignment	+= −= *= /= %=
bitwise assign.	>>= <<= >>>=
boolean assign.	&= ˆ= |=

Another example



C++ Operator Precedence

The following table lists the precedence and associativity of C++ operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right

2	`a++ a--`
	type`()` *type*`{}`
	`a()`
	`a[]`
	`. ->`
		Suffix/postfix increment and decrement
		Functional cast
		Function call
		Subscript
		Member access	

3	`++a --a`
	`+a -a`
	`! ~`
	`(`*type*`)`
	`*a`
	`&a`
	`sizeof`
	`new new[]`
	`delete delete[]`
		Prefix increment and decrement
		Unary plus and minus
		Logical NOT and bitwise NOT
		C-style cast
		Indirection (dereference)
		Address-of
		Size-of[note 1]
		Dynamic memory allocation
		Dynamic memory deallocation
			Right-to-left

4	`.* ->*`	Pointer-to-member	Left-to-right
5	`a*b a/b a%b`	Multiplication, division, and remainder	
6	`a+b a-b`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	

8	`< <=`
	`> >=`
		For relational operators < and ≤ respectively
		For relational operators > and ≥ respectively	

9	`== !=`	For relational operators = and ≠ respectively	
10	`a&b`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	

15	`a?b:c`
	`throw`
	`=`
	`+= -=`
	`*= /= %=`
	`<<= >>=`
	`&= ^= |=`
		Ternary conditional[note 2]
		throw operator
		Direct assignment (provided by default for C++ classes)
		Compound assignment by sum and difference
		Compound assignment by product, quotient, and remainder
		Compound assignment by bitwise left shift and right shift
		Compound assignment by bitwise AND, XOR, and OR
			Right-to-left

16	`,`	Comma	Left-to-right

1. ↑ The operand of `sizeof` can't be a C-style type cast: the expression `sizeof (int) * p` is unambiguously interpreted as `(sizeof(int)) * p`, but not `sizeof((int)*p)`.
2. ↑ The expression in the middle of the conditional operator (between **?** and **:**) is parsed as if parenthesized: its precedence relative to `?:` is ignored.

When parsing an expression, an operator which is listed on some row of the table above with a precedence will be bound tighter (as if by parentheses) to its arguments than any operator that is listed on a row further below it with a lower precedence. For example, the expressions `std::cout << a & b` and `*p++` are parsed as `(std::cout << a) & b` and `*(p++)`, and not as `std::cout << (a & b)` or `(*p)++`.

Operators that have the same precedence are bound to their arguments in the direction of their associativity. For example, the expression `a = b = c` is parsed as `a = (b = c)`, and not as `(a = b) = c` because of right-to-left associativity of assignment, but `a + b - c` is parsed `(a + b) - c` and not `a + (b - c)` because of left-to-right associativity of addition and subtraction.

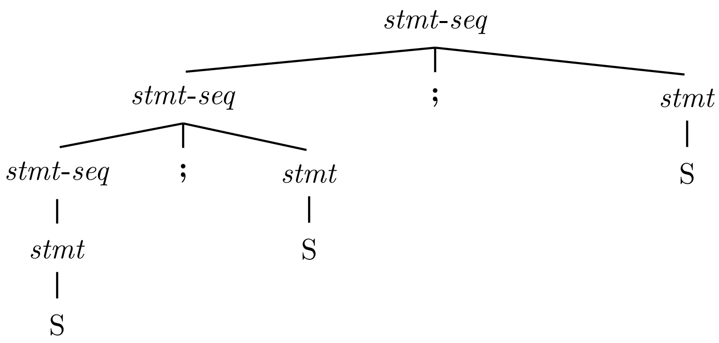
Associativity specification is redundant for unary operators and is only shown for completeness: unary prefix operators always associate right-to-left (`delete ++*p` is `delete(++(*p))`) and unary postfix operators always associate left-to-right (`a[1][2]++` is `((a[1])[2])++`). Note that the associativity is meaningful for member access operators, even though they are grouped with unary postfix operators: `a.b++` is parsed `(a.b)++` and not `a.(b++)`.

Operator precedence is unaffected by operator overloading. For example, `std::cout << a ? b : c;` parses as

Non-essential ambiguity

left-assoc

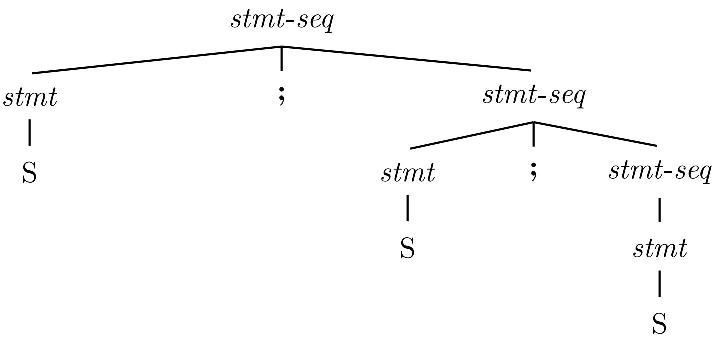
$$\begin{aligned} \mathit{stmt\text{-}seq} &\rightarrow \mathit{stmt\text{-}seq}\,\mathbf{;}\,\mathit{stmt} \mid \mathit{stmt} \\ \mathit{stmt} &\rightarrow S \end{aligned}$$



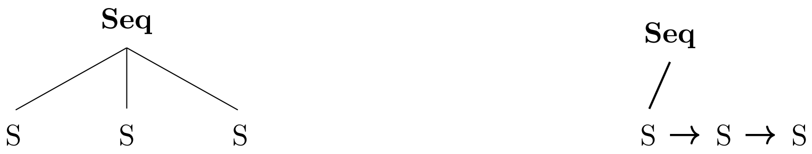
Non-essential ambiguity (2)

right-assoc representation instead

$$
\begin{array}{rcl}
\mathit{stmt\text{-}seq} & \rightarrow & \mathit{stmt}\,;\mathit{stmt\text{-}seq} \;\mid\; \mathit{stmt} \\
\mathit{stmt} & \rightarrow & S
\end{array}
$$



Possible AST representations



Dangling else

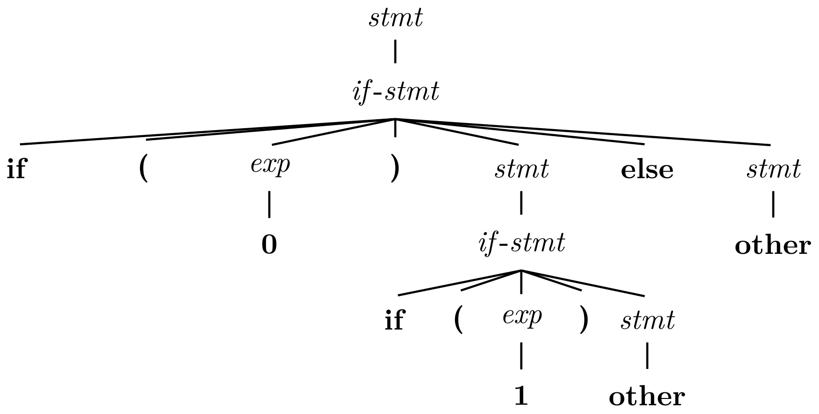
Nested if's

$$
\mathbf{if}\,(\,\mathbf{0}\,)\,\mathbf{if}\,(\,\mathbf{1}\,)\,\mathbf{other}\ \mathbf{else}\ \mathbf{other}
$$

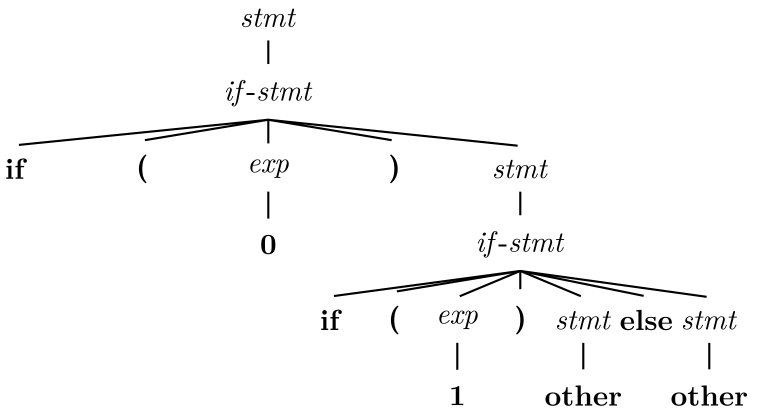
Remember grammar from equation (3.5):

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{if\text{-}stmt} \;\mid\; \mathbf{other} \\
\mathit{if\text{-}stmt} & \rightarrow & \mathbf{if}\,(\,\mathit{exp}\,)\,\mathit{stmt} \\
 & \mid & \mathbf{if}\,(\,\mathit{exp}\,)\,\mathit{stmt}\,\mathbf{else}\,\mathit{stmt} \\
\mathit{exp} & \rightarrow & \mathbf{0} \;\mid\; \mathbf{1}
\end{array}
$$

Should it be like this . . .



. . . or like this



- common convention: connect **else** to closest "free" (= dangling) occurrence

Unambiguous grammar

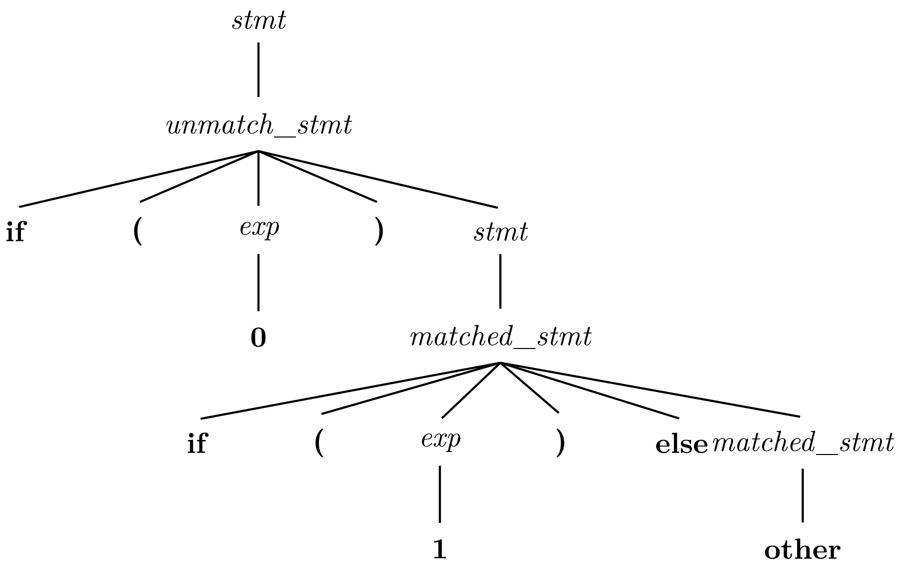
Grammar

$$
\begin{array}{rcl}
\mathit{stmt} & \to & \mathit{matched\_stmt} \mid \mathit{unmatch\_stmt} \\
\mathit{matched\_stmt} & \to & \textbf{if}\ \textbf{(}\ \mathit{exp}\ \textbf{)}\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{matched\_stmt} \\
 & \mid & \textbf{other} \\
\mathit{unmatch\_stmt} & \to & \textbf{if}\ \textbf{(}\ \mathit{exp}\ \textbf{)}\ \mathit{stmt} \\
 & \mid & \textbf{if}\ \textbf{(}\ \mathit{exp}\ \textbf{)}\ \mathit{matched\_stmt}\ \textbf{else}\ \mathit{unmatch\_stmt} \\
\mathit{exp} & \to & \textbf{0} \mid \textbf{1}
\end{array}
$$

- never have an unmatched statement inside a matched one
- complex grammar, seldomly used
- instead: ambiguous one, with extra "rule": connect each **else** to closest free **if**
- alternative: *different* syntax, e.g.,

  - *mandatory* **else**,
  - or require **endif**

CST



Adding sugar: extended BNF

- make CFG-notation more "convenient" (but without more theoretical expressiveness)
- syntactic sugar

EBNF

Main additional notational freedom: use regular expressions on the rhs of productions. They can contain terminals and non-terminals.

- EBNF: officially standardized, but often: all "sugared" BNFs are called EBNF
- in the standard:
  - α^* written as $\{\alpha\}$
  - $\alpha?$ written as $[\alpha]$
- supported (in the standardized form or other) by some parser tools, but not in all
- remember equation (3.2)

EBNF examples

$$\begin{array}{rcl} A & \rightarrow & \beta\{\alpha\} \qquad\qquad \text{for} \quad A \rightarrow A\alpha \mid \beta \\ A & \rightarrow & \{\alpha\}\beta \qquad\qquad \text{for} \quad A \rightarrow \alpha A \mid \beta \end{array}$$

$$\begin{array}{rcl} \textit{stmt-seq} & \rightarrow & \textit{stmt}\ \{\mathbf{;}\,\textit{stmt}\} \\ \textit{stmt-seq} & \rightarrow & \{\textit{stmt}\,\mathbf{;}\}\ \textit{stmt} \\ \textit{if-stmt} & \rightarrow & \textbf{if}\ \mathbf{(}\ \textit{exp}\ \mathbf{)}\ \textit{stmt}[\textbf{else}\ \textit{stmt}] \end{array}$$

greek letters: for non-terminals or terminals.

Some yacc style grammar

```
/* Infix notation calculator--calc */
%{
#define YYSTYPE double
#include <math.h>
%}

/* BISON Declarations */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation        */

/* Grammar follows */
%%
input:    /* empty string */
        | input line
;

line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
%%
```

3.4 Syntax of a “Tiny” language

BNF-grammar for TINY

$$
\begin{array}{rcl}
\textit{program} & \to & \textit{stmt-seq} \\
\textit{stmt-seq} & \to & \textit{stmt-seq}\,;\textit{stmt} \mid \textit{stmt} \\
\textit{stmt} & \to & \textit{if-stmt} \mid \textit{repeat-stmt} \mid \textit{assign-stmt} \\
 & \mid & \textit{read-stmt} \mid \textit{write-stmt} \\
\textit{if-stmt} & \to & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{end} \\
 & \mid & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt}\ \textbf{end} \\
\textit{repeat-stmt} & \to & \textbf{repeat}\ \textit{stmt-seq}\ \textbf{until}\ \textit{expr} \\
\textit{assign-stmt} & \to & \textbf{identifier}\ \textbf{:=}\ \textit{expr} \\
\textit{read-stmt} & \to & \textbf{read}\ \textbf{identifier} \\
\textit{write-stmt} & \to & \textbf{write}\ \ \textit{expr} \\
\textit{expr} & \to & \textit{simple-expr}\ \textit{comparison-op}\ \textit{simple-expr} \mid \textit{simple-expr} \\
\textit{comparison-op} & \to & \texttt{<} \mid \texttt{=} \\
\textit{simple-expr} & \to & \textit{simple-expr}\ \textit{addop}\ \textit{term} \mid \textit{term} \\
\textit{addop} & \to & \texttt{+} \mid \texttt{-} \\
\textit{term} & \to & \textit{term}\ \textit{mulop}\ \textit{factor} \mid \textit{factor} \\
\textit{mulop} & \to & \texttt{*} \mid \texttt{/} \\
\textit{factor} & \to & \textbf{(}\ \textit{expr}\ \textbf{)} \mid \textbf{number} \mid \textbf{identifier}
\end{array}
$$

Syntax tree nodes

```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* ExpType is used for type checking */
typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];
     struct treeNode * sibling;
     int lineno;
     NodeKind nodekind;
     union { StmtKind stmt; ExpKind exp;} kind;
     union { TokenType op;
             int val;
             char * name; } attr;
     ExpType type; /* for type checking of exps */
```

Comments on C-representation

- typical use of `enum` type for that (in C)
- `enum`’s in C can be very efficient
- `treeNode` struct (records) is a bit “unstructured”

- newer languages/higher-level than C: better structuring advisable, especially for languages larger than Tiny.
- in Java-kind of languages: inheritance/subtyping and abstract classes/interfaces often used for better structuring

Sample Tiny program

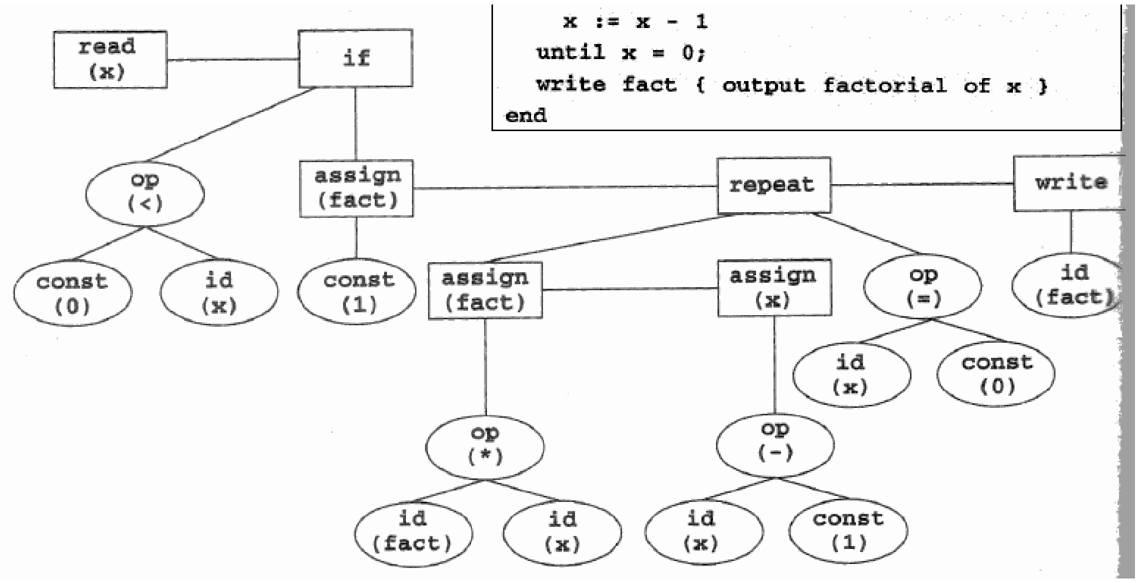
```
read x; { input as integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0;
  write fact    { output factorial of x }
end
```

Same Tiny program again

```
read x; { input as integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0;
  write fact    { output factorial of x }
end
```

- *keywords / reserved words* highlighted by bold-face type setting
- reserved syntax like `0`, `:=`, ... is not bold-faced
- comments are italicized

Abstract syntax tree for a tiny program



```
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

Some questions about the Tiny grammy

- is the grammar unambiguous?
- How can we change it so that the Tiny allows empty statements?
- What if we want semicolons *in between* statements and not *after*?
- What is the precedence and associativity of the different operators?

3.5 Chomsky hierarchy

The Chomsky hierarchy

- linguist Noam Chomsky [5]
- **important** classification of (formal) languages (sometimes Chomsky-Sch\"utzenberger)
- 4 levels: type 0 languages – type 3 languages
- levels related to machine models that generate/recognize them
- so far: regular languages and CF languages

Overview

	rule format	languages	machines	closed
3	$A \to aB$, $A \to a$	regular	NFA, DFA	all
2	$A \to \alpha_1 \beta \alpha_2$	CF	pushdown automata	$\cup$, $*$, $\circ$
1	$\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$	context-sensitive	(linearly restricted automata)	all
0	$\alpha \to \beta$, $\alpha \neq \epsilon$	recursively enumerable	Turing machines	all, except complement

Conventions

- terminals $a, b, \ldots \in \Sigma_T$,
- non-terminals $A, B, \ldots \in \Sigma_N$
- general words $\alpha, \beta \ldots \in (\Sigma_T \cup \Sigma_N)^*$

Remark: Chomsky hierarchy

The rule format for type 3 languages (= regular languages) is also called *right-linear*. Alternatively, one can use *left-linear* rules. If one mixes right- and left-linear rules, one leaves the class of regular languages. The rule-format above allows only *one* terminal symbol. In principle, if one had sequences of terminal symbols in a right-linear (or else left-linear) rule, that would be ok too.

Phases of a compiler & hierarchy

“Simplified” design?

1 big grammar for the whole compiler? Or at least a CSG for the front-end, or a CFG combining parsing and scanning?

theoretically possible, but bad idea:

- efficiency
- bad design
- especially combining scanner + parser in one BNF:
  - grammar would be needlessly large
  - separation of concerns: much clearer/ more efficient design
- for scanner/parsers: regular expressions + (E)BNF: simply **the formalisms of choice!**
  - front-end needs to do more than checking syntax, CFGs not expressive enough
  - for level-2 and higher: situation gets less clear-cut, plain CSG not too useful for compilers

Chapter
Parsing

What is it about?

Learning Targets of this Chapter

1. (context-free) grammars + BNF
2. ambiguity and other properties
3. terminology: tokens, lexemes,
4. different trees connected to grammars/parsing
5. derivations, sentential forms

The chapter corresponds to [6, Section 3.1–3.2] (or [9, Chapter 3]).

Contents

4.1 Introduction to parsing

What's a parser generally doing

task of parser = syntax analysis

- input: stream of **tokens** from lexer
- output:
  - **abstract syntax tree**
  - or meaningful diagnosis of source of *syntax error*

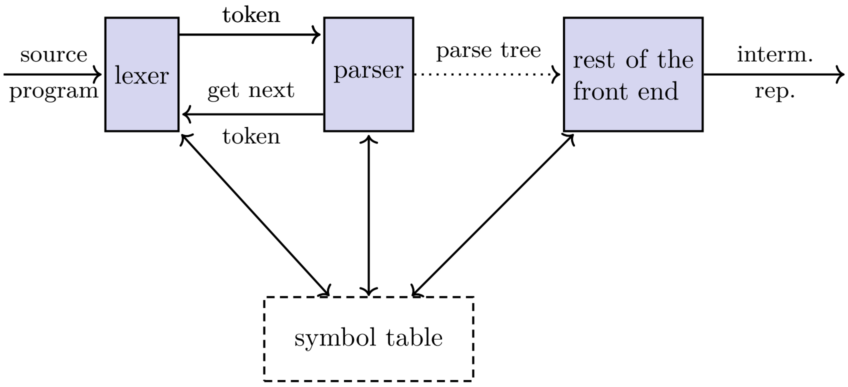
- the full "power" (i.e., expressiveness) of CFGs not used
- thus:
  - consider *restrictions* of CFGs, i.e., a specific subclass, and/or
  - *represented* in specific ways (no left-recursion, left-factored . . .)

Syntax errors (and other errors)

Since almost by definition, the *syntax* of a language are those aspects covered by a context-free grammar, a *syntax error* thereby is a violation of the grammar, something the parser has to detect. Given a CFG, typically given in BNF resp. implemented by a tool supporting a BNF variant, the parser (in combination with the lexer) must generate an AST *exactly* for those programs that adhere to the grammar and must *reject* all others. One says, the parser *recognizes* the given grammar. An important practical part when rejecting a program is to generate a meaningful *error message*, giving hints about potential locations of the error and potential reasons. In the most minimal way, the parser should inform the programmer where the parser tripped, i.e., telling how far, from left to right, it was able to proceed and informing where it stumbled: “parser error in line xxx/at character position yyy”). One typically has higher expectations for a real parser than just the line number, but that’s the basics.

It may be noted that also the subsequent phase, the *semantic analysis*, which takes the abstract syntax tree as input, may report errors, which are then no longer syntax errors but more complex kind of errors. One typical kind of error in the semantic phase is a *type error*. Also there, the minimal requirement is to indicate the probable location(s) where the error occurs. To do so, in basically all compilers, the nodes in an abstract syntax tree will contain information concerning the position in the original file the resp.\ node corresponds to (like line-numbers, character positions). If the parser would not add that information into the AST, the semantic analysis would have no way to relate potential errors it finds to the original, concrete code in the input. Remember: the compiler goes in *phases*, and once the parsing phase is over, there’s no going back to scan the file *again*.

Lexer, parser, and the rest



Top-down vs. bottom-up

- all parsers (together with lexers): *left-to-right*
- remember: parsers operate with *trees*
  - parse tree (concrete syntax tree): representing grammatical derivation
  - abstract syntax tree: data structure
- 2 fundamental classes
- while parser eats through the token stream, it grows, i.e., builds up (at least conceptually) the parse tree:

Bottom-up

Parse tree is being grown from the leaves to the root.

Top-down

Parse tree is being grown from the root to the leaves.

AST

- while parse tree mostly conceptual: parsing builds up the concrete data structure of AST bottom-up vs. top-down.

Parsing restricted classes of CFGs

- parser: better be "efficient"
- full complexity of CFLs: not really needed in practice
- classification of CF languages vs. CF grammars, e.g.:
  - left-recursion-freedom: condition on a grammar
  - ambiguous language vs. ambiguous grammar
- classification of grammars ⇒ classification of *languages*
  - a CF language is (inherently) ambiguous, if there's no unambiguous grammar for it
  - a CF language is top-down parseable, if there exists a grammar that allows top-down parsing . . .

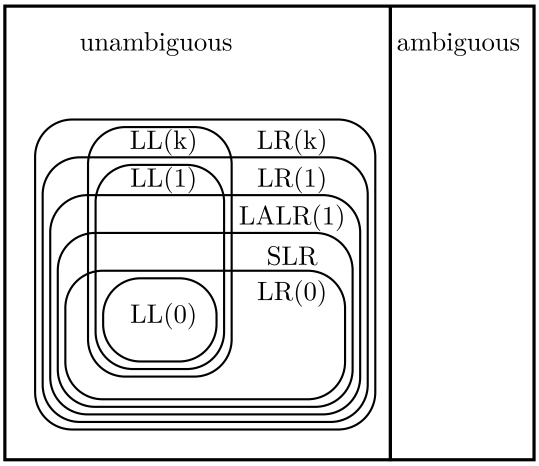
- in practice: classification of parser generating tools:
  - based on accepted notation for grammars: (BNF or some form of EBNF etc.)

Concerning the need (or the lack of need) for very expressive grammars, one should consider the following: if a parser has trouble to figure out if a program has a syntax error or not (perhaps using back-tracking), probably humans will have similar problems. So better keep it simple. And time in a compiler may be better spent elsewhere (optimization, semantical analysis).

Classes of CFG grammars/languages

- *maaaany* have been proposed & studied, including their relationships
- lecture concentrates on
  - top-down parsing, in particular
    - **LL(1)**
    - **recursive descent**
  - bottom-up parsing
    - **LR(1)**
    - **SLR**
    - **LALR(1)** (the class covered by yacc-style tools)
- grammars typically written in *pure* BNF

Relationship of some grammar (not language) classes



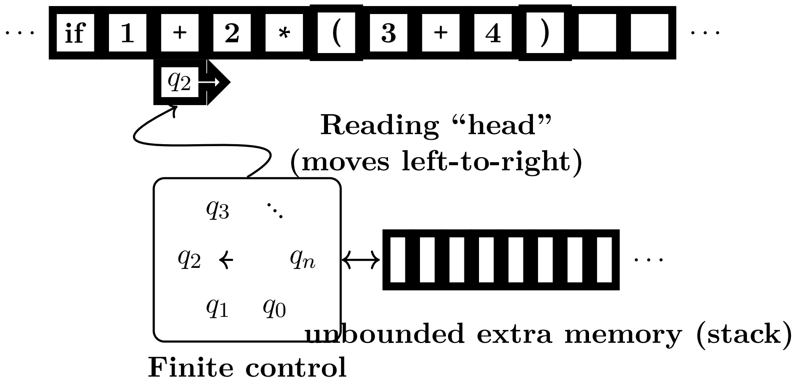
taken from [4]

4.2 Top-down parsing

General task (once more)

- Given: a CFG (but appropriately restricted)
- Goal: “systematic method” s.t.
  1. for every given word w: check syntactic correctness
  2. [build AST/representation of the parse tree as side effect]
  3. [do reasonable error handling]

Schematic view on “parser machine”



Note: sequence of *tokens* (not characters)

Derivation of an expression

Derivation

The slides contain some big series of overlays, showing the derivation. This derivation process is not reproduced here (resp. only a few slides later as some big array of steps).

factors and terms

$$\begin{array}{rcl} \mathit{exp} & \rightarrow & \mathit{term}\ \mathit{exp}' \\ \mathit{exp}' & \rightarrow & \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \boldsymbol{\epsilon} \\ \mathit{addop} & \rightarrow & \mathtt{+} \mid \mathtt{-} \\ \mathit{term} & \rightarrow & \mathit{factor}\ \mathit{term}' \\ \mathit{term}' & \rightarrow & \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \boldsymbol{\epsilon} \\ \mathit{mulop} & \rightarrow & \mathtt{*} \\ \mathit{factor} & \rightarrow & \mathtt{(}\ \mathit{exp}\ \mathtt{)} \mid \mathbf{n} \end{array} \tag{4.1}$$

Remarks concerning the derivation

Note:

- input = stream of tokens
- there: **1**... stands for token class **number** (for readability/concreteness), in the grammar: just **number**
- in full detail: pair of token class and token value $\langle \mathbf{number}, 1 \rangle$

Notation:

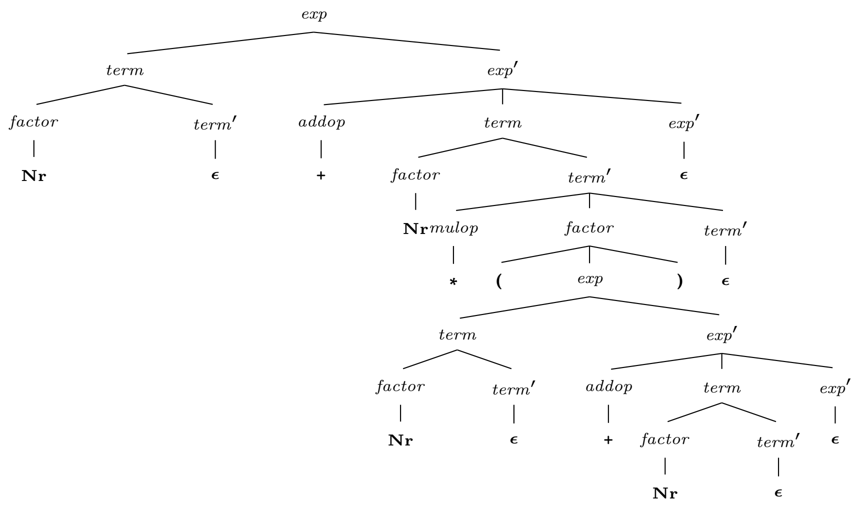
- underline: the *place* (occurrence of *non-terminal* where production is used)
- ~~*crossed out*~~:
  - *terminal* = *token* is considered treated
  - parser "moves on"
  - later implemented as `match` or `eat` procedure

Not as a “film” but at a glance: reduction sequence

$$
\begin{array}{ll}
\underline{\mathit{exp}} & \Rightarrow \\
\underline{\mathit{term}}\ \mathit{exp}' & \Rightarrow \\
\underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\cancel{\mathbf{number}}\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\,\underline{\mathit{term}'}\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\,\cancel{\epsilon}\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\,\underline{\mathit{exp}'} & \Rightarrow \\
\mathbf{number}\,\underline{\mathit{addop}}\ \mathit{term}\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\cancel{+}\ \mathit{term}\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\underline{\mathit{term}}\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\cancel{\mathbf{number}}\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\,\underline{\mathit{term}'}\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\,\underline{\mathit{mulop}}\ \mathit{factor}\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\cancel{*}\ \underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ \underline{(\ \mathit{exp}\ )}\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\cancel{(}\ \mathit{exp}\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \underline{\mathit{exp}}\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \underline{\mathit{term}}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \cancel{\mathbf{number}}\ \mathit{term}'\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}\,\underline{\mathit{term}'}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}\cancel{\epsilon}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}\,\underline{\mathit{exp}'}\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}\,\underline{\mathit{addop}}\ \mathit{term}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}\cancel{+}\ \mathit{term}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\underline{\mathit{term}}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\underline{\mathit{factor}}\ \mathit{term}'\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\cancel{\mathbf{number}}\ \mathit{term}'\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\mathbf{number}\,\underline{\mathit{term}'}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\mathbf{number}\cancel{\epsilon}\ \mathit{exp}'\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\mathbf{number}\,\underline{\mathit{exp}'}\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\mathbf{number}\cancel{\epsilon}\ )\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\mathbf{number}\cancel{)}\ \mathit{term}'\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\mathbf{number}\ )\ \underline{\mathit{term}'}\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\mathbf{number}\ )\ \cancel{\epsilon}\ \mathit{exp}' & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\mathbf{number}\ )\ \underline{\mathit{exp}'} & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\mathbf{number}\ )\ \cancel{\epsilon} & \Rightarrow \\
\mathbf{number}\ +\mathbf{number}\ *\ (\ \mathbf{number}+\mathbf{number}\ ) &
\end{array}
$$

Besides this derivation sequence, the slide version contains also an “overlay” version, expanding the sequence step by step. The derivation is a *left-most* derivation.

Best viewed as a tree



The tree does no longer contain information, which parts have been expanded first. In particular, the information that we have concretely done a left-most derivation when building up the tree in a top-down fashion is not part of the tree (as it is not important). The tree is an example of a *parse tree* as it contains information about the derivation process using rules of the grammar.

Non-determinism?

- not a "free" expansion/reduction/generation of some word, but
  - reduction of start symbol towards the *target word of terminals*

$$exp \ \Rightarrow^* \ \mathbf{1 + 2 * (3 + 4)}$$

  - i.e.: input stream of tokens "guides" the derivation process (at least it fixes the target)
- but: how much "guidance" does the target word (in general) gives?

Oracular derivation

$$\begin{array}{rcl} exp & \rightarrow & exp \texttt{ + } term \mid exp \texttt{ - } term \mid term \\ term & \rightarrow & term \texttt{ * } factor \mid factor \\ factor & \rightarrow & \textbf{(}\, exp \,\textbf{)} \mid \textbf{number} \end{array}$$

$$
\begin{array}{lll}
\underline{\mathit{exp}} & \Rightarrow_1 & \downarrow 1 + 2 * 3 \\
\underline{\mathit{exp}} + \mathit{term} & \Rightarrow_3 & \downarrow 1 + 2 * 3 \\
\underline{\mathit{term}} + \mathit{term} & \Rightarrow_5 & \downarrow 1 + 2 * 3 \\
\underline{\mathit{factor}} + \mathit{term} & \Rightarrow_7 & \downarrow 1 + 2 * 3 \\
\mathbf{number} + \mathit{term} & & \downarrow 1 + 2 * 3 \\
\mathbf{number} + \mathit{term} & & 1 \downarrow + 2 * 3 \\
\mathbf{number} + \underline{\mathit{term}} & \Rightarrow_4 & 1 + \downarrow 2 * 3 \\
\mathbf{number} + \underline{\mathit{term}} * \mathit{factor} & \Rightarrow_5 & 1 + \downarrow 2 * 3 \\
\mathbf{number} + \underline{\mathit{factor}} * \mathit{factor} & \Rightarrow_7 & 1 + \downarrow 2 * 3 \\
\mathbf{number} + \mathbf{number} * \mathit{factor} & & 1 + \downarrow 2 * 3 \\
\mathbf{number} + \mathbf{number} * \mathit{factor} & & 1 + 2 \downarrow * 3 \\
\mathbf{number} + \mathbf{number} * \underline{\mathit{factor}} & \Rightarrow_7 & 1 + 2 * \downarrow 3 \\
\mathbf{number} + \mathbf{number} * \mathbf{number} & & 1 + 2 * \downarrow 3 \\
\mathbf{number} + \mathbf{number} * \mathbf{number} & & 1 + 2 * 3 \downarrow
\end{array}
$$

The derivation shows a left-most derivation. Again, the “redex” is underlined. In addition, we show on the right-hand column the input and the progress which is being done on that input. The subscripts on the derivation arrows indicate which rule is chosen in that particular derivation step.

The point of the example is the following: Consider lines 7 and 8, and the steps the parser does. In line 7, it is about to expand *term* which is the left-most terminal. Looking into the “future” the unparsed part is `2 * 3`. In that situation, the parser chooses production 4 (indicated by $\Rightarrow_4$). In the next line, the left-most non-terminal is *term again* and also the non-processed input has not changed. However, in that situation, the “oracular” parser chooses $\Rightarrow_5$.

What does that mean? It means, that the look-ahead did not help the parser! It used all look-ahead there is, namely until the very end of the word. And it *still* cannot make the right decision with all the knowledge available at that given point. Note also: choosing wrongly (like $\Rightarrow_5$ instead of $\Rightarrow_4$ or the other way around) would lead to a failed parse (which would require *backtracking*). That means, it’s unparseable without backtracking (and not amount of look-ahead will help), at least we need backtracking, if we do left-derivations and top-down.

Right-derivations are not really an option, as typically we want to eat the input left-to-right. Secondly, right-most derivations will suffer from the same problem (perhaps not for the very grammar but in general, so nothing would even be gained.)

On the other hand: bottom-up parsing later works on different principles, so the particular problem illustrate by this example will not bother that style of parsing (but there are other challenges then).

So, what *is* the problem then here? The reason why the parser could not make a uniform decision (for example comparing line 7 and 8) comes from the fact that these two particular lines are connected by $\Rightarrow_4$, which corresponds to the production

$$\mathit{term} \rightarrow \mathit{term} * \mathit{factor}$$

there the derivation step replaces the left-most *term* by *term* again without moving ahead with the input. This form of rule is said to be *left-recursive* (with recursion on *term*). This is something that recursive descent parsers *cannot deal with* (or at least not without doing backtracking, which is *not* an option).

Note also: the grammar is not *ambigious* (without proof). If a grammar is ambiguous, also then parsing won't work properly (in this case neither will bottom-up parsing), but ambiguity is not the problem right here.

We will learn how to transform grammars automatically to *remove* left-recursion. It's an easy construction. Note, however, that the construction not necessarily results in a grammar that afterwards *is* top-down parsable. It simple removes a "feature" of the grammar which definitely cannot be treated by top-down parsing.

As side remark, for being super-precise: If a grammar contains left-recursion on a non-terminal which is "irrelevant" (i.e., no word will ever lead to a parse invovling that particular non-terminal), in that case, obviously, the left-recursion does not hurt. Of course, the grammar in that case would be "silly". We in general do not consider grammars which contain such irrelevant symbols (or have other such obviously meaningless defects). But unless we exclude such silly grammars, it's not 100% true that grammars with left-recursion cannot be treated via top-down parsing. But apart from that, it's the case:

> left-recursion destroys top-down parseability

(when based on left-most derivations/left-to-right parsing as it is always done for top-down).

Two principle sources of non-determinism here

Using production $A \to \beta$

$$S \Rightarrow^* \alpha_1 \; A \; \alpha_2 \Rightarrow \alpha_1 \; \beta \; \alpha_2 \Rightarrow^* w$$

Conventions

- $\alpha_1, \alpha_2, \beta$: word of terminals and nonterminals
- w: word of terminals, only
- A: one non-terminal

2 choices to make

1. **where**, i.e., on **which occurrence of a non-terminal** in $\alpha_1 A \alpha_2$ to apply a production[1]
2. **which production** to apply (for the chosen non-terminal).

[1] Note that α_1 and α_2 may contain non-terminals, including further occurrences of A.

Left-most derivation

- that's the *easy* part of non-determinism
- taking care of "where-to-reduce" non-determinism: *left-most* derivation
- notation $\Rightarrow_l$
- some of the example derivations earlier used that

Non-determinism vs. ambiguity

- Note: the "where-to-reduce"-non-determinism $\neq$ ambiguitiy of a grammar
- in a way ("theoretically"): where to reduce next is *irrelevant*:
  - the order in the sequence of derivations *does not matter*
  - what does matter: the **derivation tree** (aka the **parse tree**)

Lemma 4.2.1 (Left or right, who cares)**.** $S \Rightarrow_l^* w$ *iff* $S \Rightarrow_r^* w$ *iff* $S \Rightarrow^* w$.

- however ("practically"): a (deterministic) parser implementation: must make a *choice*

Using production $A \to \beta$

$$S \Rightarrow^* \alpha_1\ A\ \alpha_2 \Rightarrow \alpha_1\ \beta\ \alpha_2 \Rightarrow^* w$$

$$S \Rightarrow_l^* w_1\ A\ \alpha_2 \Rightarrow w_1\ \beta\ \alpha_2 \Rightarrow_l^* w$$

Remember the notational conventions used here: w stand for words containing *terminals* only, whereas α represents arbitrary words.

What about the "which-right-hand side" non-determinism?

$$A \to \beta \mid \gamma$$

Is that the correct choice?

$$S \Rightarrow_l^* w_1\ A\ \alpha_2 \Rightarrow w_1\ \beta\ \alpha_2 \Rightarrow_l^* w$$

- reduction with "guidance": don't loose sight of the target w
  - "past" is fixed: $w = w_1 w_2$
  - "future" is not:

$$A\alpha_2 \Rightarrow_l \beta\alpha_2 \Rightarrow_l^* w_2 \quad \text{or else} \quad A\alpha_2 \Rightarrow_l \gamma\alpha_2 \Rightarrow_l^* w_2\ ?$$

Needed (minimal requirement):

In such a situation, "future target" w_2 must *determine* which of the rules to take!

Deterministic, yes, but still impractical

$$A\alpha_2 \Rightarrow_l \beta\alpha_2 \Rightarrow_l^* w_2 \quad \text{or else} \quad A\alpha_2 \Rightarrow_l \gamma\alpha_2 \Rightarrow_l^* w_2 \;?$$

- the "target" w_2 is of *unbounded length*!

⇒ impractical, therefore:

Look-ahead of length k

resolve the "which-right-hand-side" non-determinism inspecting only fixed-length prefix of w_2 (for *all* situations as above)

LL(k) grammars

CF-grammars which *can* be parsed doing that.[2]

4.3 First and follow sets

We had a general look of what a look-ahead is, and how it helps in top-down parsing. We also saw that left-recursion is bad for top-down parsing (in particular, there can't be any look-ahead to help the parser). The definition discussed so far, being based on arbitrary derivations, were impractical. What is needed is a criterion not for derivations, but on *grammars* that can be used to check, whether the grammar is parseable in a top-down manner with a look-ahead of, say k. Actually we will concentrate on a look-ahead of $k = 1$, which is practically a decent thing to do.

The considerations leading to a useful criterion for top-down parsing with backtracking will involve the definition of the so-called "first-sets". In connection with that definition, there will be also the (related) definition of *follow-sets*.

The definitions, as mentioned, will help to figure out if a grammar is top-down parseable. Such a grammar will then be called an LL(1) grammar. One could straightforwardly generalize the definition to LL(k) (which would include generalizations of the first and follow sets), but that's not part of the pensum. Note also: the first and follow set definition will *also* be used when discussing *bottom-up* parsing later.

Besides that, in this section we will also discuss *what to do* if the grammar is not LL(1). That will lead to a transformation removing left-recursion. That is not the only defect that one wants to transform away. A second problem that is a show-stopper for LL(1)-parsing is known as "common left factors". If a grammar suffers from that, there is another transformation called *left factorization* which can remedy that.

[2] Of course, one can always write a parser that "just makes some decision" based on looking ahead k symbols. The question is: will that allow to capture *all* words from the grammar and *only* those.

First and Follow sets

- general concept for grammars
- certain types of analyses (e.g. parsing):
  - info needed about possible "forms" of *derivable* words,

First-set of A

which terminal symbols can appear at the start of strings *derived from* a given nonterminal A

Follow-set of A

Which terminals can follow A in some *sentential form*.

Remarks

- sentential form: word *derived from* grammar's starting symbol
- later: different algos for first and follow sets, for all non-terminals of a given grammar
- mostly straightforward
- one complication: *nullable* symbols (non-terminals)
- Note: those sets depend on grammar, not the language

First sets

Definition 4.3.1 (First set)**.** Given a grammar G and a non-terminal A. The *first-set* of A, written $First_G(A)$ is defined as

$$First_G(A) = \{a \mid A \Rightarrow_G^* a\alpha, \quad a \in \Sigma_T\} + \{\epsilon \mid A \Rightarrow_G^* \epsilon\} \ . \tag{4.2}$$

Definition 4.3.2 (Nullable)**.** Given a grammar G. A non-terminal $A \in \Sigma_N$ is *nullable*, if $A \Rightarrow^* \epsilon$.

Nullable

The definition here of being nullable refers to a non-terminal symbol. When concentrating on context-free grammars, as we do for parsing, that's basically the only interesting case. In principle, one can define the notion of being nullable analogously for arbitrary words from the whole alphabet $\Sigma = \Sigma_T + \Sigma_N$. The form of productions in CFGs makes it obvious, that the only words which actually may be nullable are words containing only non-terminals. Once a terminal is derived, it can never be "erased". It's equally easy to see, that a word $\alpha \in \Sigma_N^*$ is nullable iff all its non-terminal symbols are nullable. The same remarks apply to context-sensitive (but not general) grammars.

For level-0 grammars in the Chomsky-hierarchy, also words containing terminal symbols may be nullable, and nullability of a word, like most other properties in that stetting, becomes undecidable.

First and follow sets

One point worth noting is that the first and the follow sets, while seemingly quite similar, *differ* in one important aspect (the follow set definition will come later). The first set is about words derivable from a given non-terminal A. The follow set is about words derivable from the starting symbol! As a consequence, non-terminals A which are not *reachable* from the grammar's starting symbol have, by definition, an empty follow set. In contrast, non-terminals unreachable from a/the start symbol may well have a non-empty first-set. In practice a grammar containing unreachable non-terminals is ill-designed, so that distinguishing feature in the definition of the first and the follow set for a non-terminal may not matter so much. Nonetheless, when *implementing* the algo's for those sets, those subtle points do matter! In general, to avoid all those fine points, one works with grammars satisfying a number of common-sense restructions. One are so called *reduced grammars*, where, informally, all symbols "play a role" (all are reachable, all can derive into a word of terminals).

Examples

- Cf. the Tiny grammar
- in Tiny, as in most languages

$$First(\textit{if-stmt}) = \{\text{"}\textbf{if}\text{"}\}$$

- in many languages:

$$First(\textit{assign-stmt}) = \{\textbf{identifier}, \text{"("}\}$$

- typical *Follow* (see later) for statements:

$$Follow(stmt) = \{\text{";"}, \text{"}\textbf{end}\text{"}, \text{"}\textbf{else}\text{"}, \text{"}\textbf{until}\text{"}\}$$

Remarks

- note: special treatment of the empty word $\boldsymbol{\epsilon}$
- in the following: if grammar G clear from the context
  - $\Rightarrow^*$ for $\Rightarrow^*_G$
  - *First* for $First_G$
  - ...
- definition so far: "top-level" for start-symbol, only
- next: a more general definition
  - definition of First set of arbitrary symbols (and even words)

  - and also: definition of First for a symbol *in terms of* First for "other symbols" (connected by *productions*)

⇒ **recursive** definition

A more algorithmic/recursive definition

- grammar *symbol* X: terminal or non-terminal or $\boldsymbol{\epsilon}$

Definition 4.3.3 (First set of a symbol)**.** Given a grammar G and grammar symbol X. The *first-set* of X, written $\mathit{First}(X)$, is defined as follows:

1. If $X \in \Sigma_T + \{\boldsymbol{\epsilon}\}$, then $\mathit{First}(X)$ contains X.
2. If $X \in \Sigma_N$: For each production

$$X \to X_1 X_2 \ldots X_n$$

   a) $\mathit{First}(X)$ contains $\mathit{First}(X_1) \setminus \{\boldsymbol{\epsilon}\}$
   b) If, for some $i < n$, *all* $\mathit{First}(X_1), \ldots, \mathit{First}(X_i)$ contain $\boldsymbol{\epsilon}$, then $\mathit{First}(X)$ contains $\mathit{First}(X_{i+1}) \setminus \{\boldsymbol{\epsilon}\}$.
   c) If all $\mathit{First}(X_1), \ldots, \mathit{First}(X_n)$ contain $\boldsymbol{\epsilon}$, then $\mathit{First}(X)$ contains $\{\boldsymbol{\epsilon}\}$.

Recursive definition of *First*?

The following discussion may be ignored, if wished. Even if details and theory behind it is beyond the scope of this lecture, it is worth considering above definition more closely. One may even consider if it is a definition at all (resp. in which way it is a definition).

One naive first impression may be: it's a kind of a "functional definition", i.e., the above Definition 4.3.3 gives a recursive definition of the function *First*. As discussed later, everything gets rather simpler if we would not have to deal with nullable words and ϵ-productions. For the point being explained here, let's assume that there are no such productions and get rid of the special cases, cluttering up Definition 4.3.3. Removing the clutter gives the following simplified definition:

Definition 4.3.4 (First set of a symbol (no $\boldsymbol{\epsilon}$-productions))**.** Given a grammar G and grammar symbol X. The *First-set* of $X \neq \boldsymbol{\epsilon}$, written $\mathit{First}(X)$ is defined as follows:

1. If $X \in \Sigma_T$, then $\mathit{First}(X) \supseteq \{X\}$.
2. If $X \in \Sigma_N$: For each production

$$X \to X_1 X_2 \ldots X_n \ ,$$

$$\mathit{First}(X) \ \supseteq \ \mathit{First}(X_1).$$

Compared to the previous condition, I did the following minor adaptation (apart from cleaning up the $\boldsymbol{\epsilon}$'s): I replaced the English word "contains" with the superset relation symbol $\supseteq$.

Now, with Definition 4.3.4 as a simplified version of the original definition being made slightly more explicit: in which way is that a definition at all?

For being a definition for *First*(X), it seems awfully lax. Already in (1), it "defines" that *First*(X) should "at least contain X". A similar remark applies to case (2) for non-terminals. Those two requirements are as such well-defined, but *they don't define First*(X) *in a unique manner!* Definition 4.3.4 defines what the set *First*(X) should *at least* contain!

So, in a nutshell, one should not consider Definition 4.3.4 a "recursive definition of *First*(X)" but rather

> "a definition of recursive conditions on *First*(X), which, when satisfied, ensures that *First*(X) contains *at least* all non-terminals we are after".

What we are *really* after is the *smallest First*(X) which satisfies those conditions of the definitions.

Now one may think: the problem is that definition is just "sloppy". Why does it use the word "contain" resp.\ the $\supseteq$-relation, instead of requiring equality, i.e., $=$? While plausible at first sight, unfortunately, whether we use $\supseteq$ or set equality $=$ in Definition 4.3.4 does not change anything.

Anyhow, the core of the matter is not $=$ vs. $\supseteq$. The core of the matter is that "Definition" 4.3.4 is *circular!*

Considering that definition of *First*(X) as a plain functional and recursive definition of a procedure missed the fact that grammar can, of course, contain "loops". Actually, it's almost a characterizing feature of reasonable context-free grammars (or even regular grammars) that they contain "loops" – that's the way they can describe infinite languages.

In that case, obviously, considering Definition 4.3.3 with $=$ instead of $\supseteq$ as the recursive definition of a function leads immediately to an "infinite regress", the recurive function won't terminate. So again, that's not helpful.

Technically, such a definition can be called a recursive *constraint* (or a constraint system, if one considers the whole definition to consist of more than one constraint, namely for different terminals and for different productions).

For words

Definition 4.3.5 (First set of a word)**.** Given a grammar G and word α. The *first-set* of

$$\alpha = X_1 \ldots X_n \ ,$$

written *First*(α) is defined inductively as follows:

1. *First*(α) contains *First*$(X_1) \setminus \{\epsilon\}$
2. for each $i = 2, \ldots n$, if *First*(X_k) contains ϵ for *all* $k = 1, \ldots, i-1$, then *First*(α) contains *First*$(X_i) \setminus \{\epsilon\}$
3. If all *First*$(X_1), \ldots,$ *First*(X_n) contain ϵ, then *First*(X) contains $\{\epsilon\}$.

Concerning the definition of First

The definition here is of course very close to the definition of inductive case of the previous definition, i.e., the first set of a non-terminal. Whereas the previous definition was a recursive, this one is not.

Note that the word α may be empty, i.e., $n = 0$, In that case, the definition gives $First(\boldsymbol{\epsilon}) = \{\epsilon\}$ (due to the 3rd condition in the above definition). In the definitions, the empty word $\boldsymbol{\epsilon}$ plays a specific, mostly technical role. The original, non-algorithmic version of Definition 4.3.1, makes it already clear, that the first set *not precisely* corresponds to the set of terminal symbols that can appear at the beginning of a derivable word. The correct intuition is that it corresponds to that set of terminal symbols *together* with ϵ as a special case, namely when the initial symbol is nullable.

That may raise two questions. 1) Why does the definition makes that as special case, as opposed to just using the more "straightforward" definition without taking care of the nullable situation? 2) What role does ϵ play here?

The second question has no "real" answer, it's a choice which is being made which could be made differently. What the definition from equation (4.3.1) in fact says is: "give the set of terminal symbols in the derivable word **and** indicate whether or not the start symbol is *nullable*." The information might as well be interpreted as a *pair* consisting of a set of terminals *and* a boolean (indicating nullability). The fact that the definition of *First* as presented here uses $\boldsymbol{\epsilon}$ to indicate that additional information is a particular choice of representation (probably due to historical reasons: "they always did it like that . . . "). For instance, the influential "Dragon book" [1, Section 4.4.2] uses the $\boldsymbol{\epsilon}$-based definition. The texbooks [3] (and its variants) don't use $\boldsymbol{\epsilon}$ as indication for nullability.

In order that this definition works, it is important, obviously, that $\boldsymbol{\epsilon}$ is *not* a terminal symbol, i.e., $\boldsymbol{\epsilon} \notin \Sigma_T$ (which is generally assumed).

Having clarified 2), namely that using ϵ is a matter of conventional choice, remains question 1), why bother to include nullability-information in the definition of the first-set *at all*, why bother with the "extra information" of nullability? For that, there is a real technical reason: For the *recursive* definitions to work, we need the information whether or not a symbol or word is *nullable*, therefore it's given back as information.

As a further point concerning the first sets: The slides give 2 definitions, Definition 4.3.1 and Definition 4.3.3. Of course they are intended to mean the same. The second version is a more recursive or algorithmic version, i.e., closer to a recursive algorithm. If one takes the first one as the "real" definition of that set, in principle we would be obliged to prove that both versions actually describe the same same (resp. that the recurive definition *implements* the original definition). The same remark applies also to the non-recursive/iterative code that is shown next.

Pseudo code

```
for all $X \in A \cup \{\epsilon\}$ do
    First[X] := X
end;

for all non-terminals $A$ do
  First[A] := {}
end
while there are changes to any First[A] do
  for each production $A \to X_1 \dots X_n$ do
    k := 1;
    continue := true
    while continue = true and $k \leq n$ do
      First[A] := First[A] $\cup$ First[$X_k$] $\setminus \{\epsilon\}$
      if $\epsilon \notin$ First[$X_k$] then continue := false
      k := k + 1
    end;
    if  continue = true
    then First[A] := First[A] $\cup \{\epsilon\}$
  end;
end
```

If only we could do away with special cases for the empty words ...

for a grammar without ϵ-*productions*.[3]

```
for all non-terminals $A$ do
  First[A] := {}          // counts as change
end
while there are changes to any First[A] do
  for each production $A \to X_1 \dots X_n$ do
      First[A] := First[A] $\cup$ First[$X_1$]
  end;
end
```

This simplification is added for illustration, only. What makes the algorithm slightly more than just immediate is the fact that symbols can be *nullable* (non-terminals can be nullable). If we don't have ϵ-transitions, then no symbol is nullable. Under this simplifying assumption, the algorithm looks quite simpler. We don't need to check for nullability (i.e., we don't need to check if ϵ is part of the first sets), and moreover, we can do without the inner while loop, walking down the right-hand side of the production as long as the symbols turn out to be nullable (since we know they are not).

Example expression grammar (from before)

$$
\begin{array}{rcl}
exp & \to & exp\ addop\ term \mid term \\
addop & \to & \mathtt{+} \mid \mathtt{-} \\
term & \to & term\ mulop\ factor \mid factor \\
mulop & \to & \mathtt{*} \\
factor & \to & \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}
\end{array}
\tag{4.3}
$$

[3] A production of the form $A \to \epsilon$.

Example expression grammar (expanded)

$$\begin{array}{rcl}
exp & \to & exp\ \ addop\ \ term \\
exp & \to & term \\
addop & \to & \mathtt{+} \\
addop & \to & \mathtt{-} \\
term & \to & term\ \ mulop\ \ factor \\
term & \to & factor \\
mulop & \to & \mathtt{*} \\
factor & \to & \mathtt{(}\ exp\ \mathtt{)} \\
factor & \to & \mathbf{n}
\end{array} \tag{4.4}$$

"Run" of the algo

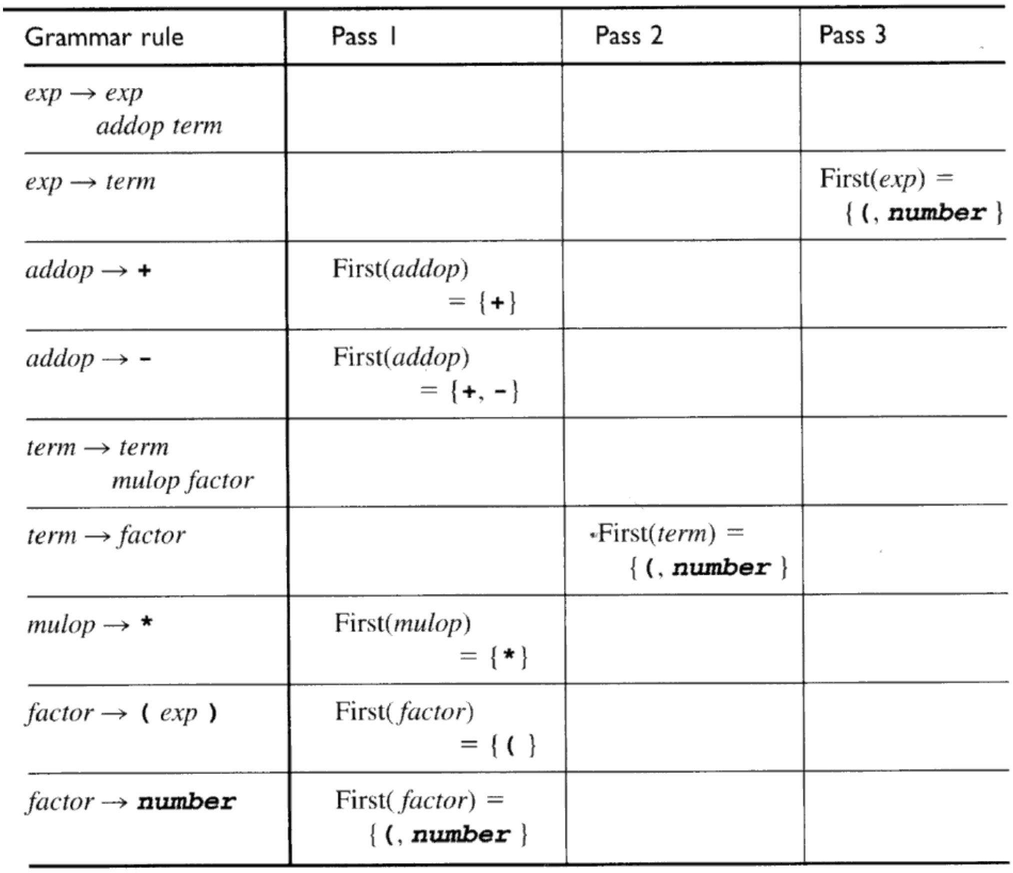
nr		pass 1	pass 2	pass 3
1	$exp \to exp\, addop\, term$			
2	$exp \to term$			
3	$addop \to \mathtt{+}$			
4	$addop \to \mathtt{-}$			
5	$term \to term\, mulop\, factor$			
6	$term \to factor$			
7	$mulop \to \mathtt{*}$			
8	$factor \to \mathtt{(}\, exp\, \mathtt{)}$			
9	$factor \to \mathbf{n}$			

How the algo works

The first thing to observe: the grammar does not contain ϵ-productions. That, very fortunately, simplifies matters considerably! It should also be noted that the table from above is a schematic illustration of a particular *execution strategy* of the pseudo-code. The pseudo-code itself leaves out details of the evaluation, notably *the order* in which non-deterministic choices are done by the code. The main body of the pseudo-code is given by two nested loops. Even if details (of data structures) are not given, one possible way of interpreting the code is as follows: the outer while-loop figures out which of the entries in the `First`-array have "recently" been changed, remembers that in a "collection" of non-terminals A's, and that collection is then worked off (i.e. iterated over) on the inner loop. Doing it like that leads to the "passes" shown in the table. In other words, the two dimensions of the table represent the fact that there are 2 nested loops.

Having said that: it's not the only way to "traverse the productions of the grammar". One could arrange a version with only one loop and a collection data structure, which contains all productions $A \to X_1 \ldots X_n$ such that `First[A]` has "recently been changed". That data structure therefore contains all the productions that "still need to be treated". Such a collection data structure containing "all the work still to be done" is known as *work-list*, even if it needs not technically be a list. It can be a queue, i.e., following a FIFO strategy, it can be a stack (realizing LIFO), or some other strategy or heuristic. Possible is also a randomized, i.e., non-deterministic strategy (which is sometimes known as chaotic iteration).

"Run" of the algo



Grammar rule	Pass I	Pass 2	Pass 3
exp → *exp* *addop term*			
exp → *term*			First(*exp*) = { **(**, **number** }
addop → **+**	First(*addop*) = { **+** }		
addop → **-**	First(*addop*) = { **+**, **-** }		
term → *term* *mulop factor*			
term → *factor*		First(*term*) = { **(**, **number** }	
mulop → *****	First(*mulop*) = { ***** }		
factor → **(** *exp* **)**	First(*factor*) = { **(** }		
factor → **number**	First(*factor*) = { **(**, **number** }		

Collapsing the rows & final result

- results per pass:

	1	2	3
exp			$\{(, \mathbf{n}\}$
addop	$\{+, -\}$		
term		$\{(, \mathbf{n}\}$	
mulop	$\{*\}$		
factor	$\{(, \mathbf{n}\}$		

- final results (at the end of pass 3):

	First[_]
exp	$\{(, \mathbf{n}\}$
addop	$\{+, -\}$
term	$\{(, \mathbf{n}\}$
mulop	$\{*\}$
factor	$\{(, \mathbf{n}\}$

Work-list formulation

```
for  all non-terminals $A$  do
   First [A]  :=  {}
   WL          :=  $P$    //  all  productions
end
while  WL $\neq \emptyset$  do
   remove  one  $(A \to X_1 \ldots X_n)$  from  WL
   if      First [A]  $\neq$  First [A]  $\cup$  First [$X_1$]
   then    First [A]  :=  First [A]  $\cup$  First [$X_1$]
        add  all  productions  $(A \to X'_1 \ldots X'_m)$  to  WL
   else   skip
end
```

- no ϵ-productions
- worklist here: "collection" of productions
- alternatively, with slight reformulation: "collection" of non-terminals instead also possible

Follow sets

Definition 4.3.6 (Follow set)**.** Given a grammar G with start symbol S, and a non-terminal A.

The *follow-set* of A, written $Follow_G(A)$, is

$$Follow_G(A) = \{a \mid S\,\$ \Rightarrow^*_G \alpha_1 A a \alpha_2, \quad a \in \Sigma_T + \{\,\$\,\}\} \,. \tag{4.5}$$

- $\$$ as special end-marker
- typically: start symbol *not* on the right-hand side of a production

Special symbol $

The symbol **$** can be interpreted as “end-of-file” (EOF) token. It’s standard to assume that the start symbol S does not occur on the right-hand side of any production. In that case, the follow set of S contains **$** as *only* element. Note that the follow set of other non-terminals may well contain **$**.

As said, it’s common to assume that S does not appear on the right-hand side of any production. For a start, S won’t occur “naturally” there anyhow in practical programming language grammars. Furthermore, with S occuring only on the left-hand side, the grammar has a slightly nicer shape insofar as it makes its algorithmic treatment slightly nicer. It’s basically the same reason why one sometimes assumes that, for instance, control-flow graphs have one “isolated” entry node (and/or an isolated exit node), where being isolated means, that no edge in the graph goes (back) into into the entry node; for exits nodes, the condition means, no edge goes out. In other words, while the graph can of course contain loops or cycles, the entry node is not part of any such loop. That is done likewise to (slightly) simplify the treatment of such graphs. Slightly more generally and also connected to control-flow graphs: similar conditions about the shape of loops (not just for the entry and exit nodes) have been worked out, which play a role in loop optimization and intermediate representations of a compiler, such as static single assignment forms.

Coming back to the condition here concerning **$**: even if a grammar would not immediatly adhere to that condition, it’s trivial to transform it into that form by adding another symbol and make that the new start symbol, replacing the old. We will do that sometimes in exercises and examples later

Follow sets, recursively

Definition 4.3.7 (Follow set of a non-terminal)**.** Given a grammar G and nonterminal A. The *Follow-set* of A, written $\mathit{Follow}(A)$ is defined as follows:

1. If A is the start symbol, then $\mathit{Follow}(A)$ contains **$**.
2. If there is a production $B \to \alpha A \beta$, then $\mathit{Follow}(A)$ contains $\mathit{First}(\beta) \setminus \{\epsilon\}$.
3. If there is a production $B \to \alpha A \beta$ such that $\epsilon \in \mathit{First}(\beta)$, then $\mathit{Follow}(A)$ contains $\mathit{Follow}(B)$.

- **$**: “end marker” special symbol, only to be contained in the follow set

More imperative representation in pseudo code

```
Follow[$S$] := {$}
for all non-terminals $A \neq S$ do
  Follow[$A$] := {}
end
while there are changes to any Follow-set do
  for each production $A \to X_1 \ldots X_n$ do
    for each $X_i$ which is a non-terminal do
```

```
      Follow[$X_i$] := Follow[$X_i$] ∪ (First($X_{i+1} \dots X_n$) ∖ {$\epsilon$})
      if $\epsilon \in$ First($X_{i+1} X_{i+2} \dots X_n$)
      then Follow[$X_i$] := Follow[$X_i$] ∪ Follow[$A$]
    end
  end
end
```

Note! $First() = \{\epsilon\}$

Expression grammar once more

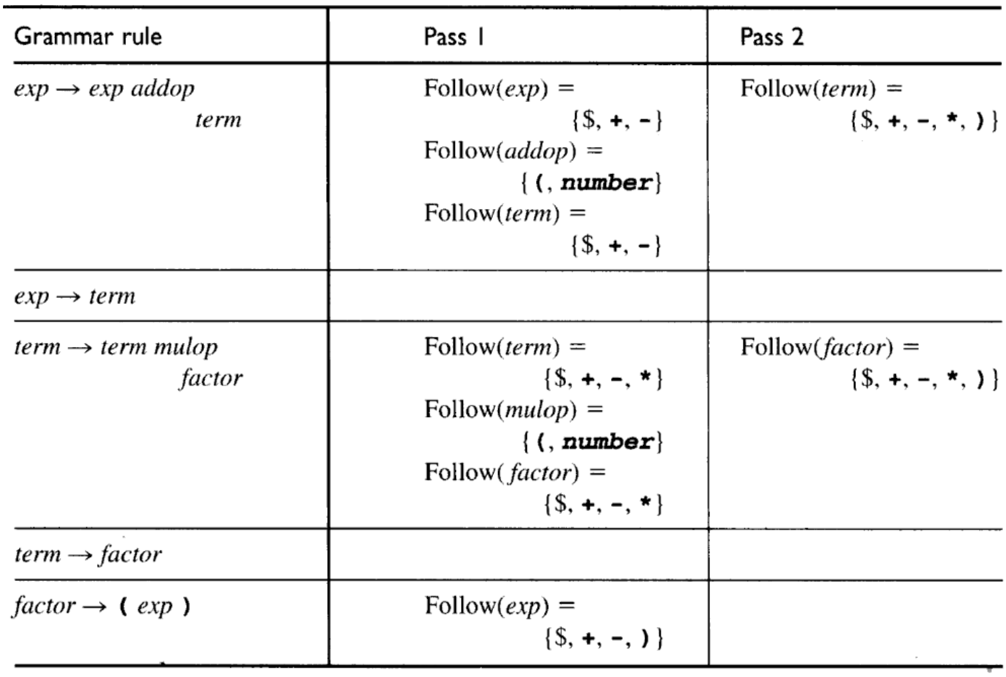
“Run” of the algo

nr		pass 1	pass 2
1	$exp \rightarrow exp\ addop\ term$		
2	$exp \rightarrow term$		
5	$term \rightarrow term\ mulop\ factor$		
6	$term \rightarrow factor$		
8	$factor \rightarrow$ **(** exp **)**		

normalsize

Recursion vs. iteration

“Run” of the algo



Grammar rule	Pass I	Pass 2

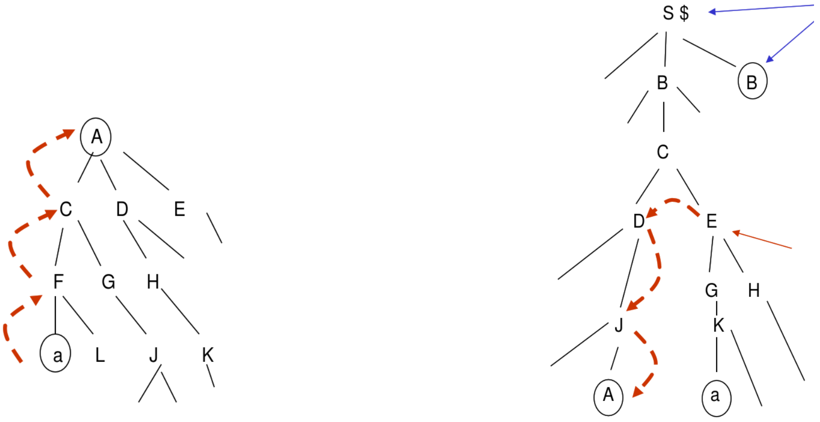
exp → *exp addop term*	Follow(*exp*) = {$, +, -}
	Follow(*addop*) = {(, **number**}
	Follow(*term*) = {$, +, -}
		Follow(*term*) = {$, +, -, *,)}

exp → *term*		

term → *term mulop factor*	Follow(*term*) = {$, +, -, *}
	Follow(*mulop*) = {(, **number**}
	Follow(*factor*) = {$, +, -, *}
		Follow(*factor*) = {$, +, -, *,)}

term → *factor*		
factor → (*exp*)	Follow(*exp*) = {$, +, -,)}	

Illustration of first/follow sets



- red arrows: illustration of information flow in the algos
- run of *Follow*:
  - relies on *First*
  - in particular $a \in First(E)$ (right tree)
- $\$ \in Follow(B)$

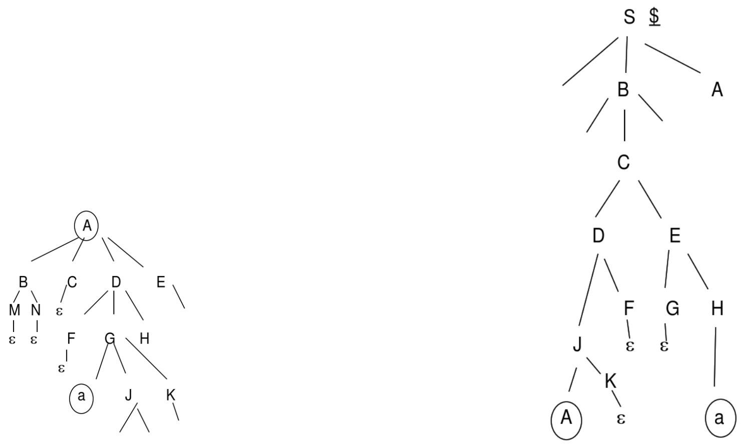
The two trees are just meant a illustrations (but still correct). The grammar itself is not given, but the tree shows relevant productions.

In case of the tree on the left (for the first sets): A is the root and must therefore be the start symbol. Since the root A has three children C, D, and E, there must be a production $A \to C\ D\ E$. etc.

The first-set definition would "immediately" detect that F has **a** in its first-set, i.e., all words derivable starting from F start with an a (and actually with no other terminal, as F is mentioned only once in that sketch of a tree). At any rate, only *after* determining that **a** is in the first-set of F, then it can enter the first-set of C, etc. and in this way percolating upwards the tree.

Note that the tree is insofar specific, in that all the internal nodes are *different* non-terminals. In more realistic settings, different nodes would represent the same non-terminal. And also in this case, one can think of the information percolating up.

More complex situation (nullability)



In the tree on the left, B, M, N, C, and F are *nullable*. That is marked in that the resulting first sets contain ϵ. There will also be exercises about that.

Some forms of grammars are less desirable than others

- **left-recursive** production:

$$A \to A\alpha$$

more precisely: example of *immediate* left-recursion

- 2 productions with **common "left factor"**:

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \qquad \text{where } \alpha \neq \epsilon$$

Left-recursive and unfactored grammars

At the current point in the presentation, the importance of those conditions might not yet be clear (but remember the discussion around “oracular” derivations). In general, it’s that certain kind of parsing techniques require absence of left-recursion and of common left-factors. Note also that a left-linear production is a special case of a production with immediate left recursion. In particular, recursive descent parsers would not work with left-recursion. For that kind of parsers, left-recursion needs to be avoided.

Why common left-factors are undesirable should at least intuitively be clear: we see this also on the next slide (the two forms of conditionals). It’s intuitively clear, that a parser, when encountering an **if** (and the following boolean condition and perhaps the **then** clause) cannot decide immediately which rule applies. It should also be intiutively clear that that’s what a parser does: inputting a stream of tokens and trying to figure out which sequence of rules are responsible for that stream (or else reject the input). The amount of additional information, at each point of the parsing process, to *determine* which rule is responsible next is called the *look-ahead*. Of course, if the grammar is ambiguous, no unique decision may be possible (no matter the look-ahead). Ambiguous grammars are unwelcome as specification for parsers.

On a very high-level, the situation can be compared with the situation for regular languages/automata. Non-deterministic automata may be ok for *specifying a language* (they can more easily be connected to regular expressions), but they are not so useful for specifying a scanner *program*. There, deterministic automata are necessary. Here, grammars with left-recursion, grammars with common factors, or even ambiguous grammars may be ok for specifying a context-free language. For instance, ambiguity may be caused by unspecified precedences or non-associativity. Nonetheless, how to obtain a grammar representation more suitable to be more or less directly translated to a parser is an issue less clear cut compared to regular languages. Already the question whether or not a given grammar is ambiguous or not is undecidable. If ambiguous, there’d be no point in turning it into a practical parser. Also the question, what’s an acceptable form of grammar depends on what class of parsers one is after (like a top-down parser or a bottom-up parser).

Some simple examples for both

- left-recursion

$$exp \to exp + term$$

- classical example for common left factor: rules for conditionals

$$\begin{array}{rcl} \textit{if-stmt} & \to & \textbf{if}\ \textbf{(}\ exp\ \textbf{)}\ stmt\ \textbf{end} \\ & \mid & \textbf{if}\ \textbf{(}\ exp\ \textbf{)}\ stmt\ \textbf{else}\ stmt\ \textbf{end} \end{array}$$

Transforming the expression grammar

$$\begin{array}{rcl} \mathit{exp} & \to & \mathit{exp}\ \mathit{addop}\ \mathit{term} \mid \mathit{term} \\ \mathit{addop} & \to & \texttt{+} \mid \texttt{-} \\ \mathit{term} & \to & \mathit{term}\ \mathit{mulop}\ \mathit{factor} \mid \mathit{factor} \\ \mathit{mulop} & \to & \texttt{*} \\ \mathit{factor} & \to & \textbf{(}\ \mathit{exp}\ \textbf{)} \mid \textbf{number} \end{array}$$

- obviously left-recursive
- remember: this variant used for proper **associativity**!

After removing left recursion

$$\begin{array}{rcl} \mathit{exp} & \to & \mathit{term}\,\mathit{exp}' \\ \mathit{exp}' & \to & \mathit{addop}\,\mathit{term}\,\mathit{exp}' \mid \epsilon \\ \mathit{addop} & \to & \texttt{+} \mid \texttt{-} \\ \mathit{term} & \to & \mathit{factor}\,\mathit{term}' \\ \mathit{term}' & \to & \mathit{mulop}\,\mathit{factor}\,\mathit{term}' \mid \epsilon \\ \mathit{mulop} & \to & \texttt{*} \\ \mathit{factor} & \to & \textbf{(}\ \mathit{exp}\ \textbf{)} \mid \mathbf{n} \end{array}$$

- still *unambiguous*
- unfortunate: *associativity* now different!
- note also: ϵ-productions & nullability

Left-recursion removal

Left-recursion removal

A transformation process to turn a CFG into one without left recursion

Explanation

- price: ϵ-productions
- *3 cases* to consider
  - immediate (or direct) recursion
    - simple
    - general
  - *indirect* (or mutual) recursion

Left-recursion removal: simplest case

Before

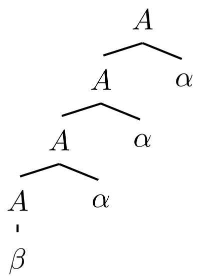
$$A \to A\alpha \mid \beta$$

After

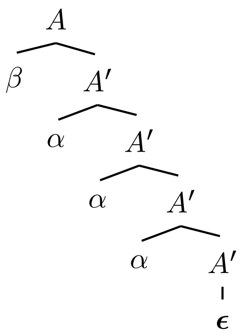
$$\begin{array}{rcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \epsilon \end{array}$$

Schematic representation

$$A \rightarrow A\alpha \mid \beta$$



$$\begin{array}{rcl} A & \rightarrow & \beta A' \\ A' & \rightarrow & \alpha A' \mid \epsilon \end{array}$$



Remarks

- both grammars generate the same (context-free) language (= set of words over terminals)
- in EBNF:

$$A \rightarrow \beta\{\alpha\}$$

- two *negative* aspects of the transformation
  1. generated language unchanged, but: change in resulting structure (parse-tree), i.a.w. change in **associativity**, which may result in change of *meaning*
  2. introduction of ϵ-productions
- more concrete example for such a production: grammar for expressions

Left-recursion removal: immediate recursion (multiple)

Before

$$\begin{array}{rcl} A & \rightarrow & A\alpha_1 \mid \ldots \mid A\alpha_n \\ & \mid & \beta_1 \mid \ldots \mid \beta_m \end{array}$$

space

After

$$
\begin{array}{rcl}
A & \to & \beta_1 A' \mid \ldots \mid \beta_m A' \\
A' & \to & \alpha_1 A' \mid \ldots \mid \alpha_n A' \\
& \mid & \epsilon
\end{array}
$$

EBNF

Note: can be written in *EBNF* as:

$$A \to (\beta_1 \mid \ldots \mid \beta_m)(\alpha_1 \mid \ldots \mid \alpha_n)^*$$

Removal of: general left recursion

Assume non-terminals $A_1, \ldots, A_m$

```
for i := 1 to m do
  for j := 1 to i-1 do
    replace each grammar rule of the form $A_i \to A_j\beta$ by // $i < j$
    rule $A_i \to \alpha_1\beta \mid \alpha_2\beta \mid \ldots \mid \alpha_k\beta$
       where $A_j \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$
       is the current rule(s) for $A_j$ // current
  end
  { corresponds to $i = j$ }
  remove, if necessary, immediate left recursion for $A_i$
end
```

"current" = rule in the current stage of algo

Example (for the general case)

let $A = A_1$, $B = A_2$

$$
\begin{array}{rcl}
A & \to & B\mathbf{a} \mid A\mathbf{a} \mid \mathbf{c} \\
B & \to & B\mathbf{b} \mid A\mathbf{b} \mid \mathbf{d}
\end{array}
$$

$$
\begin{array}{rcl}
A & \to & B\mathbf{a}A' \mid \mathbf{c}A' \\
A' & \to & \mathbf{a}A' \mid \epsilon \\
B & \to & B\mathbf{b} \mid A\mathbf{b} \mid \mathbf{d}
\end{array}
$$

$$
\begin{array}{rcl}
A & \to & B\mathbf{a}A' \mid \mathbf{c}A' \\
A' & \to & \mathbf{a}A' \mid \epsilon \\
B & \to & B\mathbf{b} \mid B\mathbf{a}A'\mathbf{b} \mid \mathbf{c}A'\mathbf{b} \mid \mathbf{d}
\end{array}
$$

$$\begin{array}{lcl} A & \to & B\mathbf{a}A' \mid \mathbf{c}A' \\ A' & \to & \mathbf{a}A' \mid \epsilon \\ B & \to & \mathbf{c}A'\mathbf{b}B' \mid \mathbf{d}B' \\ B' & \to & \mathbf{b}B' \mid \mathbf{a}A'\mathbf{b}B' \mid \epsilon \end{array}$$

Left factor removal

- CFG: not just describe a context-free languages
- also: intended (indirect) description of a **parser** for that language

⇒ common left factor undesirable

- cf.: *determinization* of automata for the lexer

Simple situation

1. before

$$A \to \alpha\beta \mid \alpha\gamma \mid \ldots$$

2. after

$$\begin{array}{lcl} A & \to & \alpha A' \mid \ldots \\ A' & \to & \beta \mid \gamma \end{array}$$

Example: sequence of statements

sequences of statements

1. Before

$$\begin{array}{lcl} \textit{stmt-seq} & \to & \textit{stmt}\,\mathbf{;}\,\textit{stmt-seq} \\ & \mid & \textit{stmt} \end{array}$$

2. After

$$\begin{array}{lcl} \textit{stmt-seq} & \to & \textit{stmt}\ \textit{stmt-seq}' \\ \textit{stmt-seq}' & \to & \mathbf{;}\,\textit{stmt-seq} \mid \epsilon \end{array}$$

Example: conditionals

1. Before

$$\begin{array}{lcl} \textit{if-stmt} & \to & \mathbf{if}\ \mathbf{(}\ \textit{exp}\ \mathbf{)}\ \textit{stmt-seq}\ \mathbf{end} \\ & \mid & \mathbf{if}\ \mathbf{(}\ \textit{exp}\ \mathbf{)}\ \textit{stmt-seq}\ \mathbf{else}\ \textit{stmt-seq}\ \mathbf{end} \end{array}$$

2. After

$$\begin{array}{lcl} \textit{if-stmt} & \to & \mathbf{if}\ \mathbf{(}\ \textit{exp}\ \mathbf{)}\ \textit{stmt-seq}\ \textit{else-or-end} \\ \textit{else-or-end} & \to & \mathbf{else}\ \textit{stmt-seq}\ \mathbf{end} \mid \mathbf{end} \end{array}$$

Example: conditionals (without else)

1. Before

$$\begin{array}{rcl} \textit{if-stmt} & \to & \textbf{if}\ (\ \textit{exp}\ )\ \textit{stmt-seq} \\ & | & \textbf{if}\ (\ \textit{exp}\ )\ \textit{stmt-seq}\ \textbf{else}\ \textit{stmt-seq} \end{array}$$

2. After

$$\begin{array}{rcl} \textit{if-stmt} & \to & \textbf{if}\ (\ \textit{exp}\ )\ \textit{stmt-seq}\ \textit{else-or-empty} \\ \textit{else-or-empty} & \to & \textbf{else}\ \textit{stmt-seq}\ \mid\ \boldsymbol{\epsilon} \end{array}$$

Not all factorization doable in "one step"

1. Starting point

$$A \ \to\ \mathbf{abc}B \mid \mathbf{ab}C \mid \mathbf{a}E$$

2. After 1 step

$$\begin{array}{rcl} A & \to & \mathbf{ab}A' \mid \mathbf{a}E \\ A' & \to & \mathbf{c}B \mid C \end{array}$$

3. After 2 steps

$$\begin{array}{rcl} A & \to & \mathbf{a}A'' \\ A'' & \to & \mathbf{b}A' \mid E \\ A' & \to & \mathbf{c}B \mid C \end{array}$$

4. longest left factor
   - note: we choose the *longest* common prefix (= longest left factor) in the first step

Left factorization

```
while  there are changes to the grammar  do
   for  each nonterminal $A$  do
      let  $\alpha$ be a prefix of max. length that is shared
                    by two or more productions for $A$
      if    $\alpha \neq \epsilon$
      then
           let  $A \to \alpha_1 \mid \dots \mid \alpha_n$ be all
                          prod. for $A$ and suppose that $\alpha_1, \dots, \alpha_k$ share $\alpha$
                          so that $A \to \alpha\beta_1 \mid \dots \mid \alpha\beta_k \mid \alpha_{k+1} \mid \dots \mid \alpha_n$ ,
                          that the $\beta_j$'s share no common prefix, and
                          that the $\alpha_{k+1}, \dots, \alpha_n$ do not share $\alpha$.
           replace rule $A \to \alpha_1 \mid \dots \mid \alpha_n$ by the rules
           $A \to \alpha A' \mid \alpha_{k+1} \mid \dots \mid \alpha_n$
           $A' \to \beta_1 \mid \dots \mid \beta_k$
      end
   end
end
```

The algorithm is pretty straightforward. This time (unlike many others) it's not a loop "until stabilization", it's a more straighforward "iteration" (with nested loops). The only thing to keep in might is that what is called α in the pseudo-code needs to be the *longest* comment prefix and the β's must include *all* right-hand sides that start with that (common longest prefix) α.

4.4 LL-parsing (mostly LL(1))

After having covered the more technical definitions of the first and follow sets and transformations to remove left-recursion resp. common left factors, we go back to top-down parsing, in particular to the specific form of LL(1) parsing.

Additionally, we discuss issues about abstract syntax trees vs. parse trees.

Parsing LL(1) grammars

- *this lecture*: we don't do LL(k) with $k > 1$
- LL(1): particularly easy to understand and to implement (efficiently)
- not as expressive than LR(1) (see later), but still kind of decent

LL(1) parsing principle

Parse from 1) left-to-right (as always anyway), do a 2) **left-most** derivation and resolve the "which-right-hand-side" non-determinism by

1. looking **1 symbol ahead**.

- two flavors for LL(1) parsing here (both are top-down parsers)
  - *recursive descent*
  - *table-based* LL(1) parser
- *predictive* parsers

If one wants to be very precise: it's recursive descent with one look-ahead and without backtracking. It's the single most common case for recursive descent parsers. Longer look-aheads are possible, but less common. Technically, even allowing back-tracking can be done using recursive descent as principle (even if not done in practice).

Sample expr grammar again

factors and terms

$$\begin{array}{rcl} exp & \to & term\ exp' \\ exp' & \to & addop\ term\ exp' \mid \epsilon \\ addop & \to & \mathtt{+} \mid \mathtt{-} \\ term & \to & factor\ term' \\ term' & \to & mulop\ factor\ term' \mid \epsilon \\ mulop & \to & \mathtt{*} \\ factor & \to & \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{n} \end{array} \tag{4.6}$$

Look-ahead of 1: straightforward, but not trivial

- look-ahead of 1:
  - not much of a look-ahead, anyhow
  - just the "current token"

⇒ read the next token, and, based on that, decide

- but: what if there's *no more symbols*?

⇒ read the next token if there is, and decide based on the token *or else* the fact that there's none left[4]

Example: 2 productions for non-terminal *factor*

$$factor \to \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}$$

That situation here is more or less *trivial*, but that's not all to LL(1) . . .

Recursive descent: general set-up

1. global variable, say `tok`, representing the "current token" (or pointer to current token)
2. parser has a way to *advance* that to the next token (if there's one)

Idea

For each *non-terminal nonterm*, write one procedure which:

- succeeds, if starting at the current token position, the "rest" of the token stream starts with a syntactically correct word of terminals representing *nonterm*
- fail otherwise

- ignored (for now): when doing the above successfully, build the *AST* for the accepted nonterminal.

[4] Sometimes "special terminal" **$** used to mark the end (as mentioned).

Recursive descent (in C-like)

method `factor` for nonterminal *factor*

```
final int LPAREN=1,RPAREN=2,NUMBER=3,
 PLUS=4,MINUS=5,TIMES=6;
```

```
void factor () {
    switch (tok) {
    case LPAREN: eat(LPAREN);expr();eat(RPAREN);
    case NUMBER: eat(NUMBER);
    }
}
```

Recursive descent (in ocaml)

```
type token = LPAREN | RPAREN |  NUMBER
 |    PLUS | MINUS | TIMES
```

```
let factor () =     (* function for factors *)
  match !tok with
    LPAREN ->  eat(LPAREN); expr(); eat(RPAREN)
  | NUMBER ->  eat(NUMBER)
```

Slightly more complex

- previous 2 rules for *factor*: situation not always as immediate as that

LL(1) principle (again)

given a non-terminal, the next *token* must determine the choice of right-hand side[5]

⇒ definition of the *First* **set**

Lemma 4.4.1 (LL(1) (without nullable symbols))**.** *A reduced context-free grammar without nullable non-terminals is an LL(1)-grammar iff for all non-terminals A and for all pairs of productions $A \to \alpha_1$ and $A \to \alpha_2$ with $\alpha_1 \neq \alpha_2$:*

$$First_1(\alpha_1) \cap First_1(\alpha_2) = \emptyset \ .$$

[5] It must be the next token/terminal in the sense of *First*, but it need not be a token *directly* mentioned on the right-hand sides of the corresponding rules.

Common problematic situation

- often: common *left factors* problematic

$$\begin{array}{rcl} \textit{if-stmt} & \rightarrow & \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt} \\ & \mid & \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \end{array}$$

- requires a look-ahead of (at least) 2
- ⇒ try to rearrange the grammar
  1. *Extended* BNF ([9] suggests that)
     $$\textit{if-stmt} \rightarrow \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt}[\textbf{else}\ \textit{stmt}]$$
  1. *left-factoring*:

$$\begin{array}{rcl} \textit{if-stmt} & \rightarrow & \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt}\ \textit{else-part} \\ \textit{else-part} & \rightarrow & \epsilon \mid \textbf{else}\ \textit{stmt} \end{array}$$

Recursive descent for left-factored *if-stmt*

```
procedure ifstmt()
  begin
    match ("if");
    match  ("(");
    exp();
    match (")");
    stmt();
    if   token = "else"
    then match ("else");
         stmt()
    end
  end;
```

Left recursion is a no-go

factors and terms

$$\begin{array}{rcl} \textit{exp} & \rightarrow & \textit{exp}\ \ \textit{addop}\ \ \textit{term} \mid \textit{term} \\ \textit{addop} & \rightarrow & \texttt{+} \mid \texttt{-} \\ \textit{term} & \rightarrow & \textit{term}\ \ \textit{mulop}\ \ \textit{factor} \mid \textit{factor} \\ \textit{mulop} & \rightarrow & \texttt{*} \\ \textit{factor} & \rightarrow & \textbf{(}\ \textit{exp}\ \textbf{)} \mid \textbf{number} \end{array} \tag{4.7}$$

- consider treatment of *exp*: *First*(*exp*)?
- whatever is in *First*(*term*), is in *First*(*exp*)[6] recursion.

[6] And it would not help to *look-ahead* more than 1 token either.

Left-recursion

Left-recursive grammar *never* works for recursive descent.

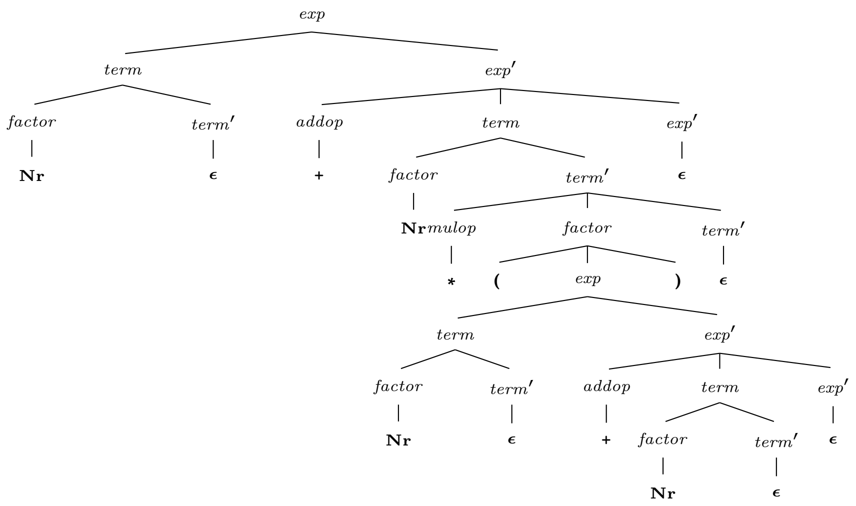
Removing left recursion may help

$$\begin{array}{rcl}
\mathit{exp} & \rightarrow & \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' & \rightarrow & \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \boldsymbol{\epsilon} \\
\mathit{addop} & \rightarrow & \mathbf{+} \mid \mathbf{-} \\
\mathit{term} & \rightarrow & \mathit{factor}\ \mathit{term}' \\
\mathit{term}' & \rightarrow & \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \boldsymbol{\epsilon} \\
\mathit{mulop} & \rightarrow & \mathbf{*} \\
\mathit{factor} & \rightarrow & \mathbf{(}\ \mathit{exp}\ \mathbf{)} \mid \mathbf{n}
\end{array}$$

```
procedure exp()
begin
      term();
      exp'()
end
```

```
procedure exp'()
begin
   case token of
      "+": match("+");
           term();
           exp'()
      "-": match("-");
           term();
           exp'()
    end
end
```

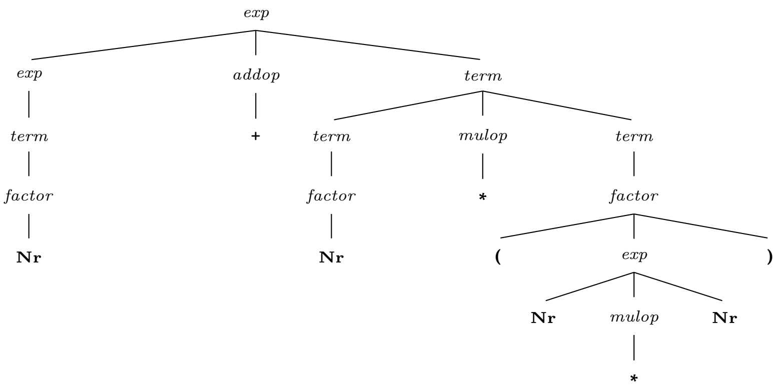
Recursive descent works, alright, but . . .



. . . who wants this form of trees?

Left-recursive grammar with nicer parse trees

$$1 + 2 * (3 + 4)$$

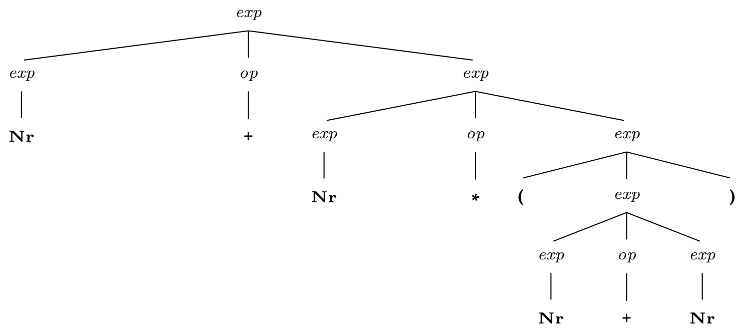


The simple "original" expression grammar (even nicer)

Flat expression grammar

$$\begin{array}{lcl} exp & \rightarrow & exp\ op\ exp \mid (\ exp\ ) \mid \textbf{number} \\ op & \rightarrow & + \mid - \mid * \end{array}$$

$$1 + 2 * (3 + 4)$$



Associtivity problematic

Precedence & assoc.

$$
\begin{array}{rcl}
\textit{exp} & \rightarrow & \textit{exp}\ \textit{addop}\ \textit{term} \mid \textit{term} \\
\textit{addop} & \rightarrow & \texttt{+} \mid \texttt{-} \\
\textit{term} & \rightarrow & \textit{term}\ \textit{mulop}\ \textit{factor} \mid \textit{factor} \\
\textit{mulop} & \rightarrow & \texttt{*} \\
\textit{factor} & \rightarrow & \textbf{(}\ \textit{exp}\ \textbf{)} \mid \textbf{number}
\end{array}
$$

Formula

$$3 + 4 + 5$$

parsed “as”

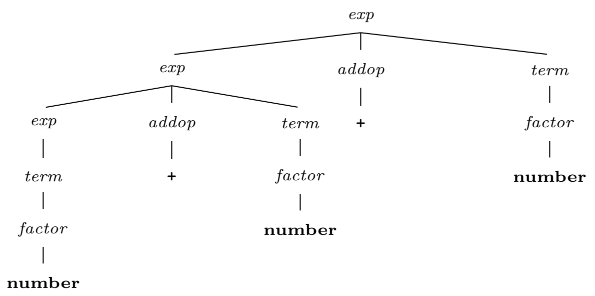
$$(3 + 4) + 5$$

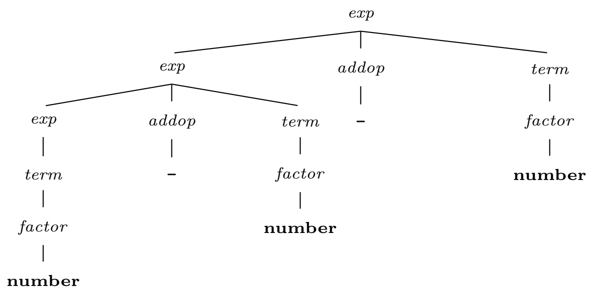
$$3 - 4 - 5$$

parsed “as”

$$(3 - 4) - 5$$

Tree





Now use the grammar without left-rec (but right-rec instead)

No left-rec.

$$\begin{array}{rcl}
\mathit{exp} & \rightarrow & \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' & \rightarrow & \mathit{addop}\ \mathit{term}\ \mathit{exp}' \ \mid\ \epsilon \\
\mathit{addop} & \rightarrow & \texttt{+} \ \mid\ \texttt{-} \\
\mathit{term} & \rightarrow & \mathit{factor}\ \mathit{term}' \\
\mathit{term}' & \rightarrow & \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \ \mid\ \epsilon \\
\mathit{mulop} & \rightarrow & \texttt{*} \\
\mathit{factor} & \rightarrow & \textbf{(}\ \mathit{exp}\ \textbf{)} \ \mid\ \textbf{n}
\end{array}$$

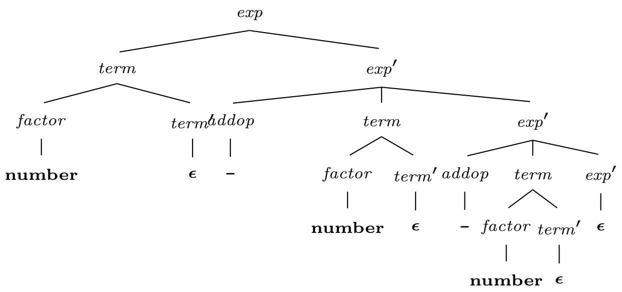
Formula

$$3 - 4 - 5$$

parsed “as”

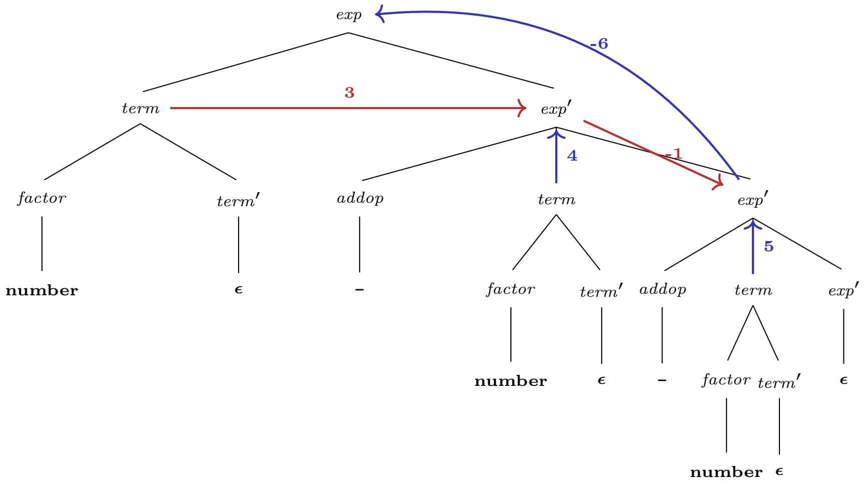
$$3 - (4 - 5)$$

Tree



But if we need a “left-associative” AST?

- we want $(3-4)-5$, *not* $3-(4-5)$



Code to “evaluate” ill-associated such trees correctly

```
function exp′ (valsofar: int): int;
begin
  if token = '+' or token = '-'
  then
    case token of
      '+': match ('+');
           valsofar := valsofar + term;
      '-': match ('-');
           valsofar := valsofar - term;
    end case;
    return exp′(valsofar);
  else return valsofar
end;
```

- extra "accumulator" argument `valsofar`
- instead of evaluating the expression, one could build the AST with the appropriate associativity instead:
- instead of `valueSoFar`, one had `rootOfTreeSoFar`

The example parses expressions and *evalutes* them while doing that. In most cases in a full-fledged parser, one does not need a value as output of a successful parse-run, but an AST. But the issue of the fact, that sometimes the associativity is "the wrong way". Also the "accumulator"-pattern illustrated here in the evaluation setting could help out with AST

"Designing" the syntax, its parsing, & its AST :B$_{\text{frame}}$:x

trade offs:

1. starting from: design of the language, how much of the syntax is left "implicit"[7]
2. which language class? Is LL(1) good enough, or something stronger wanted?
3. how to parse? (top-down, bottom-up, etc.)
4. parse-tree/concrete syntax trees vs. ASTs

AST vs. CST

- once steps 1.–3. are fixed: *parse-trees* fixed!
- parse-trees = *essence* of grammatical derivation process
- often: parse trees only "conceptually" present in a parser
- AST:
  - *abstractions* of the parse trees
  - *essence* of the parse tree
  - actual tree data structure, as output of the parser
  - typically on-the fly: AST built while the parser parses, i.e. while it executes a derivation in the grammar

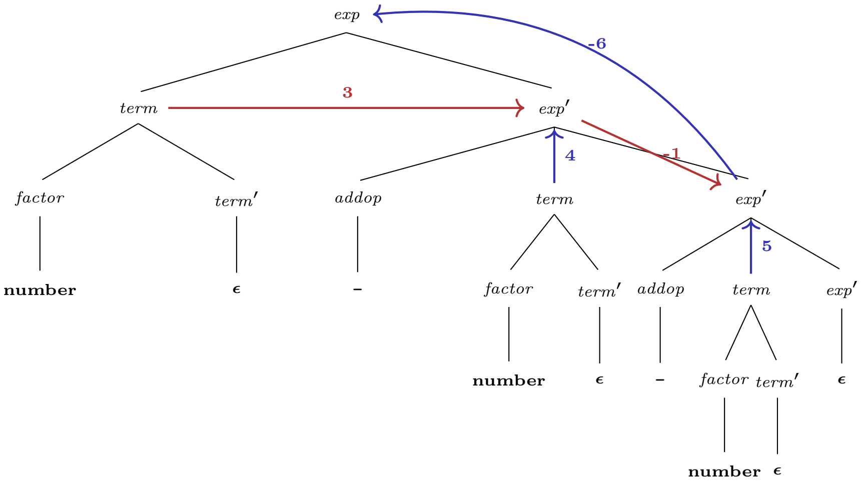
AST vs. CST/parse tree

Parser "**builds**" the AST data structure while "**doing**" the parse tree

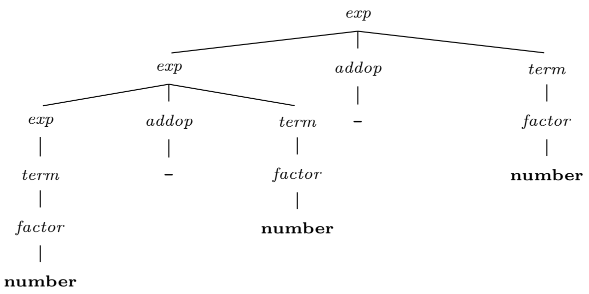
AST: How "far away" from the CST?

- AST: only thing relevant for later phases ⇒ better be *clean* ...
- AST "=" CST?
  - building AST becomes straightforward
  - possible choice, **if** the grammar is not designed "weirdly",

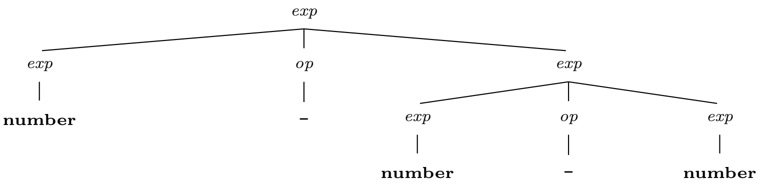
[7] Lisp is famous/notorious in that its surface syntax is more or less an explicit notation for the ASTs. Not that it was originally planned like this ...



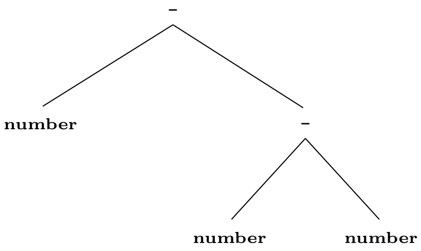
parse-trees like that better be cleaned up as AST

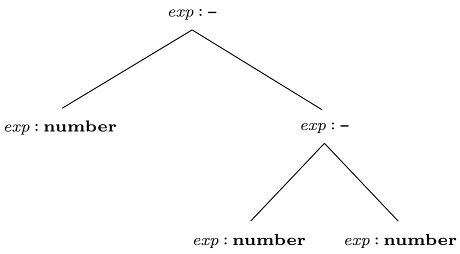


slightly more reasonably looking as AST (but underlying grammar not directly useful for recursive descent)



That parse tree looks reasonable clear and intuitive





Certainly minimal amount of nodes, which is nice as such. However, what is missing (which might be interesting) is the fact that the 2 nodes labelled "−" are *expressions!*

This is how it's done (a recipe)

Assume, one has a "non-weird" grammar

$$\begin{array}{rcl} exp & \rightarrow & exp\ \ op\ \ exp \mid \texttt{(}\, exp\, \texttt{)} \mid \textbf{number} \\ op & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \end{array}$$

- typically that means: assoc. and precedences etc. are fixed *outside* the non-weird grammar
  - by massaging it to an equivalent one (no left recursion etc.)
  - or (better): use parser-generator that allows to *specify* assoc . . . like " "*" *binds stronger* than "+", it *associates* to the left . . . " , without cluttering the grammar.
- if grammar for *parsing* is not as clear: do a second one describing the ASTs

Remember (independent from parsing)

BNF describe **trees**

This is how it's done (recipe for OO data structures)

Recipe

- turn each **non-terminal** to an **abstract class**
- turn each **right-hand** side of a given non-terminal as (non-abstract) **subclass** of the class for considered non-terminal
- chose fields & constructors of concrete classes appropriately
- **terminal**: concrete class as well, field/constructor for token's *value*

Example in Java

$$\begin{array}{rcl} exp & \to & exp\ op\ exp \mid \mathbf{(}\, exp\, \mathbf{)} \mid \mathbf{number} \\ op & \to & \mathtt{+} \mid \mathtt{-} \mid \mathtt{*} \end{array}$$

```
abstract public class Exp {
}
```

```
public class BinExp extends Exp {  // exp -> exp op exp
   public Exp left, right;
   public Op  op;
   public BinExp(Exp l, Op o, Exp r) {
       left=l; op=o; right=r;}
}
```

```
public class ParentheticExp extends Exp {  // exp -> ( op )
   public Exp exp;
   public ParentheticExp(Exp e) {exp = l;}
}
```

```
public class NumberExp extends Exp {  // exp -> NUMBER
    public  number;                     // token value
    public Number(int i) {number = i;}
}
```

```
abstract public class Op {   // non-terminal = abstract
}
```

```
public class Plus  extends Op {  // op -> "+"
}
```

```
public class Minus  extends Op {  // op -> "-"
}
```

```
public class Times extends Op {  // op -> "*"
}
```

The latter classes are perhaps pushing it too far. It's done to show that one can mechanically use the *recipe* once grammar is given, so it's a clean solution (perhaps one get better efficiency if one would not make classes / objects out of everything, though).

3 – (4 – 5)

```
Exp e =  new BinExp(
            new NumberExp(3),
            new Minus(),
            new BinExp(new ParentheticExpr(
                new NumberExp(4),
                new Minus(),
                new NumberExp(5))))
```

Pragmatic deviations from the recipe

- it's nice to have a guiding principle, but no need to carry it too far …
- To the very least: the `ParentheticExpr` is completely without purpose: grouping is captured by the tree structure

⇒ that class is *not* needed

- some might prefer an implementation of

$$op \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*}$$

as simply integers, for instance arranged like

```
public class BinExp extends Exp {  // exp -> exp op exp
   public Exp left , right;
   public int  op;
   public BinExp(Exp l, int o, Exp r) {
     pos=p; left=l; oper=o; right=r;}
   public final static int PLUS=0, MINUS=1, TIMES=2;
}
```

and used as `BinExpr.PLUS` etc.

Recipe for ASTs, final words:

- space considerations for AST representations are irrelevant in most cases
- clarity and cleanness trumps "quick hacks" and "squeezing bits"
- some deviation from the recipe or not, the advice still holds:

Do it systematically

A clean grammar is **the** specification of the syntax of the language and thus the parser. It is also a means of **communicating** with humans what the syntax of the language is, at least communicating with pros, like participants of a compiler course, who of course can read BNF … A clean grammar is a very systematic and structured thing which consequently *can* and *should* be **systematically** and **cleanly** represented in an AST, including judicious and systematic choice of names and conventions (nonterminal *exp* represented by class `Exp`, non-terminal *stmt* by class `Stmt` etc)

Extended BNF may help alleviate the pain

BNF

$$\begin{array}{rcl} exp & \rightarrow & exp\ addop\ term \mid term \\ term & \rightarrow & term\ mulop\ factor \mid factor \end{array}$$

EBNF

$$\begin{array}{rcl} exp & \rightarrow & term\{ \; addop \; term \; \} \\ term & \rightarrow & factor\{ \; mulop \; factor \; \} \end{array}$$

but remember:

- EBNF just a notation, just because we do not see (left or right) recursion in $\{ \ldots \}$, does not mean there is no recursion.
- not all parser generators support EBNF
- however: often easy to translate into loops- [8]
- does not offer a *general* solution if associativity etc. is problematic

Pseudo-code representing the EBNF productions

```
procedure exp;
begin
  term ;        { recursive call }
  while token = "+" or token = "-"
  do
    match(token);
    term;       // recursive call
  end
end
```

```
procedure term;
begin
  factor;        { recursive call }
  while token = "*"
  do
    match(token);
    factor;       // recursive call
  end
end
```

How to produce “something” during RD parsing?

Recursive descent

So far (mostly): RD = top-down (parse-)tree traversal via recursive procedure.[9] Possible outcome: termination or failure.

- Now: instead of returning “nothing” (return type `void` or similar), return some meaningful, and build that up during traversal
- for illustration: procedure for expressions:
  - return type `int`,
  - while traversing: *evaluate* the expression

[8] That results in a parser which is somehow not “pure recursive descent”. It’s “recursive descent, but sometimes, let’s use a while-loop, if more convenient concerning, for instance, associativity”

[9] Modulo the fact that the tree being traversed is “conceptual” and not the input of the traversal procedure; instead, the traversal is “steered” by stream of tokens.

Evaluating an *exp* during RD parsing

```
function exp() : int;
var temp: int
begin
  temp := term ();        { recursive call }
  while token = "+" or token = "-"
    case token  of
      "+": match ("+");
           temp := temp + term();
      "-": match ("-")
           temp := temp - term();
    end
  end
  return temp;
end
```

Building an AST: expression

```
function exp() : syntaxTree;
var temp, newtemp: syntaxTree
begin
  temp := term ();        { recursive call }
  while token = "+" or token = "-"
    case token  of
      "+": match ("+");
           newtemp := makeOpNode("+");
           leftChild(newtemp)  := temp;
           rightChild(newtemp) := term();
           temp := newtemp;
      "-": match ("-")
           newtemp := makeOpNode("-");
           leftChild(newtemp)  := temp;
           rightChild(newtemp) := term();
           temp := newtemp;
    end
  end
  return temp;
end
```

- note: the use of `temp` and the `while` loop

Building an AST: factor

$$\mathit{factor} \to \textbf{(}\ \mathit{exp}\ \textbf{)} \ \mid\ \textbf{number}$$

```
function factor() : syntaxTree;
var fact: syntaxTree
begin
  case token  of
    "(": match ("(");
         fact := exp();
         match (")");
    number:
        match (number)
        fact := makeNumberNode(number);
    else : error ...    // fall through
  end
  return fact;
end
```

Building an AST: conditionals

$$\textit{if-stmt} \to \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt}\ [\textbf{else}\ \textit{stmt}]$$

```
function ifStmt() : syntaxTree;
var temp: syntaxTree
begin
  match ("if");
  match ("(");
  temp := makeStmtNode("if")
  testChild(temp) := exp();
  match (")");
  thenChild(temp) := stmt();
  if   token = "else"
  then match "else";
       elseChild(temp) := stmt();
  else elseChild(temp) := nil;
  end
  return temp;
end
```

Building an AST: remarks and "invariant"

- LL(1) requirement: each procedure/function/method (covering one specific non-terminal) decides on alternatives, looking only at the current token
- call of function `A` for non-terminal A:
  - upon entry: first terminal symbol for A in `token`
  - upon exit: first terminal symbol *after* the unit derived from A in `token`
- `match("a")` : checks for `"a"` in `token` *and eats* the token (if matched).

LL(1) parsing

- remember LL(1) grammars & LL(1) parsing principle:

LL(1) parsing principle

1 look-ahead enough to resolve "which-right-hand-side" non-determinism.

- instead of recursion (as in RD): *explicit stack*
- decision making: collated into the **LL(1) parsing table**
- LL(1) parsing table:
  - finite data structure M (for instance, a 2 dimensional array)[10]
    $$M : \Sigma_N \times \Sigma_T \to ((\Sigma_N \times \Sigma^*) + \texttt{error})$$
  - $M[A, a] = w$
- we assume: pure BNF

[10] Often, depending on the book, the entry in the parse table does not contain a full rule as here, needed is only the *right-hand-side*. In that case the table is of type $\Sigma_N \times \Sigma_T \to (\Sigma^* + \texttt{error})$.

Construction of the parsing table

Table recipe

1. If $A \to \alpha \in P$ and $\alpha \Rightarrow^* \mathbf{a}\beta$, then add $A \to \alpha$ to table entry $M[A, \mathbf{a}]$
2. If $A \to \alpha \in P$ and $\alpha \Rightarrow^* \boldsymbol{\epsilon}$ and $S\,\$ \Rightarrow^* \beta A \mathbf{a} \gamma$ (where $\mathbf{a}$ is a token (=non-terminal) *or* $\$$), then add $A \to \alpha$ to table entry $M[A, \mathbf{a}]$

Table recipe (again, now using our old friends *First* and *Follow*)

Assume $A \to \alpha \in P$.

1. If $\mathbf{a} \in \mathit{First}(\alpha)$, then add $A \to \alpha$ to $M[A, \mathbf{a}]$.
2. If α is *nullable* and $\mathbf{a} \in \mathit{Follow}(A)$, then add $A \to \alpha$ to $M[A, \mathbf{a}]$.

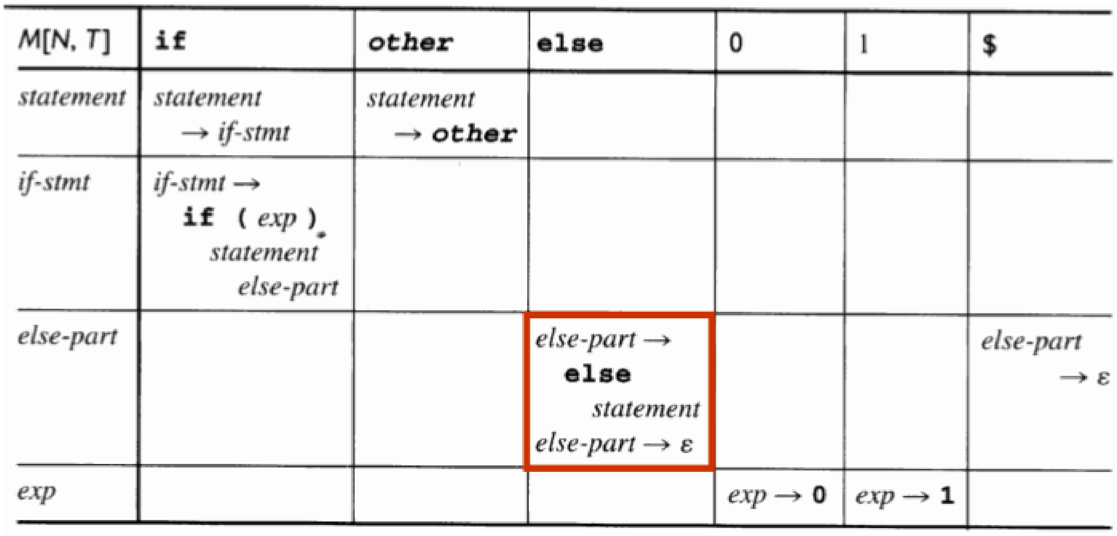
Example: if-statements

- grammars is left-factored and not left recursive

$$
\begin{array}{rcl}
\mathit{stmt} & \to & \mathit{if\text{-}stmt} \mid \mathbf{other} \\
\mathit{if\text{-}stmt} & \to & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmt}\ \mathit{else{-}part} \\
\mathit{else{-}part} & \to & \mathbf{else}\ \mathit{stmt} \mid \epsilon \\
\mathit{exp} & \to & \mathbf{0} \mid \mathbf{1}
\end{array}
$$

	First	*Follow*
stmt	**other**, **if**	$, **else**
if-stmt	**if**	$, **else**
else−part	**else**, ϵ	$, **else**
exp	**0**, **1**	**)**

Example: if statement: "LL(1) parse table"



M[N, T]	**if**	**other**	**else**	0	1	$
statement	*statement* → *if-stmt*	*statement* → **other**				
if-stmt	*if-stmt* → **if** (*exp*) *statement* *else-part*					

else-part			*else-part* → **else** *statement*
			else-part → ε
						else-part → ε

exp				*exp* → **0**	*exp* → **1**	

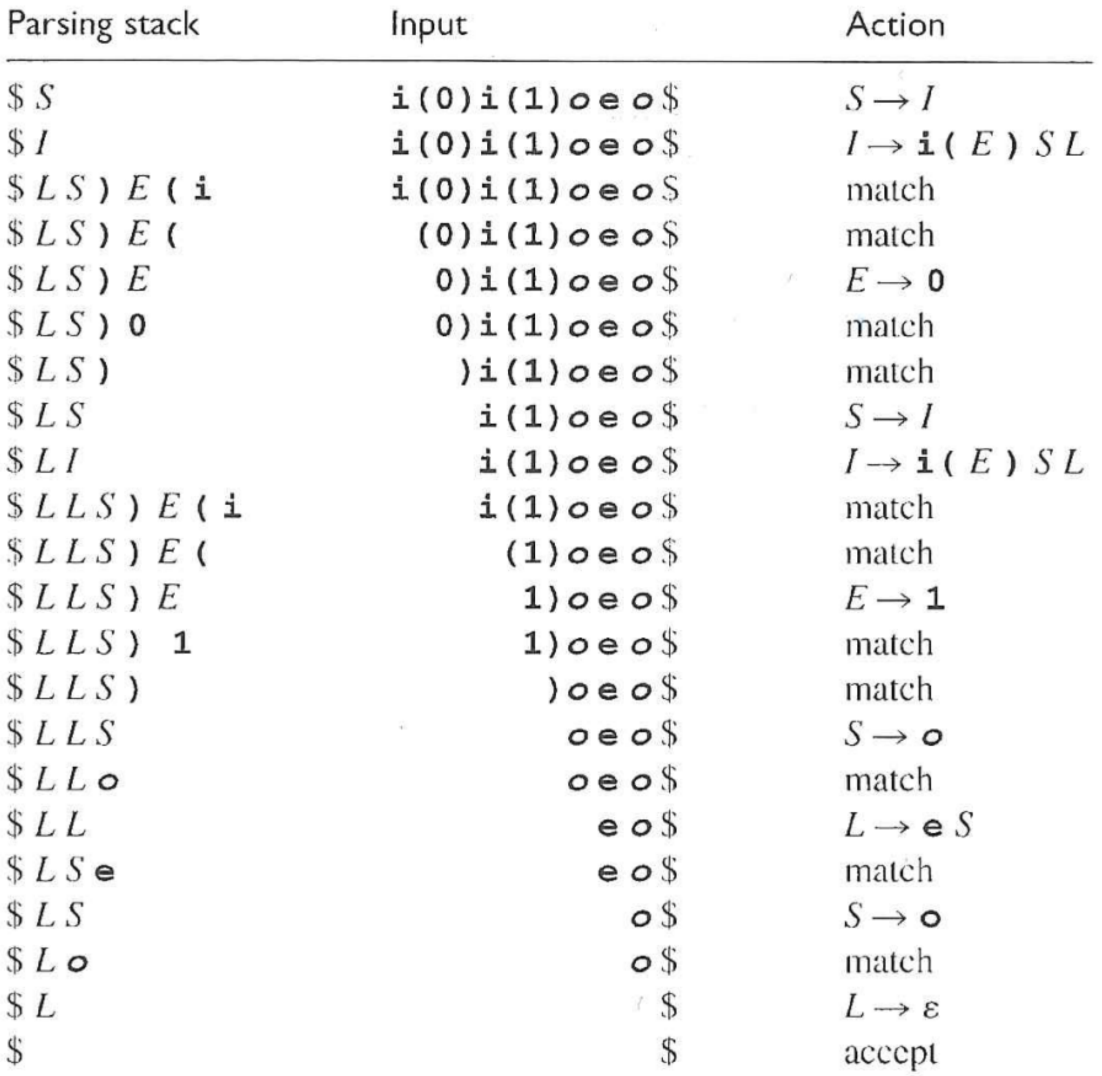
- 2 productions in the “red table entry”
- thus: it’s technically *not* an LL(1) table (and it’s not an LL(1) grammar)
- note: removing left-recursion and left-factoring did not help!

Saying that it’s “not-an-LL(1)-table” is perhaps a bit nit-picking. The shape *is* according to the required format. It’s only that in the slot marked red, there are two rules. That’s a conflict and makes it at least not a legal LL(1) table. So, if in an exam question, the task is “build the LL(1)-table for the following grammar . . . .. Is the grammar LL(1)”. Then one is supposed to fill up a table like that, and then point out, if there is a double entry. to point out that the grammar is not LL(1). Similar remarks later for LR-parsers. Actually, for LR-parsers, tools like `yacc` build up a table (not an LL, but an LR-table) and. in case of double entries, making a choice which one to include. The user, in those cases, will reveive a warning about the grammar containing a corresponding *conflict*.

LL(1) table-based algo

```
while the top of the parsing stack $\neq$ $
   if the top of the parsing stack is terminal  a
      and the next input token  = a
   then
      pop the parsing stack;
      advance the input;  // ``match''
   else if    the top the parsing is non-terminal  $A$
          and the next input token is  a  terminal or  $
          and parsing table  $M[A,\mathbf{a}]$  contains
              production  $A \rightarrow X_1 X_2 \ldots X_n$
          then (* generate *)
              pop the parsing stack
              for  $i := n$  to  1  do
              push  $X_i$  onto the stack;
         else  error
   if    the top of the stack =  $
   then  accept
end
```

LL(1): illustration of run of the algo



Parsing stack	Input	Action
$ S	i(0)i(1)oeo$	S → I
$ I	i(0)i(1)oeo$	I → i (E) S L
$ L S) E (i	i(0)i(1)oeo$	match
$ L S) E (	(0)i(1)oeo$	match
$ L S) E	0)i(1)oeo$	E → 0
$ L S) 0	0)i(1)oeo$	match
$ L S)	)i(1)oeo$	match
$ L S	i(1)oeo$	S → I
$ L I	i(1)oeo$	I → i (E) S L
$ L L S) E (i	i(1)oeo$	match
$ L L S) E (	(1)oeo$	match
$ L L S) E	1)oeo$	E → 1
$ L L S) 1	1)oeo$	match
$ L L S)	)oeo$	match
$ L L S	oeo$	S → o
$ L L o	oeo$	match
$ L L	eo$	L → e S
$ L S e	eo$	match
$ L S	o$	S → o
$ L o	o$	match
$ L	$	L → ε
$	$	accept

The most interesting steps are of course those dealing with the dangling else, namely those with the non-terminal *else–part* at the top of the stack. That's where the LL(1) table is ambiguous. In principle, with *else–part* on top of the stack (in the picture it's just `L`), the parser table allows always to make the decision that the "current statement" resp "current conditional" is done.

Expressions

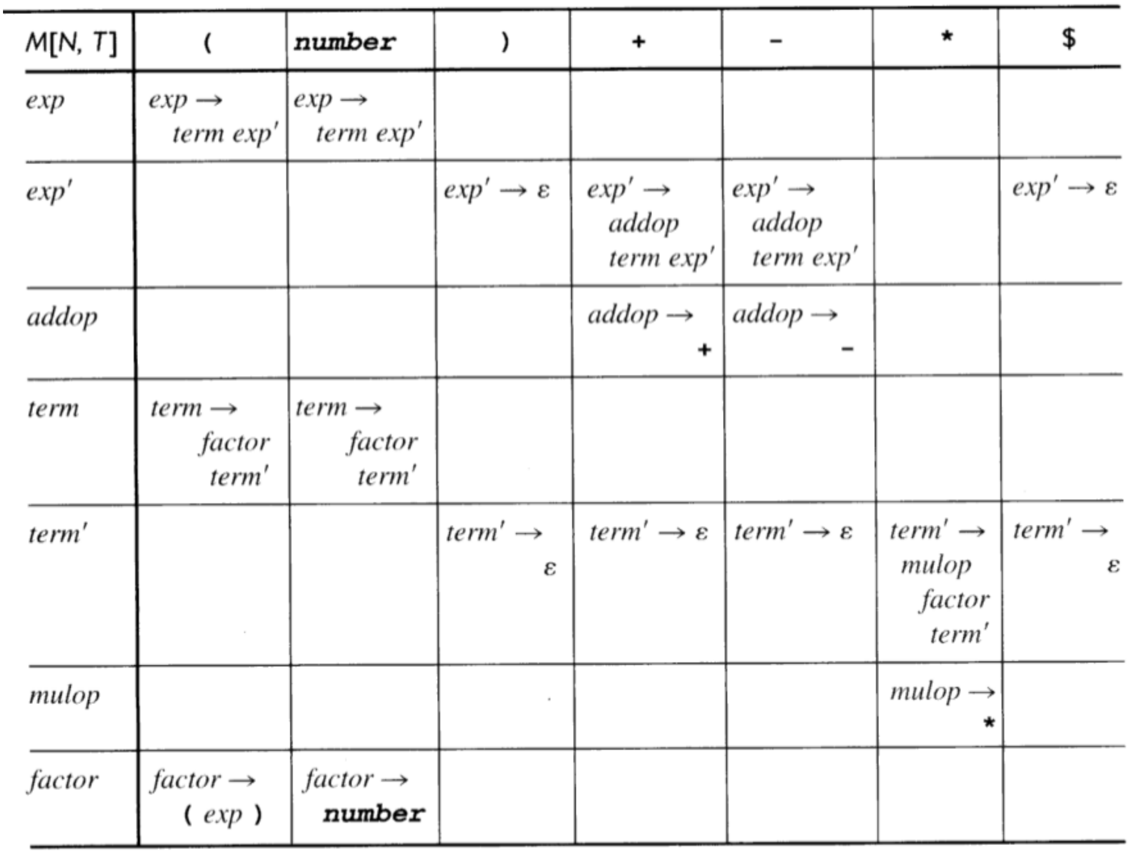
$$\begin{array}{rcl}
\mathit{exp} & \to & \mathit{exp}\ \mathit{addop}\ \mathit{term} \mid \mathit{term} \\
\mathit{addop} & \to & \texttt{+} \mid \texttt{-} \\
\mathit{term} & \to & \mathit{term}\ \mathit{mulop}\ \mathit{factor} \mid \mathit{factor} \\
\mathit{mulop} & \to & \texttt{*} \\
\mathit{factor} & \to & \textbf{(}\ \mathit{exp}\ \textbf{)} \mid \textbf{number}
\end{array}$$

left-recursive ⇒ not LL(k)

$$
\begin{array}{rcl}
\mathit{exp} & \to & \mathit{term}\ \mathit{exp}' \\
\mathit{exp}' & \to & \mathit{addop}\ \mathit{term}\ \mathit{exp}' \mid \epsilon \\
\mathit{addop} & \to & + \mid - \\
\mathit{term} & \to & \mathit{factor}\ \mathit{term}' \\
\mathit{term}' & \to & \mathit{mulop}\ \mathit{factor}\ \mathit{term}' \mid \epsilon \\
\mathit{mulop} & \to & * \\
\mathit{factor} & \to & (\ \mathit{exp}\ ) \mid \mathbf{n}
\end{array}
$$

	First	*Follow*
exp	(, **number**	\$,)
exp′	+, −, ϵ	\$,)
addop	+, −	(, **number**
term	(, **number**	\$,), +, −
term′	∗, ϵ	\$,), +, −
mulop	∗	(, **number**
factor	(, **number**	\$,), +, −, ∗

Expressions: LL(1) parse table



M[N, T]	(	**number**	)	+	-	*	$
exp	*exp* → *term exp′*	*exp* → *term exp′*					
exp′			*exp′* → ε	*exp′* → *addop term exp′*	*exp′* → *addop term exp′*		*exp′* → ε
addop				*addop* → +	*addop* → -		
term	*term* → *factor term′*	*term* → *factor term′*					
term′			*term′* → ε	*term′* → ε	*term′* → ε	*term′* → *mulop factor term′*	*term′* → ε
mulop						*mulop* → *	
factor	*factor* → (*exp*)	*factor* → **number**					

Error handling

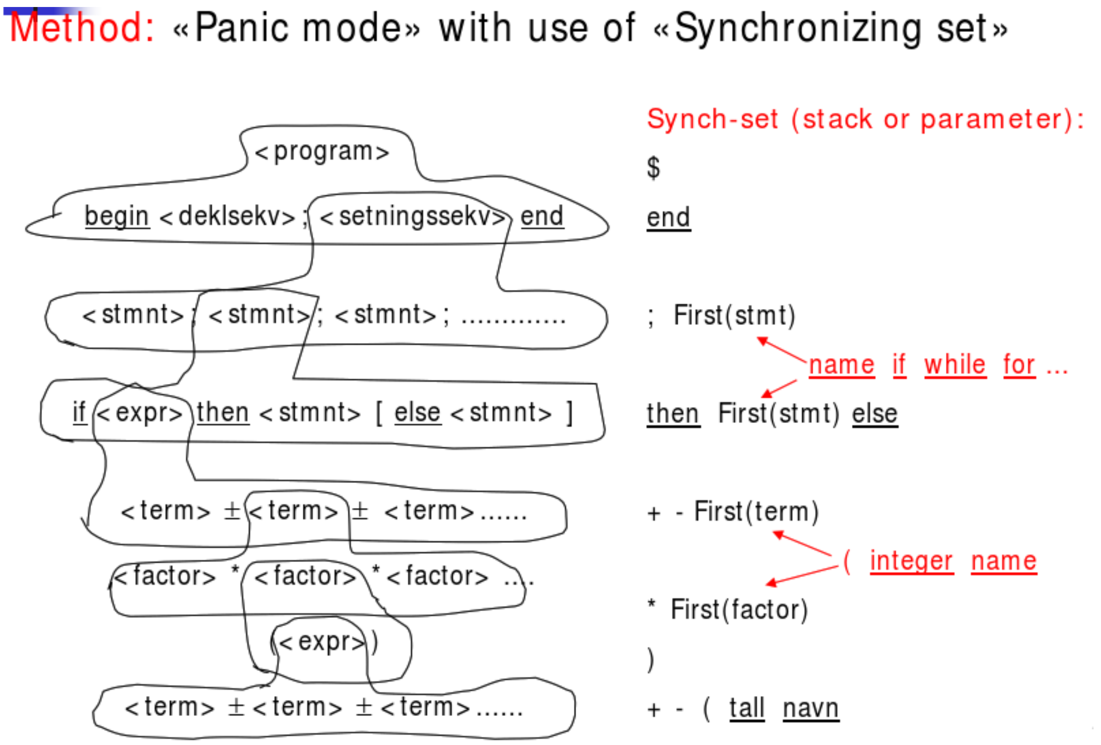
- at the least: do an understandable error message

- give indication of line / character or region responsible for the error in the source file
- potentially *stop* the parsing
- some compilers do *error recovery*
  - give an understandable error message (as minimum)
  - continue reading, until it's plausible to resume parsing ⇒ find more errors
  - however: when finding at least 1 error: no code generation
  - observation: resuming after syntax error is not easy

Error messages

- important:
  - try to avoid error messages that only occur because of an already reported error!
  - report error as early as possible, if possible at the first point where the program cannot be extended to a correct program.
  - make sure that, after an error, one doesn't end up in a infinite loop without reading any input symbols.
- What's a good error message?
  - assume: that the method `factor()` chooses the alternative **(** *exp* **)** but that it, when control returns from method `exp()`, does not find a **)**
  - one could report : `left paranthesis missing`
  - But this may often be confusing, e.g. if what the program text is: `( a + b c )`
  - here the `exp()` method will terminate after `( a + b`, as `c` cannot extend the expression). You should therefore rather give the message `error in expression or left paranthesis missing`.

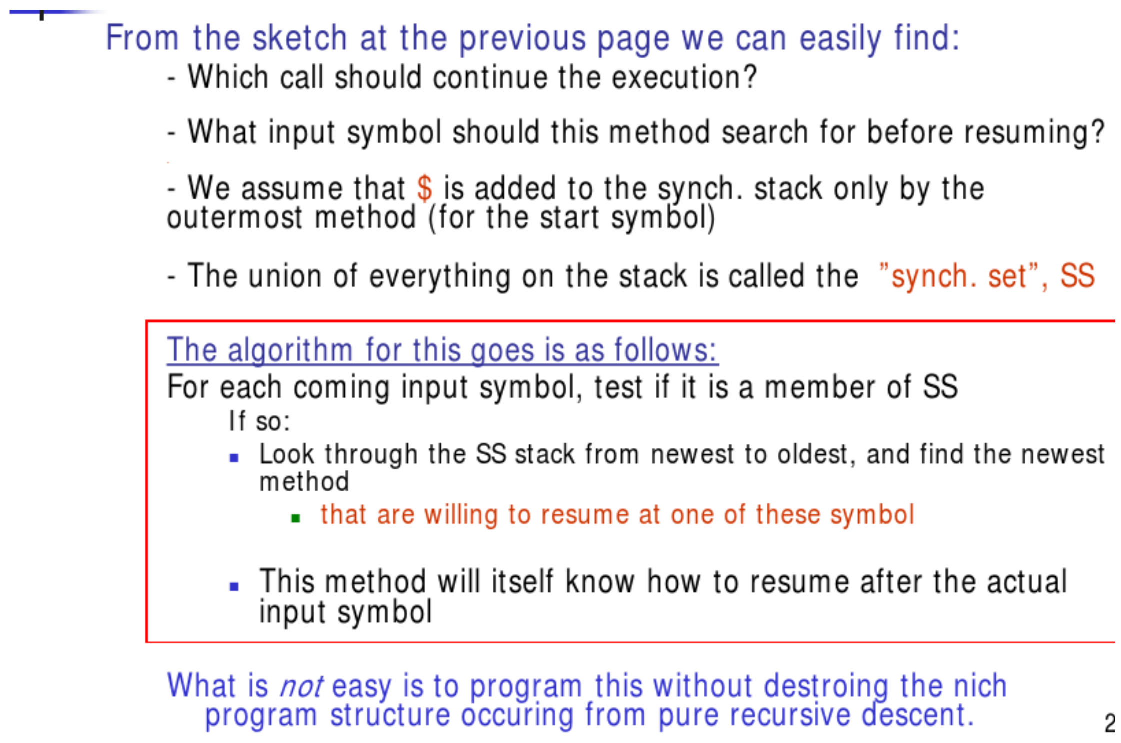
Handling of syntax errors using recursive descent



Method: «Panic mode» with use of «Synchronizing set»

< program >
begin < deklsekv > ; < setningssekv > end
< stmnt > ; < stmnt > ; < stmnt > ; ....
if < expr > then < stmnt > [else < stmnt >]
< term > ± < term > ± < term > ......
< factor > * < factor > * < factor > ...
(< expr >)
< term > ± < term > ± < term > ......
Synch-set (stack or parameter):
$
end
; First(stmt)
name if while for ...
then First(stmt) else
+ - First(term)
(integer name
* First(factor)
)
+ - (tall navn

Syntax errors with sync stack



From the sketch at the previous page we can easily find:

- Which call should continue the execution?
- What input symbol should this method search for before resuming?
- We assume that $ is added to the synch. stack only by the outermost method (for the start symbol)
- The union of everything on the stack is called the "synch. set", SS

The algorithm for this goes is as follows:

For each coming input symbol, test if it is a member of SS

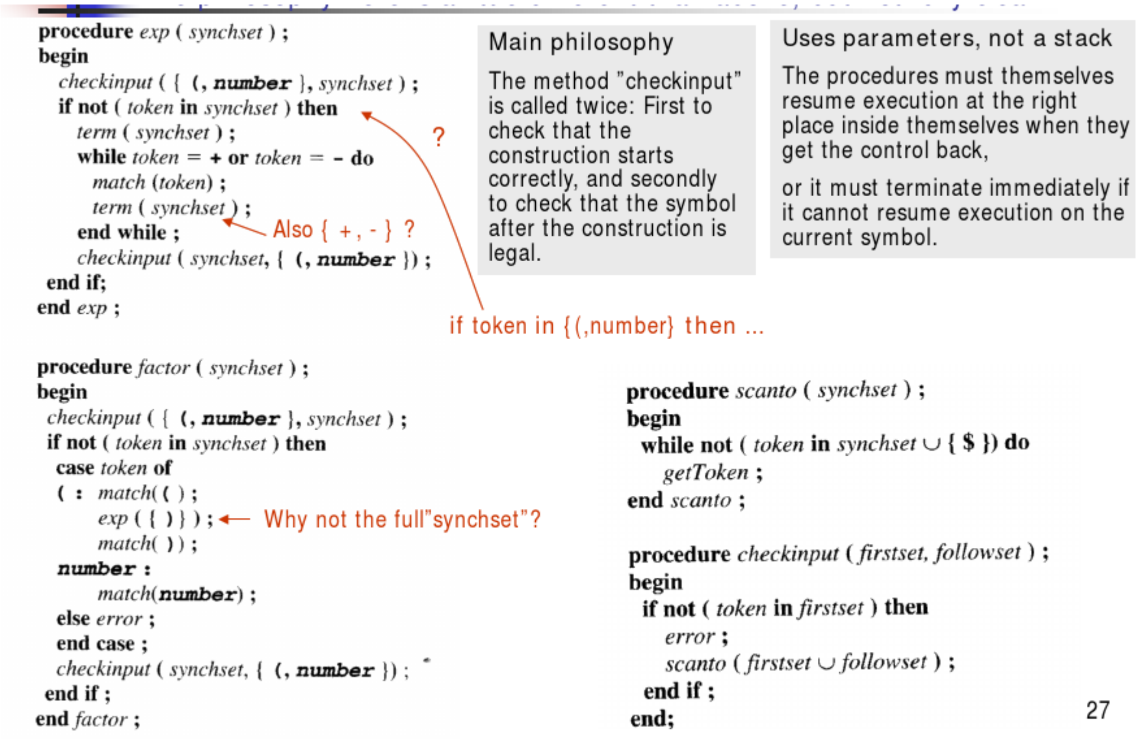
If so:

- Look through the SS stack from newest to oldest, and find the newest method
  - that are willing to resume at one of these symbol
- This method will itself know how to resume after the actual input symbol

What is *not* easy is to program this without destroing the nich program structure occuring from pure recursive descent.

2

Procedures for expression with "error recovery"



```
procedure exp ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    term ( synchset ) ;
    while token = + or token = - do
      match (token) ;
      term ( synchset ) ;
    end while ;
    checkinput ( synchset, { (, number }) ;
 end if;
end exp ;
```

?

Also { + , - } ?

if token in {(,number} then ...

Main philosophy

The method "checkinput" is called twice: First to check that the construction starts correctly, and secondly to check that the symbol after the construction is legal.

Uses parameters, not a stack

The procedures must themselves resume execution at the right place inside themselves when they get the control back,

or it must terminate immediately if it cannot resume execution on the current symbol.

```
procedure factor ( synchset ) ;
begin
 checkinput ( { (, number }, synchset ) ;
 if not ( token in synchset ) then
  case token of
  ( :  match( ( ) ;
       exp ( { ) } ) ;
       match( ) ) ;
  number :
       match(number) ;
  else error ;
  end case ;
  checkinput ( synchset, { (, number }) ;
 end if ;
end factor ;
```

Why not the full"synchset"?

```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset ∪ { $ }) do
    getToken ;
end scanto ;

procedure checkinput ( firstset, followset ) ;
begin
  if not ( token in firstset ) then
    error ;
    scanto ( firstset ∪ followset ) ;
  end if ;
end;
```

27

4.5 Bottom-up parsing

Bottom-up parsing: intro

"R" stands for *right-most* derivation.

LR(0)
- only for very simple grammars
- approx. 300 states for standard programming languages
- only as warm-up for SLR(1) and LALR(1)

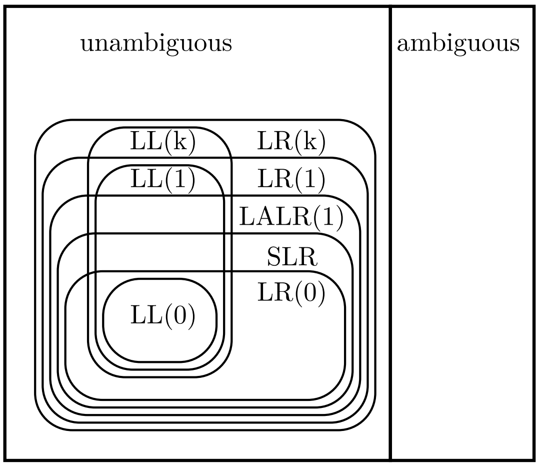
SLR(1)
- expressive enough for most grammars for standard PLs
- same number of states as LR(0)
- main focus here

LALR(1)
- slightly more expressive than SLR(1)
- same number of states as LR(0)
- we look at ideas behind that method as well

LR(1) covers all grammars, which can in principle be parsed by looking at the next token

There might seem to be a contradiction in the explanation of LR(0): if LR(0) is so weak that it works only for unreasonably simple languages, how can one speak about that standard languages have 300 states? The answer is, the other more expressive parsers (SLR(1) and LALR(1)) use the *same* construction of states, so that's why one can estimate the number of states, even if standard languages don't have an LR(0) parser; they may have an LALR(1)-parser, which has, it its core, LR(0)-states.

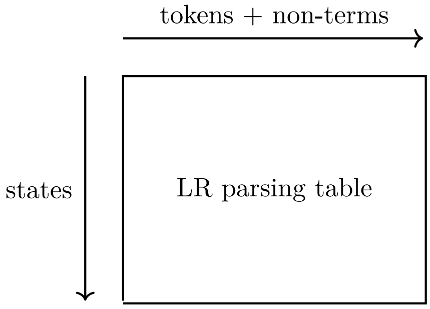
Grammar classes overview (again)



LR-parsing and its subclasses

- *right-most* derivation (but left-to-right parsing)
- in general: bottom-up: more powerful than top-down
- typically: tool-supported (unlike recursive descent, which may well be hand-coded)
- based on *parsing tables* + explicit *stack*
- thankfully: *left-recursion* no longer problematic
- typical tools: yacc and friends (like bison, CUP, etc.)

- another name: *shift-reduce* parser



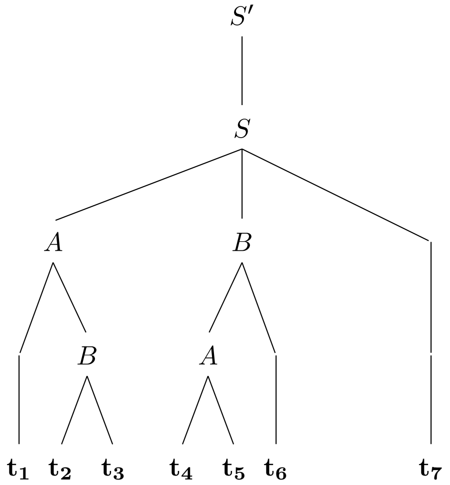
Example grammar

$$\begin{array}{rcl} S' & \to & S \\ S & \to & AB\mathbf{t_7} \mid \ldots \\ A & \to & \mathbf{t_4t_5} \mid \mathbf{t_1}B \mid \ldots \\ B & \to & \mathbf{t_2t_3} \mid A\mathbf{t_6} \mid \ldots \end{array}$$

- assume: grammar unambiguous
- assume word of terminals $\mathbf{t_1t_2}\ldots\mathbf{t_7}$ and its (unique) parse-tree

- general agreement for bottom-up parsing:
  - start symbol *never* on the right-hand side or a production
  - **routinely add another "extra" start-symbol** (here S')[11]

Parse tree for $\mathbf{t_1}\ldots\mathbf{t_7}$



Remember: parse tree independent from left- or right-most-derivation

[11] That will later be relied upon when constructing a DFA for "scanning" the stack, to control the reactions of the stack machine. This restriction leads to a unique, well-defined initial state.

LR: left-to right scan, right-most derivation?

Potentially puzzling question at first sight:

what?: *right*-most derivation, when parsing *left*-to-right?

- short answer: parser builds the parse tree **bottom-up**
- derivation:
  - replacement of nonterminals by right-hand sides
  - *derivation*: builds (implicitly) a parse-tree *top-down*

- sentential form: word from Σ^* derivable from start-symbol

Right-sentential form: right-most derivation

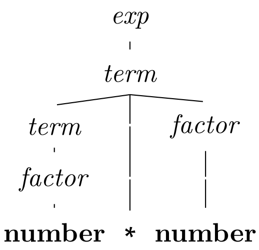
$$S \Rightarrow_r^* \alpha$$

Slighly longer answer

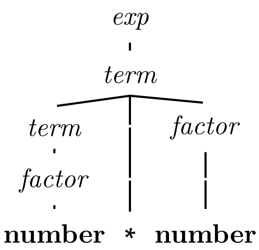
LR parser parses from left-to-right and builds the parse tree bottom-up. When doing the parse, the parser (implicitly) builds a *right-most* derivation **in reverse** (because of bottom-up).

Example expression grammar (from before)

$$\begin{array}{rcl} \mathit{exp} & \rightarrow & \mathit{exp}\ \mathit{addop}\ \mathit{term} \mid \mathit{term} \\ \mathit{addop} & \rightarrow & \mathtt{+} \mid \mathtt{-} \\ \mathit{term} & \rightarrow & \mathit{term}\ \mathit{mulop}\ \mathit{factor} \mid \mathit{factor} \\ \mathit{mulop} & \rightarrow & \mathtt{*} \\ \mathit{factor} & \rightarrow & \mathtt{(}\ \mathit{exp}\ \mathtt{)} \mid \mathbf{number} \end{array} \tag{4.8}$$



Bottom-up parse: Growing the parse tree



$$\begin{array}{rcl}
\underline{\textbf{number}} * \textbf{number} & \hookrightarrow & \underline{\mathit{factor}} * \textbf{number} \\
 & \hookrightarrow & \mathit{term} * \underline{\textbf{number}} \\
 & \hookrightarrow & \underline{\mathit{term} * \mathit{factor}} \\
 & \hookrightarrow & \underline{\mathit{term}} \\
 & \hookrightarrow & \mathit{exp}
\end{array}$$

Reduction in reverse = right derivation

Reduction

$$\begin{array}{rcl}
\underline{\mathbf{n}} * \mathbf{n} & \hookrightarrow & \underline{\mathit{factor}} * \mathbf{n} \\
 & \hookrightarrow & \mathit{term} * \underline{\mathbf{n}} \\
 & \hookrightarrow & \underline{\mathit{term} * \mathit{factor}} \\
 & \hookrightarrow & \underline{\mathit{term}} \\
 & \hookrightarrow & \mathit{exp}
\end{array}$$

Right derivation

$$\begin{array}{rcl}
\mathbf{n} * \mathbf{n} & \Leftarrow_r & \underline{\mathit{factor}} * \mathbf{n} \\
 & \Leftarrow_r & \underline{\mathit{term}} * \mathbf{n} \\
 & \Leftarrow_r & \mathit{term} * \underline{\mathit{factor}} \\
 & \Leftarrow_r & \underline{\mathit{term}} \\
 & \Leftarrow_r & \underline{\mathit{exp}}
\end{array}$$

- underlined part:
  - *different* in reduction vs. derivation
  - represents the "part being replaced"
    - for derivation: right-most non-terminal
    - for reduction: indicates the so-called **handle** (or part of it)
- consequently: all intermediate words are *right-sentential forms*

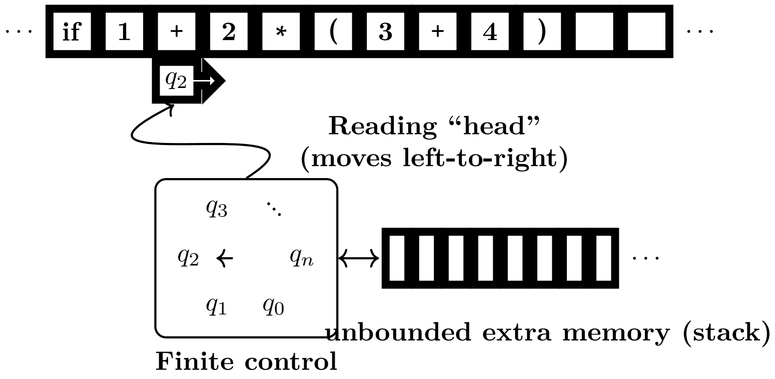
Handle

Definition 4.5.1 (Handle)**.** Assume $S \Rightarrow_r^* \alpha A w \Rightarrow_r \alpha\beta w$. A production $A \to \beta$ at position k following α is a *handle of* $\alpha\beta w$. We write $\langle A \to \beta, k\rangle$ for such a handle.

Note:

- w (right of a handle) contains only terminals
- w: corresponds to the future input still to be parsed!
- $\alpha\beta$ will correspond to the stack content (β the part touched by reduction step).
- the $\Rightarrow_r$ -derivation-step *in reverse*:
  - one **reduce**-step in the LR-parser-machine
  - adding (implicitly in the LR-machine) a new parent to children β (= **bottom-up**!)
- "handle"-part β can be *empty* (= $\boldsymbol{\epsilon}$)

Schematic picture of parser machine (again)



General LR "parser machine" configuration

- *stack*:
  - contains: terminals + non-terminals (+ **$**)
  - containing: what has been read already but not yet "processed"
- *position* on the "tape" (= token stream)
  - represented here as word of terminals *not yet read*
  - end of "rest of token stream": **$**, as usual
- *state* of the machine
  - in the following schematic illustrations: *not* yet part of the discussion
  - *later*: part of the parser table, currently we explain *without* referring to the state of the parser-engine
  - currently we assume: tree and rest of the input given
  - the trick ultimately will be: how do achieve the same *without that tree already given* (just parsing left-to-right)

Schematic run (reduction: from top to bottom)

$\$$	$\mathbf{t}_1\mathbf{t}_2\mathbf{t}_3\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\$$
$\$\,\mathbf{t}_1$	$\mathbf{t}_2\mathbf{t}_3\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\$$
$\$\,\mathbf{t}_1\mathbf{t}_2$	$\mathbf{t}_3\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\$$
$\$\,\mathbf{t}_1\mathbf{t}_2\mathbf{t}_3$	$\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\$$
$\$\,\mathbf{t}_1 B$	$\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\$$
$\$\,A$	$\mathbf{t}_4\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\$$
$\$\,A\mathbf{t}_4$	$\mathbf{t}_5\mathbf{t}_6\mathbf{t}_7\,\$$
$\$\,A\mathbf{t}_4\mathbf{t}_5$	$\mathbf{t}_6\mathbf{t}_7\,\$$
$\$\,AA$	$\mathbf{t}_6\mathbf{t}_7\,\$$
$\$\,AA\mathbf{t}_6$	$\mathbf{t}_7\,\$$
$\$\,AB$	$\mathbf{t}_7\,\$$
$\$\,AB\mathbf{t}_7$	$\$$
$\$\,S$	$\$$
$\$\,S'$	$\$$

2 basic steps: shift and reduce

- parsers reads input and uses stack as intermediate storage
- so far: no mention of look-ahead (i.e., action depending on the value of the next token(s)), but that may play a role, as well

Shift

Move the next input symbol (terminal) over to the top of the stack ("push")

Reduce

Remove the symbols of the *right-most* subtree from the stack and replace it by the non-terminal at the root of the subtree (replace = "pop + push").

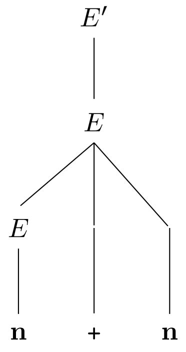
- decision *easy* to do **if one has the parse tree already**!
- *reduce* step: popped resp. pushed part = right- resp. left-hand side of handle

The remark that it's "easy to do" refers to something that is illustrated next: the question namely the decision-making process of the parser. should the parser do a shift or a reduce and if so, reduce with what rule. If one assumes the "target" parse-tree as already given (as we currently do in our presentation, for instance also in the following slides), then tree embodies those decisions. Ultimately, of course, the tree is *not* given a priori, it's the parser's task to build the tree (at least implicitly) by making those decisions about what the next step is (shift or reduce).

Example: LR parse for "+" (given the tree)

$$\begin{array}{rcl} E' & \to & E \\ E & \to & E + \mathbf{n} \mid \mathbf{n} \end{array}$$

CST



Run

	parse stack	input	action
1	$\$$	$\mathbf{n}+\mathbf{n}\,\$$	shift
2	$\$\,\mathbf{n}$	$+\mathbf{n}\,\$$	red:. $E \rightarrow \mathbf{n}$
3	$\$\,E$	$+\mathbf{n}\,\$$	shift
4	$\$\,E+$	$\mathbf{n}\,\$$	shift
5	$\$\,E+\mathbf{n}$	$\$$	reduce $E \rightarrow E+\mathbf{n}$
6	$\$\,E$	$\$$	red.: $E' \rightarrow E$
7	$\$\,E'$	$\$$	accept

note: line 3 vs line 6!; both contain E on top of stack

(right) derivation: reduce-steps "in reverse"

$$\underline{E'} \Rightarrow \underline{E} \Rightarrow \underline{E}+\mathbf{n} \Rightarrow \mathbf{n}+\mathbf{n}$$

The example is supposed to shed light on how the machine can make decisions assuming that the tree is already given. For that, one should compare the situation in stage 3 and state 6. In both situations, the machine has *the same stack content* (containing only the end-marker and E on top of the stack). However, at stage 3, the machine does a shift, whereas in stage 6, it does a reduce. Since the stack content (representing the "past" of the parse, i.e., the already processed input) is the identical in both cases, the parser machine is necessarily *in the same state* in both stages, which mean, it cannot be the state that makes the difference. What then? In the example, the form of the parse tree shows what the parser should do. But of course the tree is not available. Instead (and not surprisingly). If the past input cannot be used to make the distinction, one takes the "future" input. Maybe not all of it (as that would correspond to the tree), but part of it. That's a form of a look-ahead (that will *not* yet be done for LR(0), as that for is without look-ahead).

Example with ϵ-transitions: parentheses

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & (\,S\,)\,S \mid \epsilon \end{array}$$

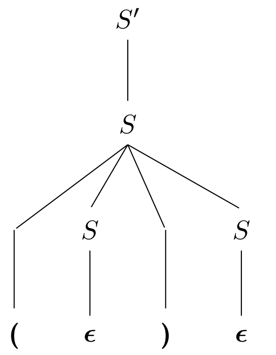
side remark: unlike previous grammar, here:

- production with *two* non-terminals on the right

⇒ difference between left-most and right-most derivations (and mixed ones)

Parentheses: run and right-most derivation

CST



Run

	parse stack	input	action
1	\$	( ) \$	shift
2	\$ (	) \$	reduce $S \to \epsilon$
3	\$ (S	) \$	shift
4	\$ (S)	\$	reduce $S \to \epsilon$
5	\$ (S) S	\$	reduce $S \to (S)S$
6	\$ S	\$	reduce $S' \to S$
7	\$ S'	\$	accept

Note: the 2 reduction steps for the ϵ productions

Right-most derivation and right-sentential forms

$$\underline{S'} \Rightarrow_r \underline{S} \Rightarrow_r (S)\underline{S} \Rightarrow_r (\underline{S}) \Rightarrow_r (\,)$$

Right-sentential forms & the stack

- sentential form: word from Σ^* derivable from start-symbol

Right-sentential form: right-most derivation

$$S \Rightarrow_r^* \alpha$$

- right-sentential forms:
  - part of the "run"
  - but: **split** between *stack* and *input*

	parse stack	input	action
1	$\$$	$\mathbf{n}+\mathbf{n}\,\$$	shift
2	$\$\,\mathbf{n}$	$+\mathbf{n}\,\$$	red:. $E \to \mathbf{n}$
3	$\$\,E$	$+\mathbf{n}\,\$$	shift
4	$\$\,E+$	$\mathbf{n}\,\$$	shift
5	$\$\,E+\mathbf{n}$	$\$$	reduce $E \to E+\mathbf{n}$
6	$\$\,E$	$\$$	red.: $E' \to E$
7	$\$\,E'$	$\$$	accept

$$\underline{E'} \Rightarrow_r \underline{E} \Rightarrow_r \underline{E}+\mathbf{n} \Rightarrow_r \mathbf{n}+\mathbf{n}$$

$$\underline{\mathbf{n}}+\mathbf{n} \hookrightarrow \underline{E+\mathbf{n}} \hookrightarrow \underline{E} \hookrightarrow E'$$

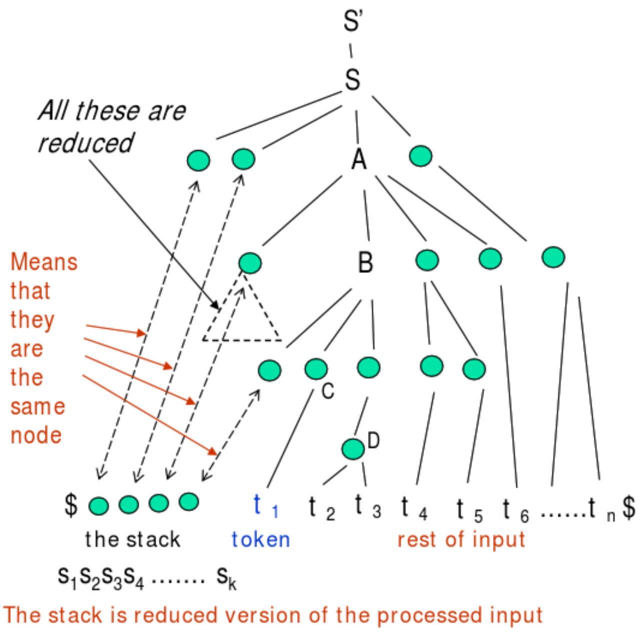
$$\underline{E'} \Rightarrow_r \underline{E} \Rightarrow_r \underline{E}+\mathbf{n} \parallel \sim\ \underline{E}+ \parallel \mathbf{n} \sim \underline{E} \parallel +\mathbf{n} \Rightarrow_r \mathbf{n} \parallel\ +\mathbf{n} \sim\parallel \mathbf{n}+\mathbf{n}$$

The $\parallel$ here is introduced as "ad-hoc" notation to illustrate the separation between the parse stack on the left and the future input on the right.

Viable prefixes of right-sentential forms and handles

- right-sentential form: $E+\mathbf{n}$
- **viable prefixes** of RSF
  - prefixes of that RSF *on the stack*
  - here: 3 viable prefixes of that RSF: E, $E+$, $E+\mathbf{n}$
- *handle*: remember the definition earlier
- here: for instance in the sentential form $\mathbf{n}+\mathbf{n}$
  - handle is production $E \to \mathbf{n}$ on the *left* occurrence of $\mathbf{n}$ in $\mathbf{n}+\mathbf{n}$ (let's write $\mathbf{n}_1+\mathbf{n}_2$ for now)
  - note: in the stack machine:
    - ∗ the left $\mathbf{n}_1$ on the stack
    - ∗ rest $+\mathbf{n}_2$ on the input (unread, because of LR(0))
- if the parser engine detects handle $\mathbf{n}_1$ on the stack, it does a *reduce*-step
- However (later): reaction depends on current *state* of the parser engine

A typical situation during LR-parsing



After a shift, the next reduction to be made is a reduction with the production:

C -> t1

Then, after two shifts, we will make a reduction with the production:

D -> t2 t3

Then, what's next?

General design for an LR-engine

- some ingredients clarified up-to now:
  - bottom-up tree building as reverse right-most derivation,
  - stack vs. input,
  - shift and reduce steps
- however: 1 ingredient missing: next step of the engine may depend on
  - top of the stack ("handle")
  - look ahead on the input (but not for LL(0))
  - and: current **state** of the machine (same stack-content, but different reactions at different stages of the parse)

But what are the states of an LR-parser?

General idea:

Construct an NFA (and ultimately DFA) which works on the **stack** (not the input). The alphabet consists of terminals and non-terminals $\Sigma_T \cup \Sigma_N$. The language

$$\mathit{Stacks}(G) = \{\alpha \;\left|\; \begin{array}{l} \alpha \text{ may occur on the stack during} \\ \text{LR-parsing of a sentence in } \mathcal{L}(G) \end{array}\right.\}$$

is **regular**!

LR(0) parsing as easy pre-stage

- LR(0): in practice *too simple*, but easy conceptual step towards LR(1), SLR(1) etc.
- LR(1): in practice good enough, LR(k) not used for $k > 1$

LR(0) item

production with specific "parser position" **.** in its right-hand side

- **.** : "meta-symbol" (not part of the production)

LR(0) item for a production $A \to \beta\gamma$

$$A \to \beta.\gamma$$

- item with dot at the beginning: *initial* item
- item with dot at the end: *complete* item

LR(0) parsing is introduced as easy pre-stage for the more expressive forms of bottom-up parsing later. In itself, it's not expressive enough to be practivally useful. But the construction underlies directly or at least conceptually the more complex parser constructions to come. In particular: for LR(0) parsing, the core of the construction is the so-called LR(0)-DFA, based on LR(0)-items. This construction is directly also used for SLR-parsing. For LR(1) and LALR(1), the construction of the corresponding DFA is not identical, but analogous to the construction of LR(0)-DFA.

Example: items of LR-grammar

Grammar for parentheses: 3 productions

$$\begin{array}{rcl} S' & \to & S \\ S & \to & (\, S\,)\, S \mid \epsilon \end{array}$$

8 items

$$\begin{array}{rcl} S' & \to & .S \\ S' & \to & S. \\ S & \to & .(\, S\,)\, S \\ S & \to & (\, .S\,)\, S \\ S & \to & (\, S.\,)\, S \\ S & \to & (\, S\,).S \\ S & \to & (\, S\,)\, S. \\ S & \to & . \end{array}$$

- $S \to \epsilon$ gives $S \to .$ as item (not $S \to \epsilon.$ and $S \to .\epsilon$)
- side remark for later: it will turn out: grammar is *not LR(0)*

Another example: items for addition grammar

Grammar for addition: 3 productions

$$\begin{array}{rcl} E' & \to & E \\ E & \to & E + \mathbf{n} \mid \mathbf{n} \end{array}$$

(coincidentally also:) 8 items

$$\begin{array}{rcl} E' & \to & .E \\ E' & \to & E. \\ E & \to & .E + \mathbf{n} \\ E & \to & E. + \mathbf{n} \\ E & \to & E + .\mathbf{n} \\ E & \to & E + \mathbf{n}. \\ E & \to & .\mathbf{n} \\ E & \to & \mathbf{n}. \end{array}$$

- also here, it will turn out: *not an LR(0)* grammar

Finite automata of items

- general set-up: *items* as **states in an automaton**
- automaton: “operates” *not* on the input, **but the stack**
- automaton either
  - first NFA, afterwards made deterministic (subset construction), or
  - directly DFA

States formed of sets of items

In a state marked by/containing item

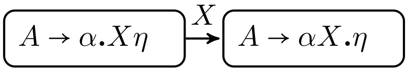
$$A \to \beta.\gamma$$

- β on the *stack*
- γ: to be treated next (terminals on the input, but can contain also non-terminals)

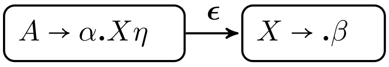
State transitions of the NFA

- $X \in \Sigma$
- two kinds of transitions

Terminal or non-terminal



Epsilon (X: non-terminal here)



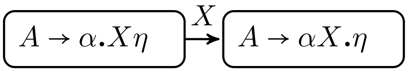
- In case X = *terminal* (i.e. token) =
  - the left step corresponds to a **shift** step[12]
- for non-terminals (see next slide):
  - interpretation more complex: non-terminals are officially never on the input
  - note: in that case, item $A \to \alpha.X\eta$ has two (kinds of) outgoing transitions

Transitions for non-terminals and ϵ

- so far: we never pushed a non-terminal from the input to the stack, we **replace** in a **reduce**-step the right-hand side by a left-hand side
- but: replacement in a **reduce** steps can be seen as
  1. pop right-hand side off the stack,
  2. instead, "assume" corresponding non-terminal on input,
  3. eat the non-terminal an push it on the stack.
- two kinds of transitions
- assume production $X \to \beta$ and *initial* item $X \to .\beta$

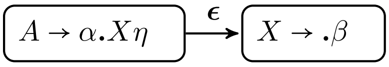
Transitions

Terminal or non-terminal



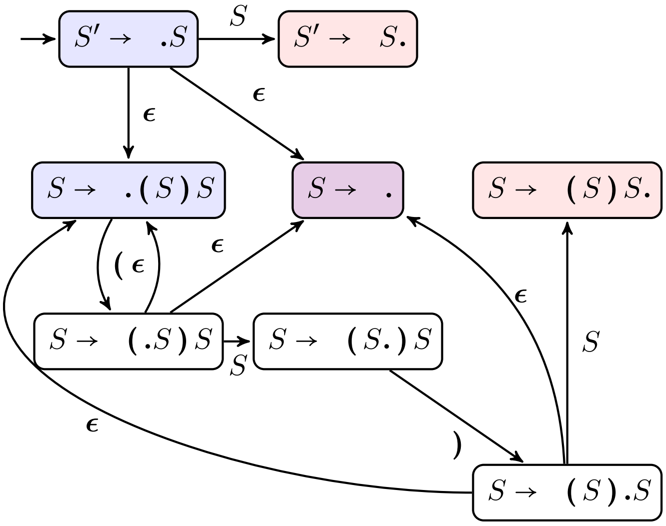
Epsilon (X: non-terminal here)

Given production $X \to \beta$:



[12] We have explained *shift* steps so far as: parser eats one *terminal* (= input token) and pushes it on the stack.

NFA: parentheses



In the figure, we us colors are used for illustration, only, i.e., they are not officially part of the construction. The colors are intended to represent the following:

- "reddish": complete items
- "blueish": init-item (less important)
- "violet'tish": both.

Furthermore, you may notice for the *initial items*:

- one per production of the grammar
- that's where the ϵ-transisitions go into, but
- *with exception* of the initial state (with S'-production): no outgoing edges from the complete items.

Note the uniformity of the ϵ-transitions in the following sense. For each production with a given non-terminal (for instance S in the given example), there is one ingoing ϵ-transition from each state/item where the **.** is in front of said non-terminal.

To look forward, and the role of the ϵ-transitions. Those are allowed for *non-determistic* automata, but not for DFAs. The underlying construction (discussed later) is building the ϵ-closure, in this case the close of $A' \to A$. If one does that directly, one obtains directly a DFA (as opposed to first do an NFA to make deterministic in a second phase).

Initial and final states

initial states:

- we make our lives *easier*: assume one *extra* start symbol say S' (augmented grammar)

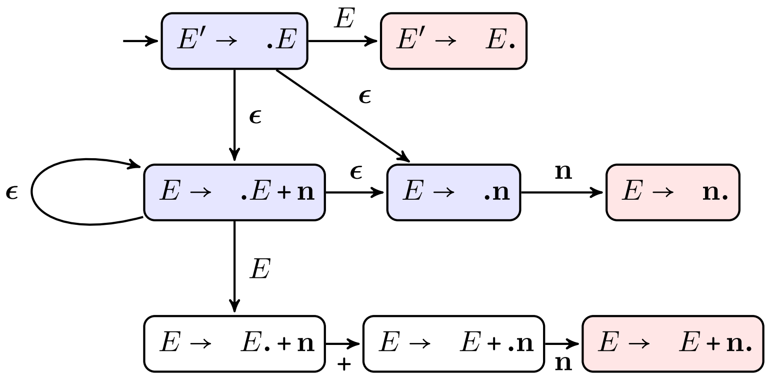
⇒ initial item $S' \to .S$ as (only) **initial state**

final states:

- acceptance condition of the *overall* machine: a bit more complex
  - input must be empty
  - stack must be empty except the (new) start symbol
  - NFA has a word to say about acceptence
    - but *not* in form of being in an accepting state
    - so: no accepting *states*
    - but: accepting *action* (see later)

The NFA (or later DFA) has a specific task, it is used to "scan" the *stack* (at least conceptually), not the input. The automaton is not so much for *accepting* a stack and then stop, it's more like determining the state that corresponds to the current stack content. Therefore there are no accepting states in the sense of a FSA!

NFA: addition



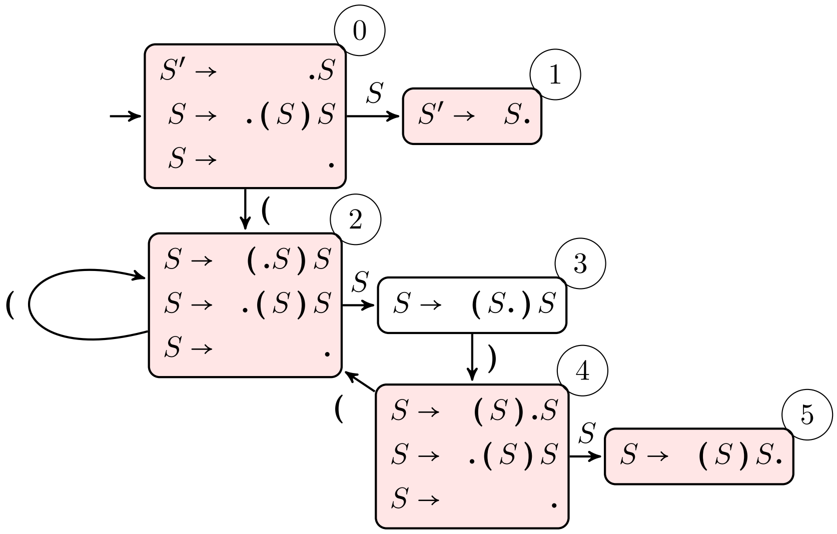
Determinizing: from NFA to DFA

- standard subset-construction[13]
- states then contain *sets* of items
- important: ϵ-closure
- also: *direct* construction of the DFA possible

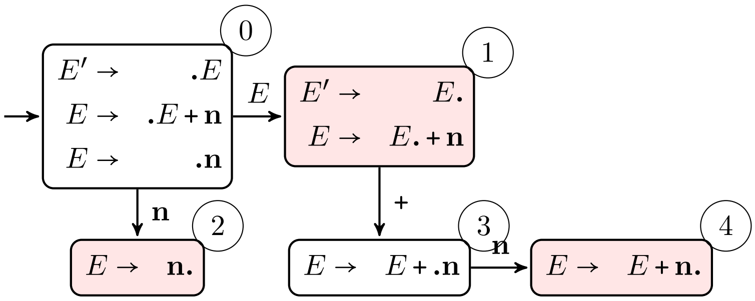
In the following two slides, we should the DFAs corresponding to the NFAs shown before. For the construction on how to determinize NFAs (and minimize them), we refer to the corresponding sections in the chapter about lexing. Anyway, we will afterwards also look at a *direct* construction of the DFA (without the detour over NFAs). That will result in the same automata anyway.

[13] Technically, we don't require here a *total* transition function, we leave out any error state.

DFA: parentheses



DFA: addition



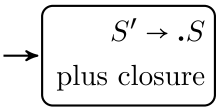
Direct construction of an LR(0)-DFA

- quite easy: simply build in the closure already

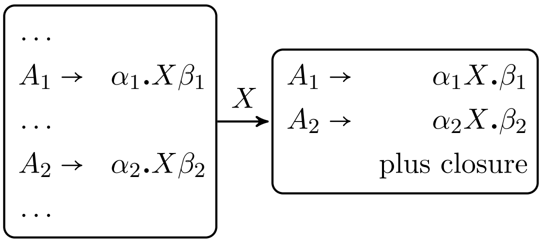
ϵ-closure

- if $A \to \alpha.B\gamma$ is an item in a state where
- there are productions $B \to \beta_1 \mid \beta_2 \ldots$ then
- add items $B \to .\beta_1$, $B \to .\beta_2$ ... to the state
- continue that process, until saturation

initial state



Direct DFA construction: transitions


...
$A_1 \to \alpha_1.X\beta_1$
...
$A_2 \to \alpha_2.X\beta_2$
...
X
$A_1 \to \alpha_1 X.\beta_1$
$A_2 \to \alpha_2 X.\beta_2$
plus closure

- X: terminal or non-terminal, both treated uniformely
- *All* items of the form $A \to \alpha.X\beta$ must be included in the post-state
- and all others (indicated by "...") in the pre-state: not included
- re-check the previous examples: outcome is the same

How does the DFA do the shift/reduce and the rest?

- we have seen: bottom-up parse tree generation
- we have seen: shift-reduce and the stack vs. input
- we have seen: the construction of the DFA

But: how does it hang together?

We need to interpret the "set-of-item-states" in the light of the stack content and figure out the **reaction** in terms of

- transitions in the automaton
- stack manipulations (shift/reduce)
- acceptance
- input (apart from shifting) not relevant when doing LR(0)

and the reaction better be uniquely determined ....

Stack contents and state of the automaton

- remember: at any config. of stack/input in a run
  1. stack contains words from Σ^*
  2. DFA operates deterministically on such words
- the stack contains "abstraction of the past":
- when feeding that "past" on the stack into the automaton
  - starting with the oldest symbol (not in a LIFO manner)
  - starting with the DFA's initial state
  - ⇒ stack content **determines** state of the DFA
- actually: each prefix also determines uniquely a state
- **top state**:
  - state after the complete stack content
  - corresponds to the **current** state of the stack-machine

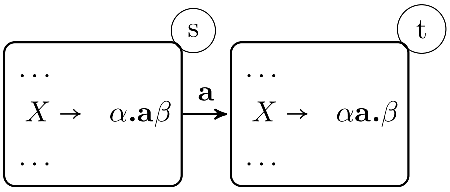
⇒ crucial when determining *reaction*

State transition allowing a shift

- assume: top-state (= current state) contains item

$$X \to \alpha.\mathbf{a}\beta$$

- construction thus has transition as follows


s
t
...
$X \to \quad \alpha.\mathbf{a}\beta$
...
a
...
$X \to \quad \alpha\mathbf{a}.\beta$
...

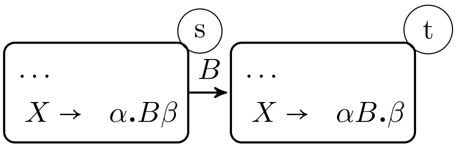
- shift is possible
- if shift is *the* correct operation and **a** is terminal symbol corresponding to the current token: state afterwards = t

State transition: analogous for non-terminals

Production

$$X \to \alpha.B\beta$$

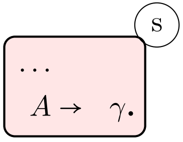
Transition



- same as before, now with non-terminal B
- note: we never read non-term from input
- not officially called a shift
- corresponds to the reaction **followed by** a *reduce* step, it's **not** the reduce step itself
- think of it as follows: reduce and subsequent step
  - not as: *replace* on top of the stack the handle (right-hand side) by non-term B,
  - but instead as:
    1. pop off the handle from the top of the stack
    2. put the non-term B "back onto the input" (corresponding to the above state s)
    3. eat the B and *shift* it to the stack
- later: a **goto** reaction in the parse table

State (not transition) where a reduce is possible

- remember: *complete items*
- assume **top state** s containing complete item $A \to \gamma.$



- a complete right-hand side ("handle") γ on the stack and thus done
- may be replaced by right-hand side A

⇒ reduce step

- builds up (implicitly) new parent node A in the bottom-up procedure
- **Note**: A on top of the stack instead of γ:
  - **new top state**!
  - remember the "goto-transition" (shift of a non-terminal)

A conceptual picture for the reduce step is as follows. As said, we remove the handle from the stack, and "pretend", as if the A is next on the input, and thus we "shift" it on top of the stack, doing the corresponding A-transition.

Remarks: states, transitions, and reduce steps

- ignoring the ϵ-transitions (for the NFA)
- there are 2 "kinds" of transitions in the DFA
  1. terminals: reals shifts
  2. non-terminals: "following a reduce step"

No edges to represent (all of) a reduce step!

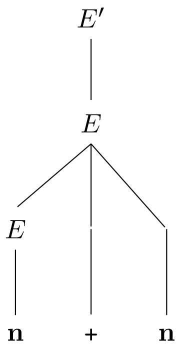
- if a reduce happens, parser engine *changes state*!
- however: this state change is **not** represented by a transition in the DFA (or NFA for that matter)
- especially *not* by outgoing errors of completed items

- if the (rhs of the) handle is *removed* from top stack ⇒
  - "go back to the (top) state before that handle had been added": *no edge for that*
- later: stack notation simply remembers the state as part of its configuration

Example: LR parsing for addition (given the tree)

$$\begin{array}{rcl} E' & \to & E \\ E & \to & E + \mathbf{n} \mid \mathbf{n} \end{array}$$

CST



Run

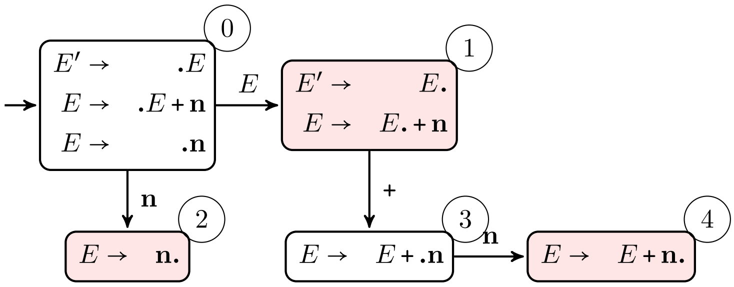
	parse stack	input	action
1	$\$$	$\mathbf{n}+\mathbf{n}\,\$$	shift
2	$\$\,\mathbf{n}$	$+\mathbf{n}\,\$$	red:. $E \to \mathbf{n}$
3	$\$\,E$	$+\mathbf{n}\,\$$	shift
4	$\$\,E+$	$\mathbf{n}\,\$$	shift
5	$\$\,E+\mathbf{n}$	$\$$	reduce $E \to E+\mathbf{n}$
6	$\$\,E$	$\$$	red.: $E' \to E$
7	$\$\,E'$	$\$$	accept

note: line 3 vs line 6!; both contain E on top of stack

This is a revisit of an example resp. slide from earlier, when we discussed how a parser can do decisions, resp. that it would be easy to do decisions for the parser machine if it had the tree already. Unfortunately it has the tree not available, the only thing it has is “the past” which is represented (partially) by the stack content. As discussed earlier, interesting in the run are stage 3 and state 6, which have the same stack content, which also means, the parser is in the same state of its LR(0)-DFA. With the automaton constructed as before, that’s state 1. The state 1 is important, as it illustrates a shift/reduce conflict. Remember: reduce-steps are *not* represented in the LR(0)-automaton via *transitions*. They are only implicitly represented by *complete items*. Thus, as shift-reduce conflict is not characterized by 2 outgoing edges. It’s one **outgoing edge from a state containing a complete item**.

Earlier we hinted at that an automaton could make decisions based on a look-head. That is not yet done: the LR(0), in state 1 especially, can do a reduce step or a shift step, which constitutes the conflict. Later, we will see under which circumstances, looking at the “next symbol” can help to make the decision. That leads to SLR parsing (or even later to LR(1)/LALR(1)). In the particular situation of state 1 in the example, the next possible symbol would be **+** or else **$**

DFA of addition example



- note line 3 vs. line 6
- both stacks $= E \Rightarrow$ same (top) state in the DFA (state 1)

The point being made when lookig at that 1 is the following: the state is a complete state (a state containing a complete item). Besides that, there is an outgoing edge. That means, in that state, there are two reactions possible: a shift (following the edge) and a reduce, as indicated by the complete item. That indicates a conflict-situation, especially if we don't make use of look-aheads, as we do currently, when discussing LR(0). The conflict-situation is called, expectely a "shift-reduce-conflict", more precisely an LR(0)-shift/reduce conflict. The qualification LR(0) is necessary, as sometimes, a more close look at the situation and taking a look-ahead into account may defuse the conflict. Those more fine-grainend considerations will lead to extensions of the plain LR(0)-parsing (like SLR(0), or LR(1) and LALR(1)).

LR(0) grammars

LR(0) grammar

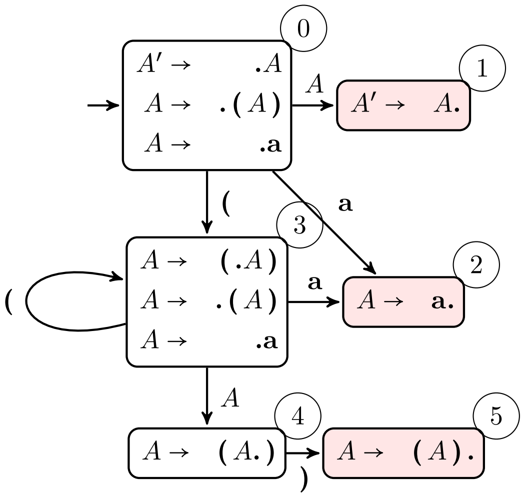
The top-state alone determines the next step.

- especially: no shift/reduce conflicts in the form shown
- thus: previous addition-grammar is *not LR(0)*

Simple parentheses

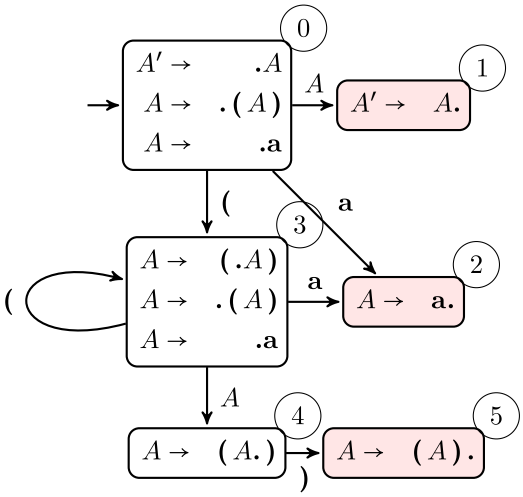
$$A \to (\, A\,) \mid \mathbf{a}$$

DFA



Simple parentheses is LR(0)

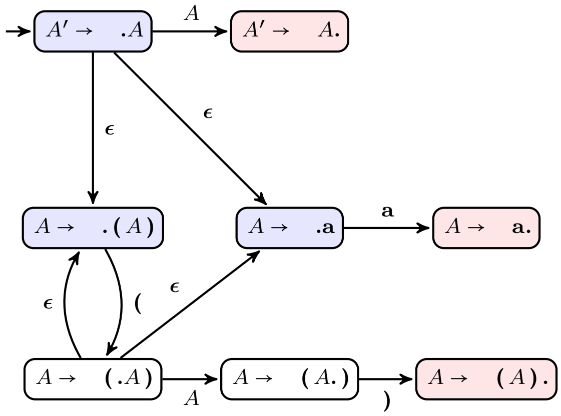
DFA



Remarks

state	possible action
0	only shift
1	only red: $(A' \to A)$
2	only red: $(A \to \mathbf{a})$
3	only shift
4	only shift
5	only red $(A \to (A))$

NFA for simple parentheses (bonus slide)



For completeness sake: that's the NFA for the "simple parentheses".

Parsing table for an LR(0) grammar

- table structure: slightly different for SLR(1), LALR(1), and LR(1) (see later)
- note: the "goto" part: "shift" on non-terminals (only 1 non-terminal A here)
- corresponding to the A-labelled transitions

state	action	rule	input			goto
			(	**a**	)	A
0	shift		3	2		1
1	reduce	$A' \to A$				
2	reduce	$A \to \mathbf{a}$				
3	shift		3	2		4
4	shift				5	
5	reduce	$A \to (A)$				

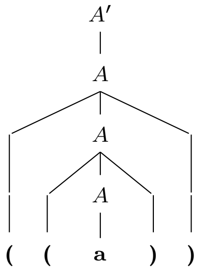
Parsing of ((a))

stage	parsing stack	input	action
1	$\$_0$	((**a**))\$	shift
2	$\$_0(_3$	(**a**))\$	shift
3	$\$_0(_3(_3$	**a**))\$	shift
4	$\$_0(_3(_3\mathbf{a}_2$	))\$	reduce $A \to \mathbf{a}$
5	$\$_0(_3(_3A_4$	))\$	shift
6	$\$_0(_3(_3A_4)_5$	)\$	reduce $A \to (A)$
7	$\$_0(_3A_4$	)\$	shift
8	$\$_0(_3A_4)_5$	\$	reduce $A \to (A)$
9	$\$_0A_1$	\$	accept

- note: stack on the left
  - contains top *state* information
  - in particular: overall **top** state on the right-most end
- note also: **accept** action
  - reduce wrt. to $A' \to A$ and
  - *empty stack* (apart from **\$**, A, and the state annotation)

  ⇒ accept

Parse tree of the parse



- As said:
  - the reduction "contains" the parse-tree
  - reduction: builds it bottom up
  - reduction in reverse: contains a *right-most* derivation (which is "top-down")
- accept action: corresponds to the parent-child edge $A' \to A$ of the tree

Parsing of erroneous input

- empty slots it the table: "errors"

stage	parsing stack	input	action
1	$\$_0$	**((a) $**	shift
2	$\$_0 (_3$	**(a) $**	shift
3	$\$_0 (_3 (_3$	**a) $**	shift
4	$\$_0 (_3 (_3 \mathbf{a}_2$	**) $**	reduce $A \to \mathbf{a}$
5	$\$_0 (_3 (_3 A_4$	**) $**	shift
6	$\$_0 (_3 (_3 A_4)_5$	**$**	reduce $A \to (\,A\,)$
7	$\$_0 (_3 A_4$	**$**	????

stage	parsing stack	input	action
1	$\$_0$	**( ) $**	shift
2	$\$_0 (_3$	**) $**	?????

Invariant

important general invariant for LR-parsing: never shift something "illegal" onto the stack

LR(0) parsing algo, given DFA

let s be the current state, on top of the parse stack

1. s contains $A \to \alpha.X\beta$, where X is a *terminal*
   - shift X from input to top of stack. The new *state* pushed on the stack: state t where $s \xrightarrow{X} t$
   - else: if s does not have such a transition: *error*
2. s contains a **complete** item (say $A \to \gamma.$): **reduce** by rule $A \to \gamma$:
   - A reduction by $S' \to S$: **accept**, if input is empty; else **error**:
   - else:

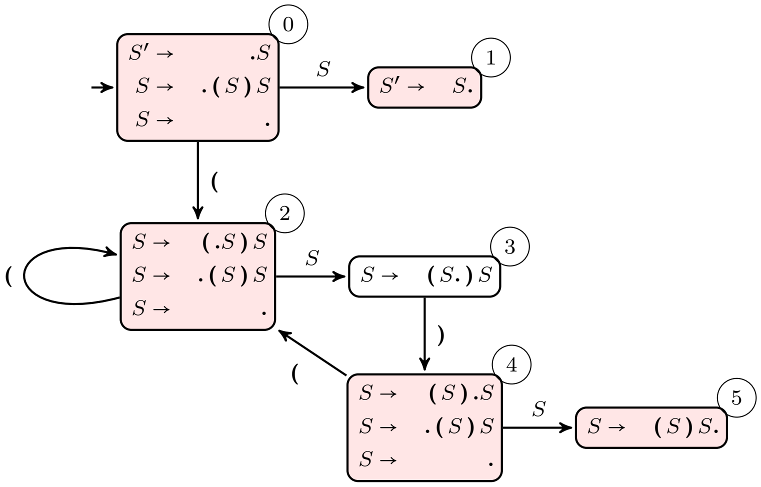
     pop: remove γ (including "its" states from the stack)

     back up: assume to be in state u which is *now* head state

     push: push A to the stack, new head state t where $u \xrightarrow{A} t$ (in the DFA)

DFA parentheses again: LR(0)?

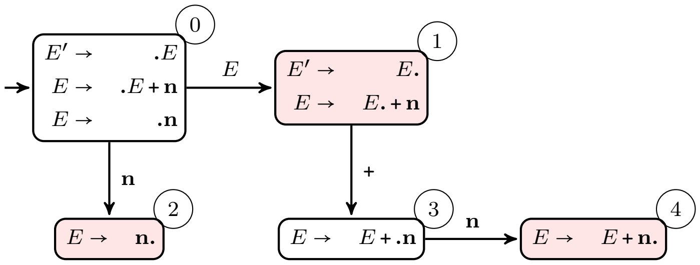
$$\begin{array}{rcl} S' & \to & S \\ S & \to & \mathbf{(}\, S\, \mathbf{)}\, S \mid \epsilon \end{array}$$



Look at states 0, 2, and 4

DFA addition again: LR(0)?

$$\begin{array}{rcl} E' & \to & E \\ E & \to & E + \mathbf{n} \mid \mathbf{n} \end{array}$$

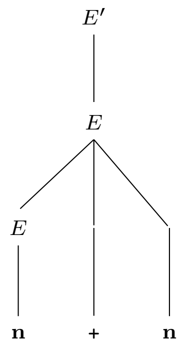


How to make a decision in state 1?

Decision? If only we knew the ultimate tree already . . .

. . . especially the parts still to come

CST



Run

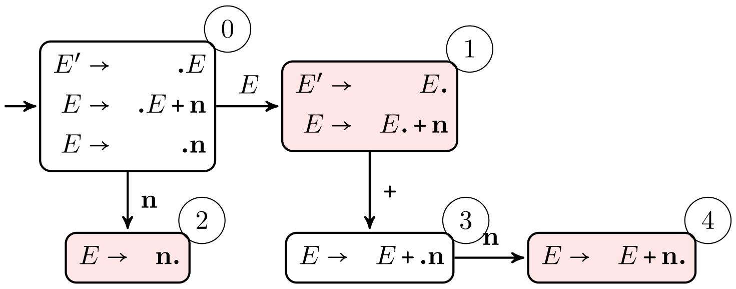
	parse stack	input	action
1	$\$$	$\mathbf{n}+\mathbf{n}\,\$$	shift
2	$\$\,\mathbf{n}$	$+\mathbf{n}\,\$$	red:. $E \to \mathbf{n}$
3	$\$\,E$	$+\mathbf{n}\,\$$	shift
4	$\$\,E+$	$\mathbf{n}\,\$$	shift
5	$\$\,E+\mathbf{n}$	$\$$	reduce $E \to E+\mathbf{n}$
6	$\$\,E$	$\$$	red.: $E' \to E$
7	$\$\,E'$	$\$$	accept

- current stack: represents already known part of the parse tree
- since we don't have the future parts of the tree yet:

⇒ **look-ahead** on the input (without building the tree yet)

- LR(1) and its variants: *look-ahead of 1* (= look at the current type of the token)

Addition grammar (again)



- *How to make a decision in state* 1*?* (here: shift vs. reduce)

⇒ look at the next input symbol (in the token)

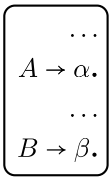
One look-ahead

- LR(0), not useful, too weak
- add look-ahead, here of *1 input symbol* (= token)

- different variations of that idea (with slight difference in expresiveness)
- tables slightly changed (compared to LR(0))
- but: *still* can use the LR(0)-DFAs

Resolving LR(0) reduce/reduce conflicts

LR(0) reduce/reduce conflict:



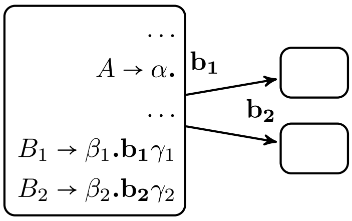
SLR(1) solution: use follow sets of non-terms

- If $\mathit{Follow}(A) \cap \mathit{Follow}(B) = \emptyset$

⇒ next symbol (in `token`) decides!
  - if `token` $\in \mathit{Follow}(\alpha)$ then reduce using $A \to \alpha$
  - if `token` $\in \mathit{Follow}(\beta)$ then reduce using $B \to \beta$
  - ...

Resolving LR(0) shift/reduce conflicts

LR(0) shift/reduce conflict:


...
$A \to \alpha.$
...
$B_1 \to \beta_1.\mathbf{b_1}\gamma_1$
$B_2 \to \beta_2.\mathbf{b_2}\gamma_2$
$\mathbf{b_1}$
$\mathbf{b_2}$

SLR(1) solution: again: use follow sets of non-terms

- If $\mathit{Follow}(A) \cap \{\mathbf{b_1}, \mathbf{b_2}, \ldots\} = \emptyset$

⇒ next symbol (in `token`) decides!
  - if `token` $\in \mathit{Follow}(A)$ then *reduce* using $A \to \alpha$, non-terminal A determines new top state
  - if `token` $\in \{\mathbf{b_1}, \mathbf{b_2}, \ldots\}$ then *shift*. Input symbol $\mathbf{b_i}$ determines new top state
  - ...

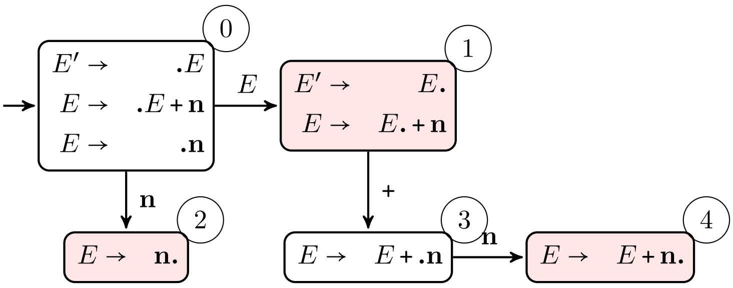
SLR(1) requirement on states (as in the book)

- formulated as conditions on the states (of LR(0)-items)
- given the LR(0)-item DFA as defined

SLR(1) condition, on all states s

1. For any item $A \to \alpha.X\beta$ in s with X a *terminal*, there is no **complete** item $B \to \gamma.$ in s with $X \in \mathit{Follow}(B)$.
2. For any **two complete** items $A \to \alpha.$ and $B \to \beta.$ in s, $\mathit{Follow}(\alpha) \cap \mathit{Follow}(\beta) = \varnothing$

Revisit addition one more time



- $\mathit{Follow}(E') = \{\$\}$

⇒
  - shift for +
  - reduce with $E' \to E$ for **$** (which corresponds to accept, in case the input is empty)

SLR(1) algo

let s be the current state, on top of the parse stack

1. s contains $A \to \alpha.X\beta$, where X is a terminal **and X is the next token on the input**, then
   - shift X from input to top of stack. The new *state* pushed on the stack: state t where $s \xrightarrow{X} t$[14]
2. s contains a *complete* item (say $A \to \gamma.$) **and the next token in the input is in** $\mathit{Follow}(A)$: *reduce* by rule $A \to \gamma$:
   - A reduction by $S' \to S$: *accept*, if input is empty[15]
   - else:

     pop: remove γ (including "its" states from the stack)

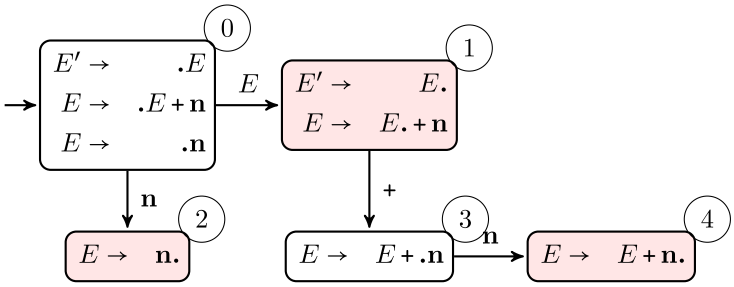
     back up: assume to be in state u which is *now* head state

     push: push A to the stack, new head state t where $u \xrightarrow{A} t$
3. if next token is such that neither 1. or 2. applies: *error*

[14] Cf. to the LR(0) algo: since we checked the existence of the transition before, the else-part is missing now.

[15] Cf. to the LR(0) algo: This happens *now* only if next token is **$**. Note that the follow set of S' in the *augmented* grammar is always only **$**

Parsing table for SLR(1)



state	input			goto
	$\mathbf{n}$	$+$	$\mathbf{\$}$	E
0	$s:2$			1
1		$s:3$	accept	
2		$r:(E \to \mathbf{n})$		
3	$s:4$			
4		$r:(E \to E + \mathbf{n})$	$r:(E \to E + \mathbf{n})$	

for state 2 and 4: $\mathbf{n} \notin \mathit{Follow}(E)$

Parsing table: remarks

- SLR(1) parsing table: rather similar-looking to the LR(0) one
- differences: reflect the differences in: LR(0)-algo vs. SLR(1)-algo
- same number of rows in the table (= same number of states in the DFA)
- only: colums "arranged" differently
  - LR(0): each state **uniformely**: either shift or else reduce (with given rule)
  - now: non-uniform, **dependent** on the input. But that does not apply to the previous example. We'll see that in the next, then.
- it should be obvious:
  - SLR(1) may resolve LR(0) conflicts
  - but: if the follow-set conditions are not met: SLR(1) *shift-shift* and/or SLR(1) *shift-reduce* conflicts
  - would result in non-unique entries in SLR(1)-table[16]

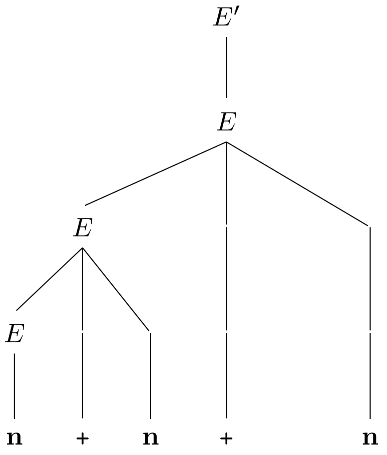
SLR(1) parser run (= "reduction")

state	input			goto
	$\mathbf{n}$	$+$	$\mathbf{\$}$	E
0	$s:2$			1
1		$s:3$	accept	
2		$r:(E \to \mathbf{n})$		
3	$s:4$			
4		$r:(E \to E + \mathbf{n})$	$r:(E \to E + \mathbf{n})$	

[16] by which it, strictly speaking, would no longer be an SLR(1)-table :-)

stage	parsing stack	input	action
1	$\$_0$	$\mathbf{n}+\mathbf{n}+\mathbf{n}\,\$$	shift: 2
2	$\$_0\mathbf{n}_2$	$+\mathbf{n}+\mathbf{n}\,\$$	reduce: $E \rightarrow \mathbf{n}$
3	$\$_0 E_1$	$+\mathbf{n}+\mathbf{n}\,\$$	shift: 3
4	$\$_0 E_1 +_3$	$\mathbf{n}+\mathbf{n}\,\$$	shift: 4
5	$\$_0 E_1 +_3 \mathbf{n}_4$	$+\mathbf{n}\,\$$	reduce: $E \rightarrow E+\mathbf{n}$
6	$\$_0 E_1$	$\mathbf{n}\,\$$	shift 3
7	$\$_0 E_1 +_3$	$\mathbf{n}\,\$$	shift 4
8	$\$_0 E_1 +_3 \mathbf{n}_4$	$\$$	reduce: $E \rightarrow E+\mathbf{n}$
9	$\$_0 E_1$	$\$$	accept

Corresponding parse tree



Revisit the parentheses again: SLR(1)?

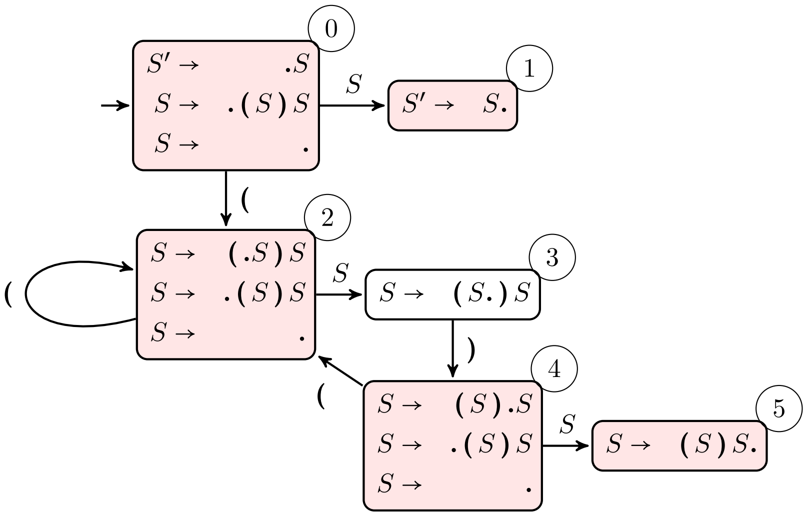
Grammar: parentheses

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & (\,S\,)\,S \mid \epsilon \end{array}$$

Follow set

$Follow(S) = \{), \$\}$

DFA for parentheses



SLR(1) parse table

state	input			goto
	(	)	$	S
0	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	1
1			accept	
2	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	3
3		$s:4$		
4	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	5
5		$r: S \to (S)S$	$r: S \to (S)S$	

Parentheses: SLR(1) parser run (= "reduction")

state	input			goto
	(	)	$	S
0	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	1
1			accept	
2	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	3
3		$s:4$		
4	$s:2$	$r: S \to \epsilon$	$r: S \to \epsilon$	5
5		$r: S \to (S)S$	$r: S \to (S)S$	

stage	parsing stack	input	action
1	$\$_0$	$()()\$$	shift: 2
2	$\$_0(_2$	$)()\$$	reduce: $S \to \epsilon$
3	$\$_0(_2S_3$	$)()\$$	shift: 4
4	$\$_0(_2S_3)_4$	$()\$$	shift: 2
5	$\$_0(_2S_3)_4(_2$	$)\$$	reduce: $S \to \epsilon$
6	$\$_0(_2S_3)_4(_2S_3$	$)\$$	shift: 4
7	$\$_0(_2S_3)_4(_2S_3)_4$	$\$$	reduce: $S \to \epsilon$
8	$\$_0(_2S_3)_4(_2S_3)_4S_5$	$\$$	reduce: $S \to (\,S\,)\,S$
9	$\$_0(_2S_3)_4S_5$	$\$$	reduce: $S \to (\,S\,)\,S$
10	$\$_0S_1$	$\$$	accept

Remarks

Note how the stack grows, and would continue to grow if the sequence of **()** would continue. That's characteristic for a right-recursive formulation of rules, and may constitute a problem for LR-parsing (stack-overflow).

Ambiguity & LR-parsing

- LR(k) (and LL(k)) grammars: *unambiguous*
- definition/construction: free of shift/reduce and reduce/reduce conflict (given the chosen level of look-ahead)
- However: ambiguous grammar tolerable, if (remaining) conflicts can be solved "meaningfully" otherwise:

Additional means of disambiguation:

1. by specifying associativity / precedence "externally"
2. by "living with the fact" that LR parser (commonly) *prioritizes shifts over reduces*

- for the second point ("let the parser decide according to its preferences"):
  - use sparingly and cautiously
  - typical example: *dangling-else*
  - even if parsers makes a decision, programmar may or may not "understand intuitively" the resulting parse tree (and thus AST)
  - grammar with many S/R-conflicts: go back to the drawing board

Example of an ambiguous grammar

$$\begin{array}{rcl} \mathit{stmt} & \to & \mathit{if\text{-}stmt} \mid \mathbf{other} \\ \mathit{if\text{-}stmt} & \to & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmt} \\ & \mid & \mathbf{if}\ (\ \mathit{exp}\ )\ \mathit{stmt}\ \mathbf{else}\ \mathit{stmt} \\ \mathit{exp} & \to & \mathbf{0} \mid \mathbf{1} \end{array}$$

In the following, E for *exp*, etc.

Simplified conditionals

Simplified "schematic" if-then-else

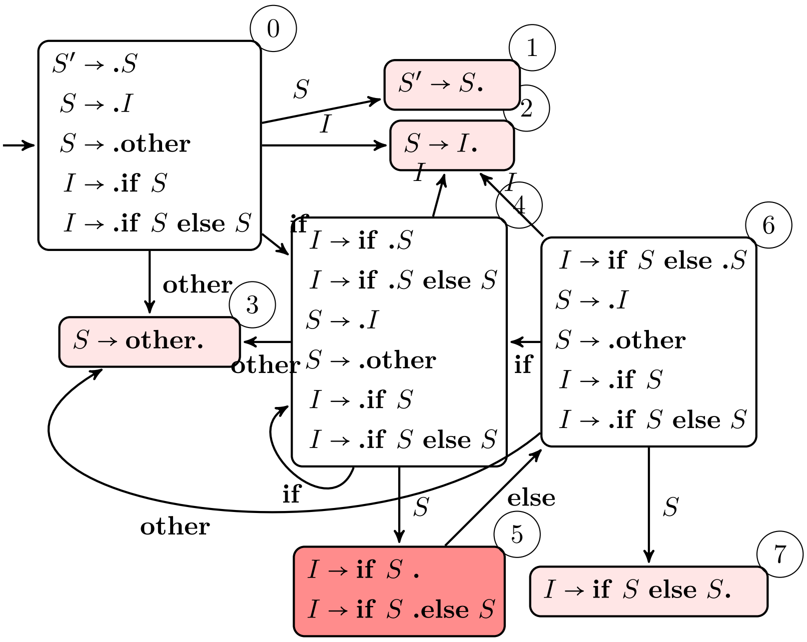
$$\begin{array}{lcl} S & \rightarrow & I \mid \textbf{other} \\ I & \rightarrow & \textbf{if}\ S \mid \textbf{if}\ S\ \textbf{else}\ S \end{array}$$

Follow-sets

	Follow
S'	$\{\$\}$
S	$\{\$, \textbf{else}\}$
I	$\{\$, \textbf{else}\}$

- since ambiguous: at least one conflict must be somewhere

DFA of LR(0) items



Checking the previously shown conditions for SLR(1) parsing, one sees that there is a SLR(1) conflict in state 5: the follow-set of I contains **else**. In the following tables, only the shift-reaction is added in the corresponding slot, since that is the conventional preferred reaction of a parser tool, facing a shift-reduce conflict.

Simple conditionals: parse table

Grammar

$$\begin{array}{lcll} S & \to & I & (1)\\ & | & \textbf{other} & (2)\\ I & \to & \textbf{if}\ S & (3)\\ & | & \textbf{if}S\ \textbf{else}\ S & (4)\end{array}$$

SLR(1)-parse-table, conflict "resolved"

state	input				goto	
	if	**else**	**other**	**$**	S	I
0	$s:4$		$s:3$		1	2
1				accept		
2		$r:1$		$r:1$		
3		$r:2$		$r:2$		
4	$s:4$		$s:3$		5	2
5		$s:6$		$r:3$		
6	$s:4$		$s:3$		7	2
7		$r:4$		$r:4$		

- *shift-reduce conflict* in state 5: reduce with *rule 3* vs. shift (to state 6)
- conflict there: **resolved** in favor of *shift* to 6
- note: extra start state left out from the table

Parser run (= reduction)

state	input				goto	
	if	**else**	**other**	**$**	S	I
0	$s:4$		$s:3$		1	2
1				accept		
2		$r:1$		$r:1$		
3		$r:2$		$r:2$		
4	$s:4$		$s:3$		5	2
5		$s:6$		$r:3$		
6	$s:4$		$s:3$		7	2
7		$r:4$		$r:4$		

stage	parsing stack	input	action
1	$\$_0$	**if if other else other $**	shift: 4
2	$\$_0\textbf{if}_4$	**if other else other $**	shift: 4
3	$\$_0\textbf{if}_4\textbf{if}_4$	**other else other $**	shift: 3
4	$\$_0\textbf{if}_4\textbf{if}_4\textbf{other}_3$	**else other $**	reduce: 2
5	$\$_0\textbf{if}_4\textbf{if}_4S_5$	**else other $**	shift 6
6	$\$_0\textbf{if}_4\textbf{if}_4S_5\textbf{else}_6$	**other $**	shift: 3
7	$\$_0\textbf{if}_4\textbf{if}_4S_5\textbf{else}_6\textbf{other}_3$	**$**	reduce: 2
8	$\$_0\textbf{if}_4\textbf{if}_4S_5\textbf{else}_6S_7$	**$**	reduce: 4
9	$\$_0\textbf{if}_4I_2$	**$**	reduce: 1
10	$\$_0S_1$	**$**	accept

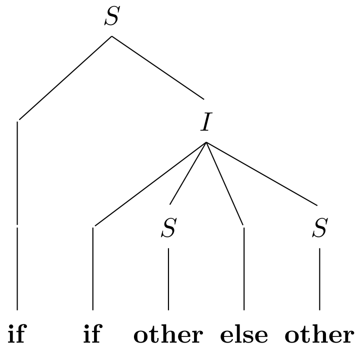
Parser run, different choice

state	input				goto	
	if	**else**	**other**	**$**	S	I
0	$s:4$		$s:3$		1	2
1				accept		
2		$r:1$		$r:1$		
3		$r:2$		$r:2$		
4	$s:4$		$s:3$		5	2
5		$s:6$		$r:3$		
6	$s:4$		$s:3$		7	2
7		$r:4$		$r:4$		

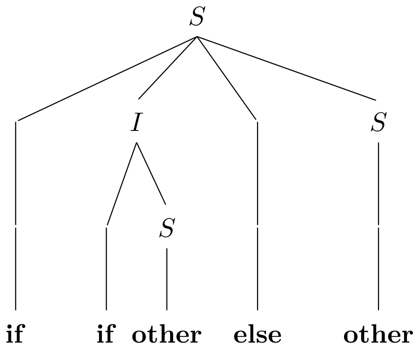
stage	parsing stack	input	action
1	$\$_0$	**if if other else other $**	shift: 4
2	$\$_0\mathbf{if}_4$	**if other else other $**	shift: 4
3	$\$_0\mathbf{if}_4\mathbf{if}_4$	**other else other $**	shift: 3
4	$\$_0\mathbf{if}_4\mathbf{if}_4\mathbf{other}_3$	**else other $**	reduce: 2
5	$\$_0\mathbf{if}_4\mathbf{if}_4 S_5$	**else other $**	reduce 3
6	$\$_0\mathbf{if}_4 I_2$	**else other $**	reduce 1
7	$\$_0\mathbf{if}_4 S_5$	**else other $**	shift 6
8	$\$_0\mathbf{if}_4 S_5\mathbf{else}_6$	**other $**	shift 3
9	$\$_0\mathbf{if}_4 S_5\mathbf{else}_6\mathbf{other}_3$	**$**	reduce 2
10	$\$_0\mathbf{if}_4 S_5\mathbf{else}_6 S_7$	**$**	reduce 4
11	$\$_0 S_1$	**$**	accept

Parse trees: simple conditions

shift-precedence: conventional



“wrong” tree



standard “dangling else” convention

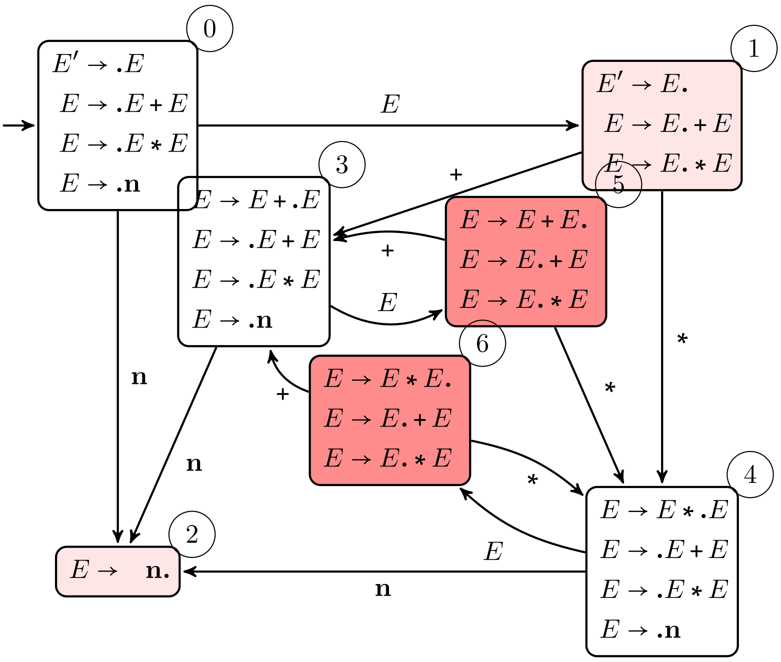
“an **else** belongs to the last previous, still open (= dangling) if-clause”

Use of ambiguous grammars

- advantage of ambiguous grammars: often simpler
- if ambiguous: grammar guaranteed to have conflicts
- can be (often) resolved by specifying *precedence* and *associativity*
- supported by tools like `yacc` and `CUP` ...

$$\begin{array}{lcl} E' & \to & E \\ E & \to & E + E \mid E * E \mid \mathbf{n} \end{array}$$

DFA for + and ×



States with conflicts

- state 5
  - stack contains $... E +E$
  - for input **$**: reduce, since shift not allowed from **$**
  - for input **+**; reduce, as **+** is *left-associative*
  - for input $*$: shift, as $*$ has *precedence* over **+**
- state 6:
  - stack contains $... E $*$E$
  - for input **$**: reduce, since shift not allowed from **$**
  - for input **+**; reduce, a $*$ has *precedence* over **+**
  - for input $*$: shift, as $*$ is *left-associative*
- see also the table on the next slide

Parse table + and ×

state	input				goto
	n	**+**	$*$	**$**	E
0	$s:2$				1
1		$s:3$	$s:4$	accept	
2		$r: E \to \mathbf{n}$	$r: E \to \mathbf{n}$	$r: E \to \mathbf{n}$	
3	$s:2$				5
4	$s:2$				6
5		$r: E \to E + E$	$s:4$	$r: E \to E + E$	
6		$r: E \to E * E$	$r: E \to E * E$	$r: E \to E * E$	

How about exponentiation (written ↑ or $**$)?

Defined as *right-associative*. See exercise

The interesting line is the one for state 5, and the difference in reaction when ecncountering a addition vs. a multiplication sign. Basically, the shift for multiplication realizes the fact that multiplication has a higher precedence than addition

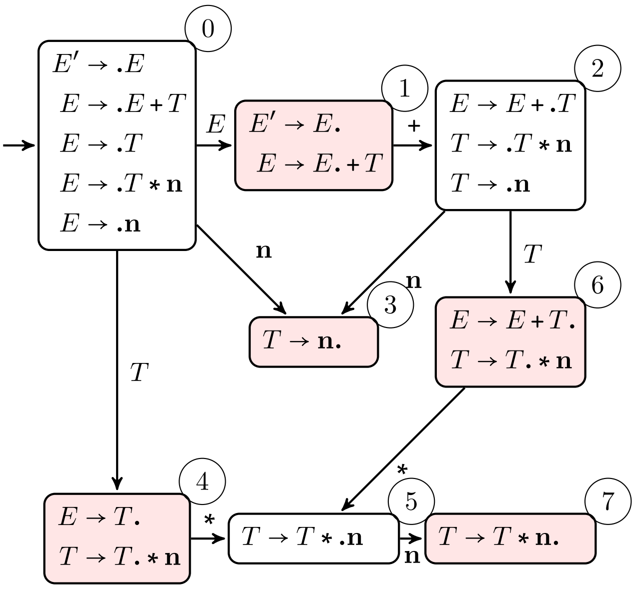
For comparison: unambiguous grammar for + and $*$

Unambiguous grammar: precedence and left-assoc built in

$$\begin{array}{rcl} E' & \to & E \\ E & \to & E + T \mid T \\ T & \to & T * \mathbf{n} \mid \mathbf{n} \end{array}$$

	Follow	
E'	$\{\$\}$	(as always for start symbol)
E	$\{\$, +\}$	
T	$\{\$, +, *\}$	

DFA for unambiguous + and ×



DFA remarks

- the DFA now is SLR(1)
  - check states with *complete* items

    state 1: $\mathit{Follow}(E') = \{\$\}$

    state 4: $\mathit{Follow}(E) = \{\$, +\}$

    state 6: $\mathit{Follow}(E) = \{\$, +\}$

state 3/7: $\mathit{Follow}(T) = \{\$, +, *\}$

- in no case there's a shift/reduce conflict (check the outgoing edges vs. the follow set)
- there's not reduce/reduce conflict either

LR(1) parsing

- most general from of LR(1) parsing
- aka: *canonical* LR(1) parsing
- usually: considered as unecessarily "complex" (i.e. LALR(1) or similar is good enough)
- "stepping stone" towards LALR(1)

Basic restriction of SLR(1)

Uses *look-ahead*, yes, but only *after* it has built a non-look-ahead DFA (based on **LR(0)**-items)

A help to remember

SLR(1) "improved" LR(0) parsing LALR(1) is "crippled" LR(1) parsing.

Limits of SLR(1) grammars

Assignment grammar fragment[17]

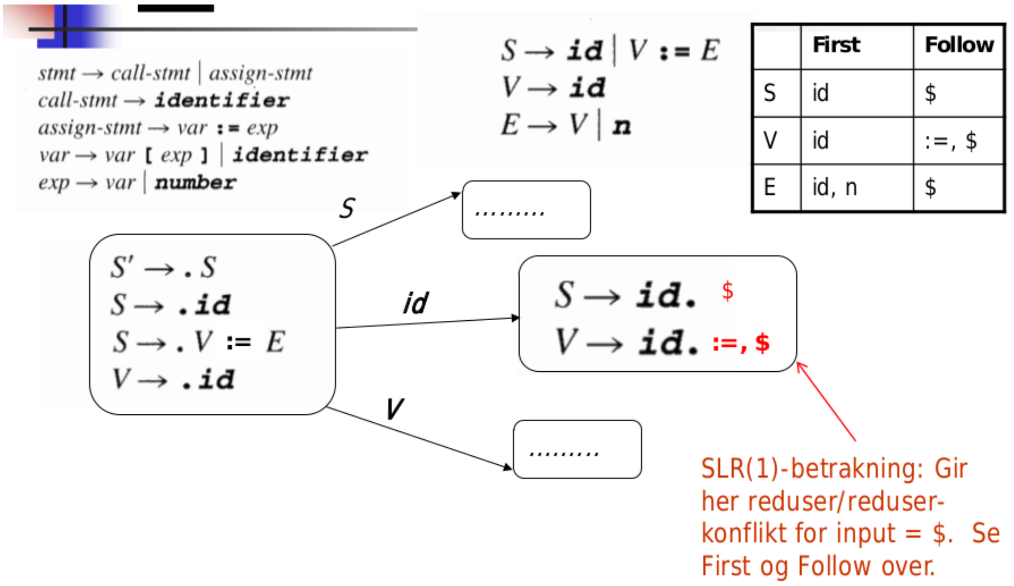
$$\begin{array}{rcl} \mathit{stmt} & \to & \mathit{call\text{-}stmt} \mid \mathit{assign\text{-}stmt} \\ \mathit{call\text{-}stmt} & \to & \mathbf{identifier} \\ \mathit{assign\text{-}stmt} & \to & \mathit{var} \mathbin{\mathbf{:=}} \mathit{exp} \\ \mathit{var} & \to & [\,\mathit{exp}\,] \mid \mathbf{identifier} \\ \mathit{exp} & \to & \mathit{var} \mid \mathbf{n} \end{array}$$

Assignment grammar fragment, simplified

$$\begin{array}{rcl} S & \to & \mathbf{id} \mid V \mathbin{\mathbf{:=}} E \\ V & \to & \mathbf{id} \\ E & \to & V \mid \mathbf{n} \end{array}$$

[17] Inspired by Pascal, analogous problems in C . . .

non-SLR(1): Reduce/reduce conflict



stmt → *call-stmt* | *assign-stmt*
call-stmt → **identifier**
assign-stmt → *var* **:=** *exp*
var → *var* **[** *exp* **]** | **identifier**
exp → *var* | **number**

$S \to \mathbf{id} \mid V := E$
$V \to \mathbf{id}$
$E \to V \mid \mathbf{n}$

	First	Follow
S	id	$
V	id	:=, $
E	id, n	$

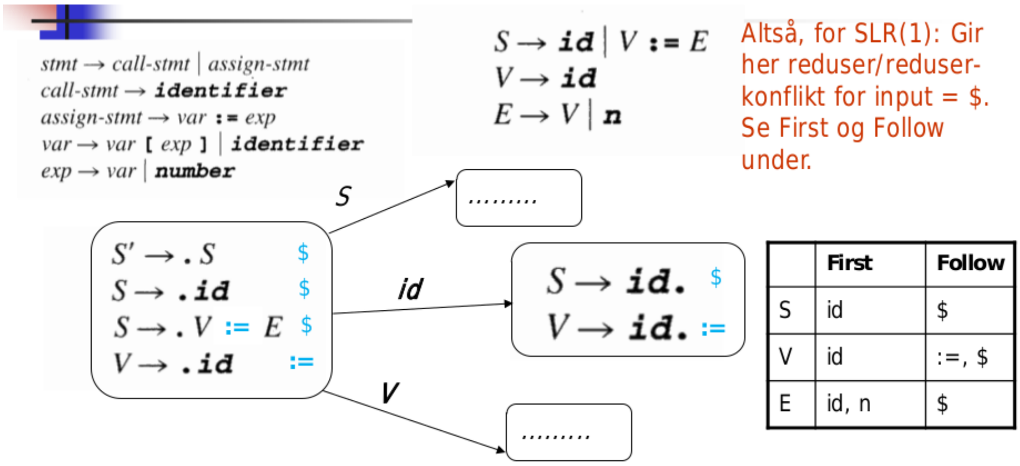
$S' \to .S$
$S \to .\mathbf{id}$
$S \to .V := E$
$V \to .\mathbf{id}$

$S \to \mathbf{id}.$ $
$V \to \mathbf{id}.$:=, $

SLR(1)-betrakning: Gir her reduser/reduser-konflikt for input = $. Se First og Follow over.

Checking the previously shown conditions for SLR(1)-parsing shows (amongst other) a reduce/reduce conflict situation in the state on the right-hand side. The R/R conflict is on the symbol **$**: the parser does not know which production to use in the reduce step. The red terminals are not part of the state, they are just shown for illustration (representing the follow symbols of S resp. of V). The LR(1) construction (sketched on the next slides) builds in one additional look-ahead symbol officially as parts of the items and thus states.

Situation can be saved: more look-ahead



stmt → *call-stmt* | *assign-stmt*
call-stmt → **identifier**
assign-stmt → *var* **:=** *exp*
var → *var* **[** *exp* **]** | **identifier**
exp → *var* | **number**

$S \to \mathbf{id} \mid V := E$
$V \to \mathbf{id}$
$E \to V \mid \mathbf{n}$

Altså, for SLR(1): Gir her reduser/reduser-konflikt for input = $. Se First og Follow under.

$S' \to .S$ $
$S \to .\mathbf{id}$ $
$S \to .V := E$ $
$V \to .\mathbf{id}$:=

$S \to \mathbf{id}.$ $
$V \to \mathbf{id}.$:=

	First	Follow
S	id	$
V	id	:=, $
E	id, n	$

The (sketch of the) automaton here looks pretty similar to the previous one. However, we should think now of the non-terminals as officially part of the items. The interesting piece in this example is the transition from the initial state following the **id**-transition, to the state containing the items → **id.** and V → **id.**. That was the state on the previous slide with the reduce/reduce conflict (on the following symbol **$**). Now, without showing the construction in detail (later we give at least the rules for the construction of the NFA, not the DFA with the closure): the interesting situation is, in the first state, the item $S \to .V := E, \$$. With the **.** in front of the V, that's when we have to take the ϵ-closure into account, basically adding also the initial items (here one initial item) for the productions for V into account. Now, by adding that item $V \to .\mathbf{id}$, we can use the additional "look-ahead piece of information" in that item to mark that V was added to the close when being /in front of an **:=**. That leads (in this situation) to the item of the form $[V \to .\mathbf{id}, :=]$. This information is more specific than the knowledge about the general follow-set of V, which constains **:=** and **$**. Now, by recording that extra piece of information in the close, the state remembers that the only thing at the current state that is allowed to follow the V is the **:=**. That will defuse the discussed conflict, namely as follows: if we follow the **id**-arrow, we end up in the state on the right-hand side. Such a transition does not touch the additional new look-ahead information (here the **$** resp the **:=** symbol). Thus, in the state at the right-hand side, the reduce-reduce conflict has disappeared!

LALR(1) (and LR(1)): Being more precise with the follow-sets

- LR(0)-items: too "indiscriminate" wrt. the follow sets
- remember the definition of SLR(1) conflicts
- LR(0)/SLR(1)-states:
  - sets of items[18] due to subset construction
  - the items are LR(0)-items
  - follow-sets as an *after-thought*

Add precision in the states of the automaton already

Instead of using LR(0)-items and, when the LR(0) DFA is done, try to add a little disambiguation with the help of the follow sets for states containing complete items, better **make more fine-grained items** from the very start:

- **LR(1) items**
- each *item* with "specific follow information": look-ahead

LR(1) items

- main idea: simply make the look-ahead part of the item
- obviously: proliferation of states[19]

LR(1) items

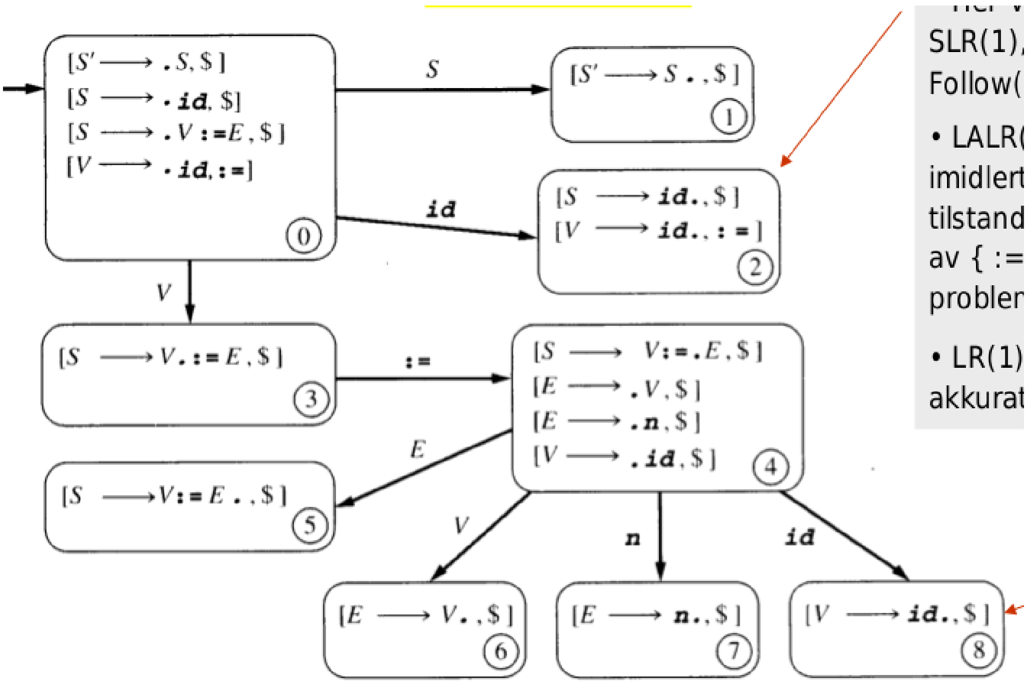
$$[A \to \alpha.\beta, \mathbf{a}] \tag{4.9}$$

- **a**: terminal/token, including **$**

[18] That won't change in principle (but the items get more complex)

[19] Not to mention if we wanted look-ahead of $k > 1$, which in practice is not done, though.

LALR(1)-DFA (or LR(1)-DFA)



Remarks on the DFA

- Cf. state *2* (seen before)
  - in SLR(1): problematic (reduce/reduce), as *Follow*(*V*) = {**:=**, **$**}
  - now: diambiguation, by the added information
- LR(1) would give the same DFA

Full LR(1) parsing

- AKA: **canonical** LR(1) parsing
- the *best* you can do with 1 look-ahead
- unfortunately: big tables
- pre-stage to LALR(1)-parsing

SLR(1)

LR(0)-item-based parsing, with *afterwards* adding some extra “pre-compiled” info (about follow-sets) to increase expressivity

LALR(1)

LR(1)-item-based parsing, but *afterwards* throwing away precision by collapsing states, to save space

LR(1) transitions: arbitrary symbol

- transitions of the **NFA** (not DFA)

X-transition

$$\boxed{[A \to \quad \alpha.X\beta, \mathbf{a}]} \xrightarrow{X} \boxed{[A \to \quad \alpha X.\beta, \mathbf{a}]}$$

LR(1) transitions: ϵ

ϵ-transition

for all

$$B \to \beta_1 \mid \beta_2 \ldots \quad \text{and all} \quad \mathbf{b} \in \mathit{First}(\gamma\mathbf{a})$$

$$\boxed{[A \to \alpha.B\gamma \quad ,\mathbf{a}]} \xrightarrow{\epsilon} \boxed{[B \to .\beta \quad ,\mathbf{b}]}$$

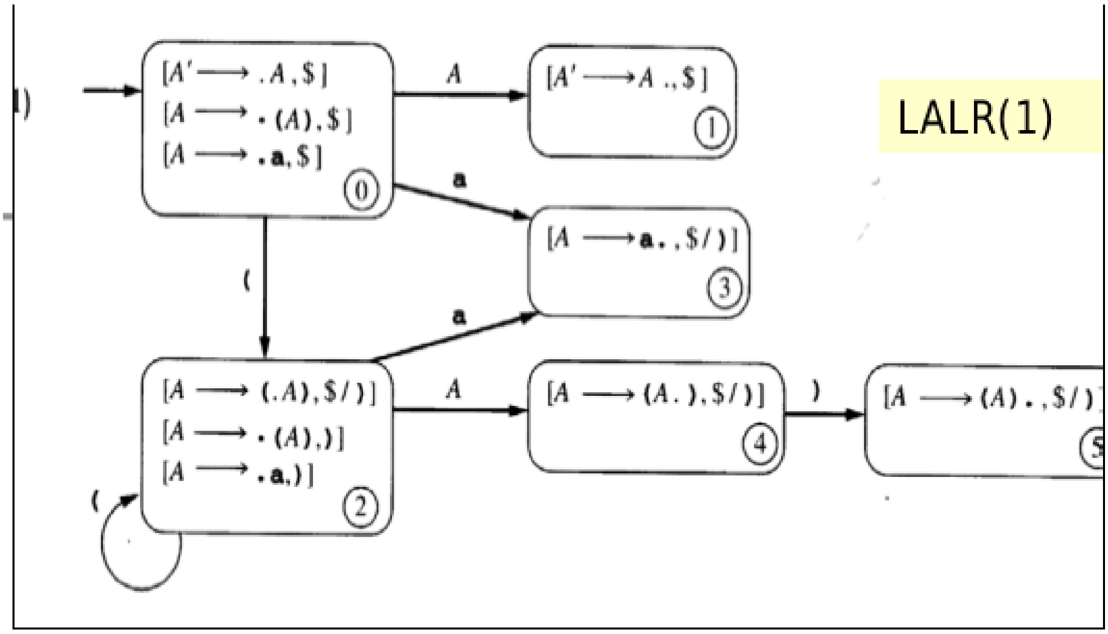
including special case $(\gamma = \epsilon)$

$$\text{for all } B \to \beta_1 \mid \beta_2 \ldots$$

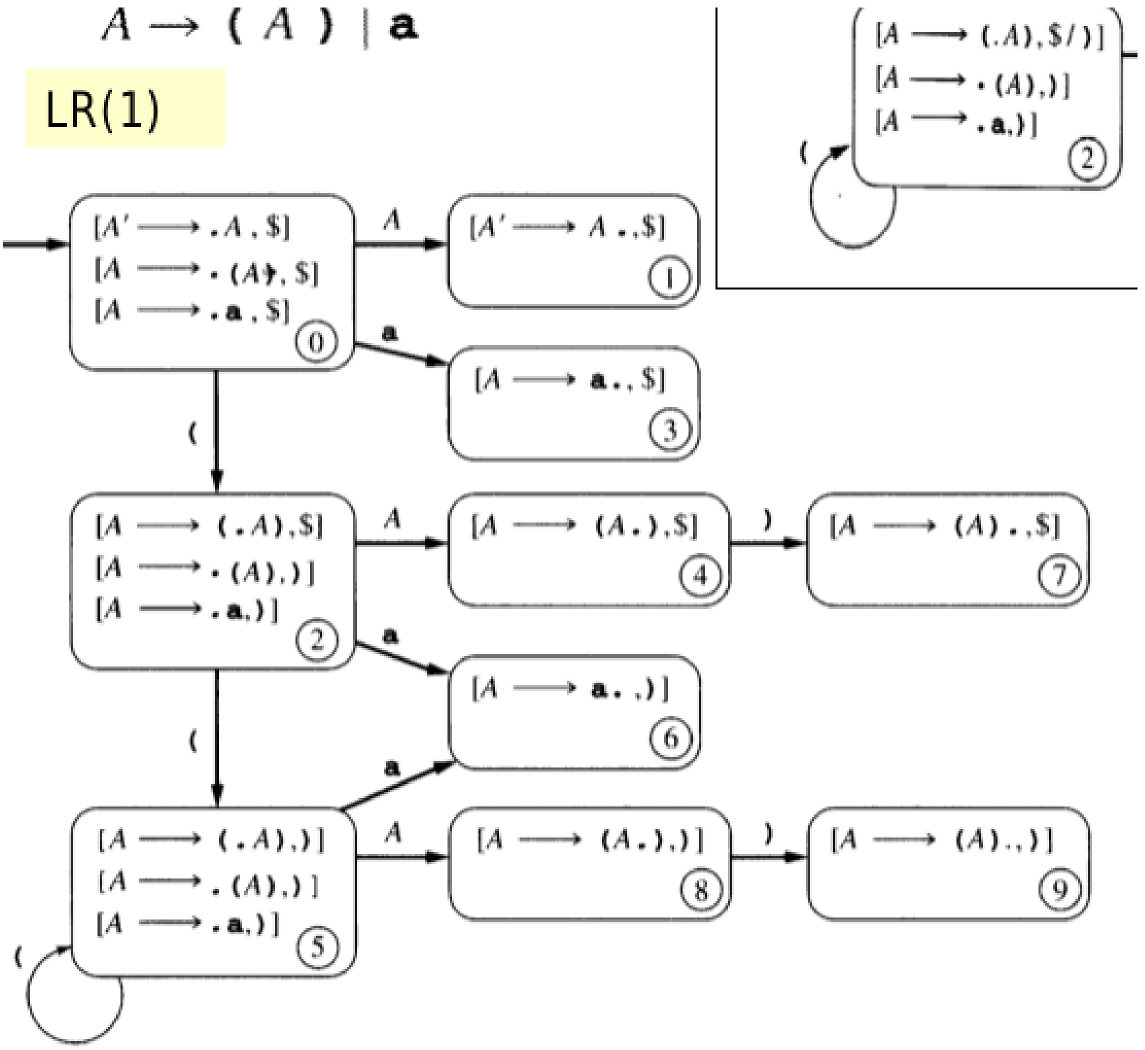
$$\boxed{[A \to \alpha.B \quad ,\mathbf{a}]} \xrightarrow{\epsilon} \boxed{[B \to .\beta \quad ,\mathbf{a}]}$$

LALR(1) vs LR(1)

LALR(1)



LR(1)



Core of LR(1)-states

- actually: not done that way in practice
- main idea: *collapse* states with the same *core*

Core of an LR(1) state

= set of *LR(0)*-items (i.e., ignoring the look-ahead)

- observation: core of the LR(1) item = LR(0) item
- 2 LR(1) states with the same core have same outgoing edges, and those lead to states with the same core

LALR(1)-DFA by as collapse

- collapse all states with the same core
- based on above observations: edges are also consistent
- Result: almost like a LR(0)-DFA but additionally
  - still each individual item has still look ahead attached: the **union** of the "collapsed" items
  - especially for states with *complete* items $[A \to \alpha, \mathbf{a}, \mathbf{b}, \ldots]$ is **smaller** than the follow set of A
  - $\Rightarrow$ less unresolved conflicts compared to SLR(1)

Concluding remarks of LR / bottom up parsing

- all constructions (here) based on BNF (not EBNF)
- *conflicts* (for instance due to ambiguity) can be solved by
  - reformulate the grammar, but generarate the same language[20]
  - use *directives* in parser generator tools like `yacc`, `CUP`, `bison` (precedence, assoc.)
  - or (not yet discussed): solve them later via *semantical analysis*
  - NB: *not all* conflics are solvable, also not in LR(1) (remember ambiguous languages)

LR/bottom-up parsing overview

	advantages	remarks
LR(0)	defines states *also* used by SLR and LALR	not really used, many conflicts, very weak
SLR(1)	clear improvement over LR(0) in expressiveness, even if using the same number of states. Table typically with 50K entries	weaker than LALR(1). but often good enough. Ok for hand-made parsers for *small* grammars
LALR(1)	almost as expressive as LR(1), but number of states as LR(0)!	method of choice for most generated LR-parsers
LR(1)	*the* method covering *all* bottom-up, one-look-ahead parseable grammars	large number of states (typically 11M of entries), mostly LALR(1) preferred

Remeber: once the *table* specific for LR(0), . . . is set-up, the parsing algorithms all work *the same*

Error handling

Minimal requirement

Upon "stumbling over" an error (= deviation from the grammar): give a *reasonable* & *understandable* error message, indicating also error *location*. Potentially stop parsing

[20] If designing a new language, there's also the option to massage the language itself. Note also: there are *inherently* ambiguous *languages* for which there is no *unambiguous* grammar.

- for parse error *recovery*
  - one cannot really recover from the fact that the program has an error (an syntax error is a syntax error), but
  - after giving decent error message:
    - move on, potentially jump over some subsequent code,
    - until parser can *pick up* normal parsing again
    - so: meaningfull checking code even following a first error
  - avoid: reporting an avalanche of subsequent *spurious* errors (those just "caused" by the first error)
  - "pick up" again after semantic errors: easier than for syntactic errors

Error messages

- important:
  - avoid error messages that only occur because of an already reported error!
  - report error as early as possible, if possible at the *first point* where the program cannot be extended to a correct program.
  - make sure that, after an error, one doesn't end up in an *infinite loop* without reading any input symbols.
- What's a good error message?
  - assume: that the method `factor()` chooses the alternative **(** *exp* **)** but that it , when control returns from method `exp()`, does not find a **)**
  - one could report : `right parenthesis missing`
  - But this may often be confusing, e.g. if what the program text is: `( a + b c )`
  - here the `exp()` method will terminate after `( a + b`, as `c` cannot extend the expression). You should therefore rather give the message `error in expression or right parenthesis missing`.

Error recovery in bottom-up parsing

- *panic recovery* in LR-parsing
  - simple form
  - the only one we shortly look at
- upon error: recovery ⇒
  - pops parts of the stack
  - ignore parts of the input
- until "on track again"
- but: how to do that
- additional problem: *non-determinism*
  - table: constructed *conflict-free* **under normal operation**
  - upon error (and clearing parts of the stack + input): no guarantee it's clear how to continue

⇒ **heuristic** needed (like panic mode recovery)

Panic mode idea

- try a **fresh start**,
- promising "fresh start" is: a possible **goto** action
- thus: back off and take the *next* such goto-opportunity

Possible error situation

	parse stack	input	action
1	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3(_4\mathbf{d}_5\mathbf{e}_6$	**f** **)** **gh** . . . **\$**	no entry for **f**
2	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3 B_v$	**gh** . . . **\$**	back to normal
3	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3 B_v \mathbf{g}_7$	**h** . . . **\$**	. . .

state	input					goto			
	. . .	**)**	**f**	**g**	. . .	. . .	A	B	. . .
. . .									
3							u	v	
4			–				–	–	
5			–				–	–	
6		–	–				–	–	
. . .									
u		–	–	reduce . . .					
v		–	–	shift : 7					
. . .									

Panic mode recovery

Algo

1. *Pop* states for the stack *until* a state is found with non-empty **goto** entries
2. - If there's legal action on the current input token from one of the goto-states, push token on the stack, *restart* the parse.
   - If there's several such states: *prefer shift* to a reduce
   - Among possible reduce actions: prefer one whose associated non-terminal is least general
3. if no legal action on the current input token from one of the goto-states: *advance input* until there is a legal action (or until end of input is reached)

Example again

	parse stack	input	action
1	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3(_4\mathbf{d}_5\mathbf{e}_6$	**f** **)** **gh** . . . **\$**	no entry for **f**
2	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3 B_v$	**gh** . . . **\$**	back to normal
3	$\$_0\mathbf{a}_1\mathbf{b}_2\mathbf{c}_3 B_v \mathbf{g}_7$	**h** . . . **\$**	. . .

- first pop, until in state 3
- then jump over input
  - until next input **g**
  - since **f** and **)** cannot be treated
- choose to goto v (shift in that state)

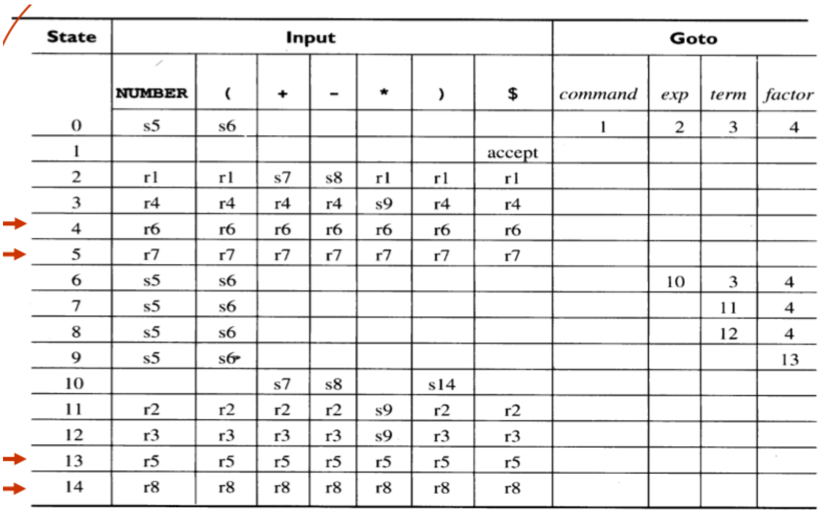
Panic mode may loop forever

	parse stack	input	action
1	$\$_0$	**(n n) \$**	
2	$\$_0(_6$	**n n) \$**	
3	$\$_0(_6\mathbf{n}_5$	**n) \$**	
4	$\$_0(_6 factor_4$	**n) \$**	
6	$\$_0(_6 term_3$	**n) \$**	
7	$\$_0(_6 exp_{10}$	**n) \$**	panic!
8	$\$_0(_6 factor_4$	**n) \$**	been there before: stage 4!

Typical `yacc` parser table

some variant of the expression grammar again

$$
\begin{array}{rcl}
\mathit{command} & \rightarrow & \mathit{exp} \\
\mathit{exp} & \rightarrow & \mathit{term} * \mathit{factor} \mid \mathit{factor} \\
\mathit{term} & \rightarrow & \mathit{term} * \mathit{factor} \mid \mathit{factor} \\
\mathit{factor} & \rightarrow & \mathbf{n} \mid (\, \mathit{exp} \,)
\end{array}
$$



State	Input							Goto			
	NUMBER	(	+	-	*	)	$	*command*	*exp*	*term*	*factor*
0	s5	s6						1	2	3	4
1							accept				
2	r1	r1	s7	s8	r1	r1	r1				
3	r4	r4	r4	r4	s9	r4	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6								12	4
9	s5	s6									13
10			s7	s8		s14					
11	r2	r2	r2	r2	s9	r2	r2				
12	r3	r3	r3	r3	s9	r3	r3				
13	r5	r5	r5	r5	r5	r5	r5				
14	r8	r8	r8	r8	r8	r8	r8				

Panicking and looping

	parse stack	input	action
1	$\$_0$	$\mathbf{(\,n\;n\,)}\,\$$	
2	$\$_0(_6$	$\mathbf{n\;n\,)}\,\$$	
3	$\$_0(_6\mathbf{n}_5$	$\mathbf{n\,)}\,\$$	
4	$\$_0(_6\mathit{factor}_4$	$\mathbf{n\,)}\,\$$	
6	$\$_0(_6\mathit{term}_3$	$\mathbf{n\,)}\,\$$	
7	$\$_0(_6\mathit{exp}_{10}$	$\mathbf{n\,)}\,\$$	panic!
8	$\$_0(_6\mathit{factor}_4$	$\mathbf{n\,)}\,\$$	been there before: stage 4!

- error raised in stage 7, no action possible
- panic:
  1. pop-off exp_{10}
  2. state 6: 3 goto's

	exp	*term*	*factor*
goto to	10	3	4
with **n** next: action there	—	reduce r_4	reduce r_6

  3. no shift, so we need to decide between the two reduces
  4. *factor*: less general, we take that one

How to deal with looping panic?

- make sure to detec loop (i.e. previous “configurations”)
- if loop detected: doen't repeat but do something special, for instance
  - pop-off more from the stack, and try again

  - pop-off and *insist* that a shift is part of the options

Left out (from the book and the pensum)

- more info on error recovery
- expecially: more on `yacc` error recovery
- it's not pensum, and for the oblig: need to deal with CUP-specifics (not classic `yacc` specifics even if similar) anyhow, and error recovery is not part of the oblig (halfway decent error *handling* is).

4.6 Material from [6]

Top-down parsing

Growing the parse tree from the root to the leaves. The fringe or frontier of the tree (i.e., the current leaves) are a mixture of terminals and non-terminals. Then growing the tree means *expanding* one non-terminal with one corresponding right-hand side. On that very abstract level, that process contains 2 forms of non-determinism: with more than one non-terminal in the current sentential form, which one should one expand? Secondly, which production or rule should be used for that expansion. Of course one should not expand randomly, but "guided" towards the input word, i.e., of course the word being parsed (which consists of terminals only), should give indication, which choice is to be made. Above we made the distinction that there are two kind of choices: "where" and "how" to expand. As it turns out, the "where"-choice is not fundamentally important in the following sense: if there are 2 non-terminal to expand, choosing the left-one first or the right first will result in the same ultimate parse tree(s). After all, when a tree is built, it does not matter any more, in which order the branches had been added (except that we grow the tree top-down, starting from the root, i.e., the start symbol of the grammar).

Now, since the "where" question does not matter, one simply decises that one routinely expands the *left-most* non-terminal. That process then is called a *left-most* derivation. In the section for top-down parsing, it will always be a parse-tree construction connected to a left-most derivation. It is also rather natural for a (top-down) parser to construct a left-most derivation considering the fact that the parser eats tokens one by one going "from left to right" through its input.

With left-most derivation so "natural" one may ask: is there a place of right-most derivations¿Not in this section about top-down parsing, it's always left-most. Later, thought, bottom-up parsers build the parse tree in a way, that is connected to right-most derivations! Still, those parsers eat the tokens left-to-right. The working of such bottom-up parsers are a bit less intuitive than top-down parsers, but they are quite important and we come back to them later.

Now with the question of "where" to expand a non-terminal out of the way, one can concentrate on "how" (i.e., which production to choose). In general, there is more an one production per non-terminal. A grammar which has one rule only per non-terminal would either be "trivial" or "defective" (one can shortly reflect on that). Now, if there is more than one production to choose from, it may be the case that one choice will lead to a successful parse-tree (i.e., a successful parse) and another one will not. To find a successful (left-most) derivation and corresponding parse-tree therefore will involve *backtracking*. Another consequence could be that there is more than one successful parse-tree. In that case, the grammar is called *ambiguous*.

Backtracking in parsing is to be avoided (and also one does generally want to avoid ambiguous grammars, or to the very least: for an ambiguous grammar there must be a rational and clear way to resolve any ambiguity, that will also be discussed later).

The intution of how to avoid backtracking is pretty simple. The parser works from left-to-right, it faces a non-terminal and "the rest of the input" and has to make a decision: which rule to apply. Now, since the parser does not want to "rewind" input (in avoidance of backtracking), it's the future rest of the input which should *determine* the production to choose. The amount of next tokens needed to make the decision is called the *look-ahead*. In the easiest case, it's the *next* single token in which case one has a look-ahead of 1.

In the more general cases, the parser might be able to make a correct decision based on a fixed look-ahead of size k. There may also be grammars for which backtracking can be avoided, but one cannot give a fixed k for having a maximal look-head up-front. The look-ahead may be arbitrarily long, depending on the word. Finally, of course, there are grammars where no amount of look-ahead can resolve the decision.

We are interested in the easier (and also practically relevant) cases where backtracking can be avoided, with, say 1 look-ahead.

Transforming a grammar for top-down parsing.

1. A top-down parser with oracular choice
   As the name implies, oracular choice is some "algorithm" where the top-down parser always chooses the right production. Oracular can be understood as having unlimited look-ahead. In that case, the running-time is proportional to the length of the input, obviously.
   One problem indicated by the example is that there are *inconsistent* choices. Inconsistent means: given a non-terminal *and* the complete rest of the input, different choices are being made. That means: there cannot be a top-down parser that avoids backtracking!
   The derivation is based on the standard naive expression grammar (which contains la

Chapter
Semantic analysis

What is it about?

Learning Targets of this Chapter

1. “attributes”
2. attribute grammars
3. synthesized and inherited attributes
4. various applications of attribute grammars

Contents

5.1 Introduction

Semantic analysis in general

Semantic analysis or *static analysis* is a very broad and diverse topic. The lecture concentrates on a few, but crucial aspects. This particular chapter here is concerned with *attribute grammars*. It’s a generic or general “framework” to “semantic analysis”. Later chapters also deal with semantic analysis, namely the one about *symbol tables* and for *type checking*. In the context of the lecture, those chapters all work basically on (abstract syntax) trees (except that for the symbol tables and for the type system, it’s not so visible). The fact that it’s a mechanism to “analyze trees” is most visible for attribute grammars: context-free grammars describe trees and the semantic rules (see later) added to the grammar specify how to analyze resulting trees.

Wrt. the general placement of semantic analysis in a compiler: First, not all semantic analyses are “tree analyses”. Data flow analysis (on which we touch upon later) often works on *graphs* (typically control flow graphs). Furthermore, it’s not the case, that semantic analysis restricted to be done directly after parsing. There are many semantic analyses that are done at later stages (and on other representations). In particular, it could be that a later intermediate reprasentation uses a different form of syntax, closer to machine code (often call *intermediate code*). That syntax could also be given by a grammar, meaning that a program in that syntax corresponds to a tree of that syntax. As a result, one can apply techniques like attribute grammars also at that level (maybe thereby using in on the AST, and later differently on some intermediate code).

Overview

On a very high level, the attribute grammar format does the following: it enhances a given grammar by additional, so called *semantic rules*, which *specify* how trees conforming to the grammar should be analysed.

Two points might be noted here. First, the AG formalism adds rules on top of context-free *grammars*, but the intention is to specify analyses on *trees* formed *according to the given grammar*. Secondly, it’s a *specification* of such tree analyses. The AG format is very general, meaning that it allows to express all kinds of ways attributes should be evaluated. If not constrained in some way, the AG formalism can be seen as too expressive in that it leads to specifications contradict themselves or does not lead to proper implementation.

Part of the chapter therefore will be concerned with *restrictions* it receives from the parser an abstract syntax tree, and then "analyses" it. On a very high level, as far as attribute grammars is concerned, semantic analysis is about "tree algorithms".[1] Attribute grammars is a formalism that takes context-free grammars and adds so-called "semantical rules" to it. AGs in their general form can be seen as a "specification" formalisism for attributes in a grammar.

Side remark: XML

As some side remark, and not part of the technical content of the lecture: XML is some "exchange format" or markup language built around "trees". "markup" is kind of like the opposite of "markdown" (tongue-in-cheek): mark-down allows easy textual representation, optimized for "human consumption". Mark-up easy consumption for "machines" (easy, unique parsing, easy exchange of "texts"). That's why XML reads so horrible to the naked eye.[2] Anyway, since pieces of XML-data are *trees*, there is also the notion of *grammars* according to which such trees are considered well-formed. In the XML terminology, that corresponds basically to *schemas*.[3] That being so, there are tools that check whether a tree adheces to a given schema, a problem that in that form does not present itself in parsing: the parser process generates *only* trees in the AST format. Since XML processing is concerned with "tree processing" (checking, transformation etc), there are some similarities with attribute grammars and some XML related technologies. We don't go deeper than that here.

Overview over the chapter resp. SA in general

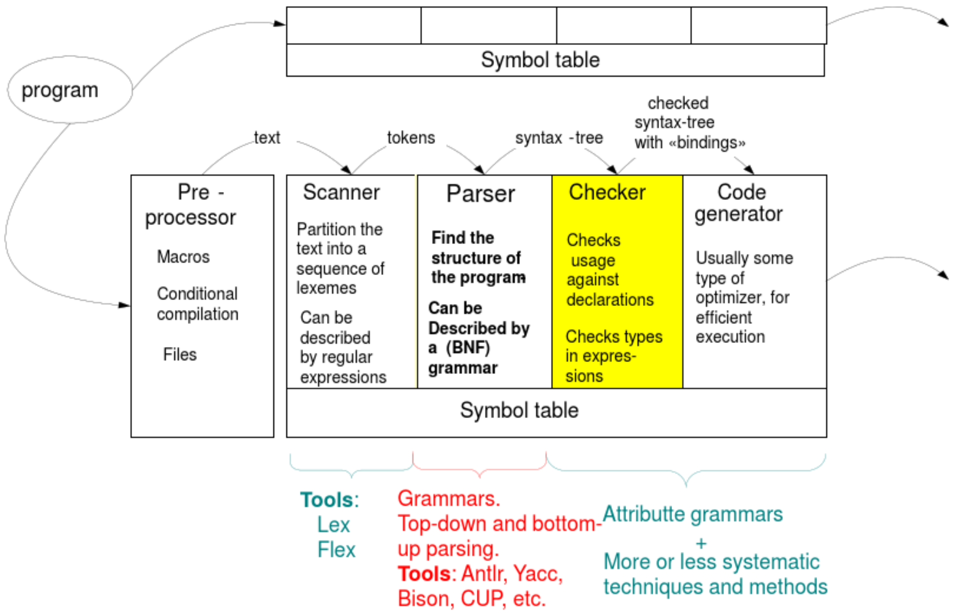
- semantic analysis in general
- attribute grammars (AGs)
- symbol tables (not today)
- data types and type checking (not today)

[1] It should be noted that semantic analysis is not restricted to analysing abstract syntax trees that come out of the parser. That's, however, the placement in the lecture. Semantic analysis may also be applied to intermediate representations *other* than abstract syntax trees. One example being control flow graphs.

[2] The `build.xml` from the oblig is some example of some xml-kind of file, used for "building" a project with *ant*.

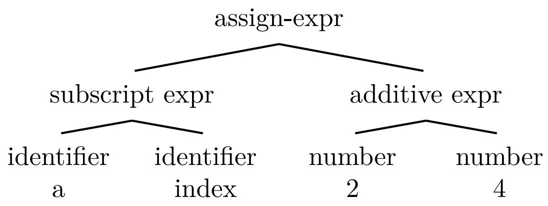
[3] In UML context, the role of a grammar is taken by something with the slightly confusing title "meta-model".

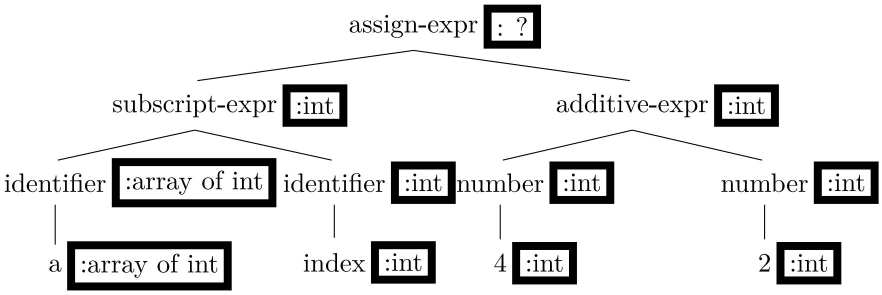
Where are we now?


Symbol table
program
text
tokens
syntax -tree
checked syntax-tree with «bindings»
Pre - processor
Macros
Conditional compilation
Files
Scanner
Partition the text into a sequence of lexemes
Can be described by regular expressions
Parser
Find the structure of the program
Can be Described by a (BNF) grammar
Checker
Checks usage against declarations
Checks types in expres- sions
Code generator
Usually some type of optimizer, for efficient execution
Symbol table
Tools: Lex Flex
Grammars. Top-down and bottom-up parsing.
Tools: Antlr, Yacc, Bison, CUP, etc.
Attributte grammars
+
More or less systematic techniques and methods

What do we get from the parser?

- output of the parser: (abstract) syntax tree
- often: in anticipation: nodes in the tree contain "space" to be filled out by SA
- examples:
  - for expression nodes: *types*
  - for identifier/name nodes: reference or pointer to the *declaration*





By “space”, one might think of *fields* or *instance variables* in an object-oriented setting. Fields can be seen as one way to implenent “attributes”. When introducing attribute grammars, the notion of *attribute* will be a specific concept, namely the specific form of attributes in an atrribute grammar. But very general, an “attribute” means just a “property attached to some element”. Typically here, attached to syntactic representations of the language, in particular to nodes in the abstract syntax tree. Since the notion of attribute is this so general, it can take very different forms (like types, data flow information, all kind of extra information). Also, attributes in that sense, need to be “attached” to abstract syntax tree only. For instance, data flow information is extra information (calculated by data flow analysis) not to a syntax tree, but to something called a *control-flow graph*. So, since such graphs are *not* described by context-free grammars, and therefore, data flow analyses will *not* be described by attribute grammars.[4]

General: semantic (or static) analysis

Rule of thumb

Check everything which is possible *before* executing (run-time vs. compile-time), but cannot already done during lexing/parsing (syntactical vs. semantical analysis)

Rest:

- Goal: fill out “semantic” info (typically in the AST)
- typically:
  - all *names declared*? (somewhere/uniquely/before use)
  - *typing*:
    - is the declared type consistent with use
    - types of (sub)-expression consistent with used operations
- *border* between sematical vs. syntactic checking not always 100% clear
  - `if a then ...`: checked for syntax
  - `if a + b then ...`: semantical aspects as well?

SA is nessessarily approximative

- note: not all can (precisely) be checked at compile-time
  - division by zero?
  - “array out of bounds”
  - “null pointer deref” (like `r.a`, if `r` is null)
- but note also: *exact* type cannot be determined statically either

`if x then 1 else "abc"`

- statically: ill-typed[5]

[4] Besides the reason mentioned —data-flow analyses typically operate on graphs, not trees— there is a second reason why DFA will in general not be done with AGs; the evaluation of AGs on a concrete tree explicitly disallows *cycles* in the dependency graph (see later). DFA in the general form *definitely* will have to handle cyclic situations.

[5] Unless some fancy behind-the-scence type conversions are done by the language (the compiler). Perhaps `print(if x then 1 else "abc")` is accepted, and the integer `1` is implicitly converted to `"1"`.

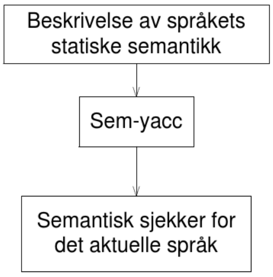
- dynamically ("run-time type"): `string` or `int`, or run-time type error, if `x` turns out not to be a boolean, or if it's null

The fact that one cannot *precisely* check everything at compile-time is due to *fundamental* reasons. It's fundamentally impossible to predict the behavior of a program (provided, the programming language is expressive enough = Turing complete, which can be taken as granted for all off-the-shelf general programming languages). The "fundamental reasons" mentioned above basically is a result of the famous *halting problem*. The particular version here is a consequence of that halting problem and is know as **Rice's theorem**. Actually it's more pessimisic than the sentence on the slide: Rice stipulates: **all** non-trivial semantic problems of a programming language are undecidable. If it were otherwise, the halting problem would be decidable as well (which it isn't, end-of-proof). Note that *approximative* checking is doable, resp. that's what the SA is doing anyhow.

As for type checking: the footnote refers to something which is a form of *polymorphism*, which is a form of "laxness" or "liberarility" of the type system, which allows that some element of the language can have more than one type. In the particular example, it would be a specific form of polymorphism, namely (operator) overloading, in that + is used for addition as well as string concatenation. Additionally, in this particular situation, `1` is not just a integere, but *also a string*. The type checker may allow that, but if so, the later phases of the compiler must arrange it so that `1` is *actually converted* to a string (assuming that integers and strings are not represented uniformely).

SA remains tricky

A dream


Beskrivelse av språkets statiske semantikk
Sem-yacc
Semantisk sjekker for det aktuelle språk

However

- no standard description language
- no standard "theory"
    - part of SA may seem ad-hoc, more "art" than "engineering", complex
- *but*: well-established/well-founded (and non-ad-hoc) fields do exist
    - *type systems*, type checking
    - *data-flow* analysis . . . .

- in general
    - semantic "rules" must be individually specified and implemented per language
    - rules: defined based on trees (for AST): often straightforward to implement
    - clean language design includes *clean semantic rules*

When saying that there is no general standard theory, of course there would be the notion of *context-free grammars*, a class of grammars more expressive than context free grammars, while not yet as expressive as Turing machines (= full compuation power). Context-sensitive languages are well-defined, but as a formalism, it's too general, too unstructured to give much guiding light when it comes to concrete problems being analysed. Context-sensitive grammars as such are not on the pensum.

5.2 Attribute grammars

Attributes

Attribute

- a "property" or characteristic feature of something
- here: of language "constructs". More specific in this chapter:
- of syntactic elements, i.e., for non-terminal and terminal nodes in syntax trees

Static vs. dynamic

- distinction between **static** and *dynamic attributes*
- association attribute ↔ element: *binding*
- *static* attributes: possible to determine at/determined at compile time
- dynamic attributes: the others ...

With the concept of *attribute* so general, basically very many things can be subsumed under being an attribute of "something". After having a look at how attribute grammars are used for "attribution" (or "binding" of values of some attribute to a syntactic element), we will normally be concerned with more concrete attributes, like the *type* of something, or the *value* (and there are many other examples). In the very general use of the word "attribute" and "attribution" (the act of attributing something to something) is almost synonymous with "analysis" (here semantic analysis). The analysis is concerned with figuring out the value of some attribute one is interested in, for instance, the *type* of a syntactic construct. After having done so, the result of the analysis is typically *remembered* (as opposed to being calculated over and over again), but that's for efficiency reasons. One way of remembering attributes is in a specific data structure, for attributes of "symbols", that kind of data structure is known as the *symbol table.*

Examples in our context

- data *type* of a variable : static/dynamic
- *value* of an expression: dynamic (but seldomly static as well)
- *location* of a variable in memory: typically dynamic (but in old FORTRAN: static)
- *object-code*: static (but also: dynamic loading possible)

The *value* of an expression, as stated, is typically *not* a static "attribute" (for reasons which I hope are clear). Later, in this chapter, we will actually *use* values of expressions as attributes. That can be done, for instance, if there are *no* variables mentioned in the expressions. The values of those values typically are not known at compile-time and would not allow to calculate the value at compile time. However, having no variables is exactly the situation we will see later.

As a side remark: even with variables, *sometimes* the compiler *can* figure out, that, in some situations, the value of a variable *is* at some point is known in advance. In that case, an *optimization* could be to *precompute* the value and use that instead. To figure out whether or not that is the case is typically done via *data-flow analysis* which operates on *control-flow graph*. That is therefore not done via attribute grammars in general.

Attribute grammar in a nutshell

- AG: general formalism to bind "attributes to trees" (where trees are given by a CFG)[6]
- two potential ways to calculate "properties" of nodes in a tree:

"Synthesize" properties

define/calculate prop's *bottom-up*

"Inherit" properties

define/calculate prop's *top-down*

- allows both *at the same time*

Attribute grammar

CFG + **attributes** one grammar symbols + **rules** specifing for each production, how to determine attributes

- *evaluation* of attributes: requires some thought, more complex if mixing bottom-up + top-down dependencies

Example: evaluation of numerical expressions

Expression grammar (similar as seen before)

$$\begin{array}{rcl} exp & \rightarrow & exp + term \mid exp - term \mid term \\ term & \rightarrow & term * factor \mid factor \\ factor & \rightarrow & (\, exp\,) \mid \mathbf{n} \end{array}$$

- goal now: **evaluate** a given expression, i.e., the syntax tree of an expression, resp:

more concrete goal

Specify, in terms of the grammar, how expressions are evaluated

- grammar: describes the "format" or "shape" of (syntax) trees
- syntax-directedness
- value of (sub-)expressions: *attribute* here

As stated earlier: values of syntactic entities are generally *dynamic* attributes and cannot therefore be treated by an AG. In this simplistic AG example, it's statically doable (because no variables, no state-change etc.).

[6] Attributes in AG's: *static*, obviously.

Expression evaluation: how to do if on one's own?

- simple problem, easy solvable without having heard of AGs
- given an expression, in the form of a syntax tree
- evaluation:
  - simple *bottom-up* calculation of values
  - the value of a compound expression (parent node) **determined by the value of its subnodes**
  - realizable, for example, by a simple recursive procedure[7]

Connection to AG's

- AGs: basically a formalism to specify things like that
- *however*: general AGs will allow *more complex* calculations:
  - not just **bottom up** calculations like here but also
  - **top-down**, including both at the same time[8]

Pseudo code for evaluation

```
eval_exp(e) =
  case
  :: e equals PLUSnode ->
       return eval_exp(e.left) + eval_term(e.right)
  :: e equals MINUSnode ->
       return eval_exp(e.left) - eval_term(e.right)
  ...
  end case
```

AG for expression evaluation

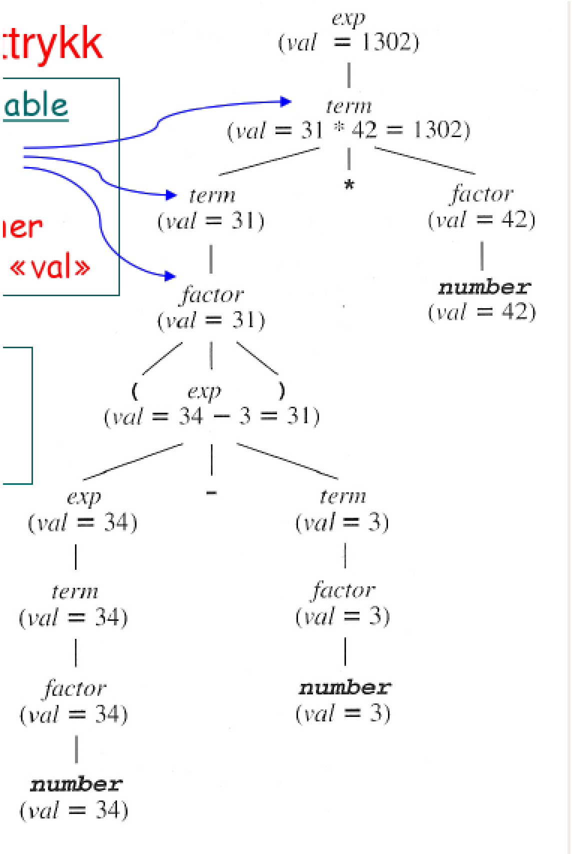
	productions/grammar rules			semantic rules
1	exp_1	→	$exp_2 + term$	exp_1 .val = exp_2 .val + $term$ .val
2	exp_1	→	$exp_2 - term$	exp_1 .val = exp_2 .val − $term$ .val
3	exp	→	$term$	exp .val = $term$ .val
4	$term_1$	→	$term_2 * factor$	$term_1$ .val = $term_2$ .val ∗ $factor$ .val
5	$term$	→	$factor$	$term$ .val = $factor$ .val
6	$factor$	→	**(** exp **)**	$factor$ .val = exp .val
7	$factor$	→	**n**	$factor$ .val = **n**.val

- *specific* for this example is:
  - only *one* attribute (for all nodes), in general: different ones possible
  - (related to that): only one semantic rule per production
  - as mentioned: rules here define values of attributes "bottom-up" only
- note: subscripts on the symbols for disambiguation (where needed)

[7] Resp. a number of mutually recursive procedures, one for factors, one for terms, etc. See the next slide.

[8] Top-down calculations will not be needed for the simple expression evaluation example.

Attributed parse tree



The attribute grammar (being purely synthesized = bottom-up) is very simple and hence, the values in the attribute `val` should be self-explanatory. It

Possible dependencies

Possible dependencies (>1 rule per production possible)

- parent attribute on *childen* attributes
- attribute in a node dependent on other attribute of the *same* node
- child attribute on *parent* attribute
- sibling attribute on *sibling* attribute
- *mixture* of all of the above at the same time
- but: **no** immediate dependence **across generations**

Attribute dependence graph

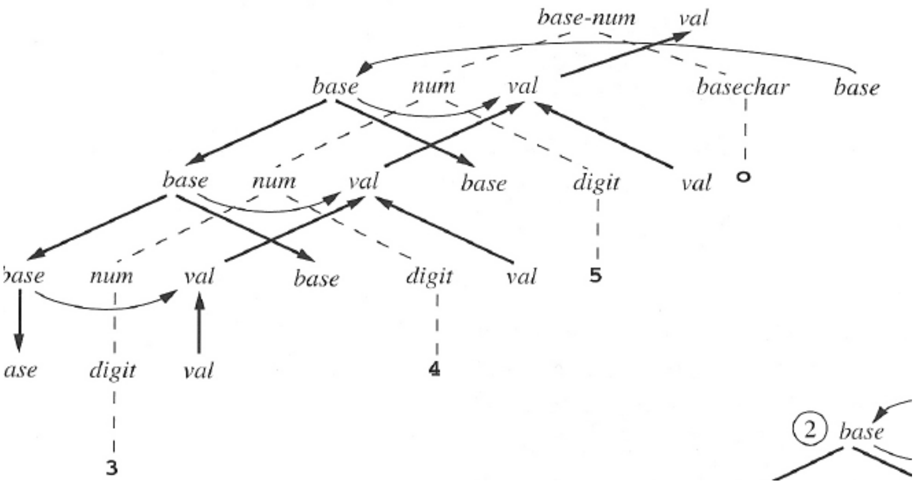
- dependencies ultimately between attributes in a syntax *tree* (instances) not between grammar symbols as such

⇒ attribute dependence graph (per syntax tree)

- complex dependencies possible:
  - evaluation complex

– invalid dependencies possible, if not careful (especially **cyclic**)

Sample dependence graph (for later example)



The graph belongs to an example we will revisit later. The dashed line represent the AST. The bold arrows the dependence graph. Later, we will classify the attributes in that `base` (at least for the non-terminals *num*) is inherited ("top-down"), whereas `val` is synthesized ("bottom-up").

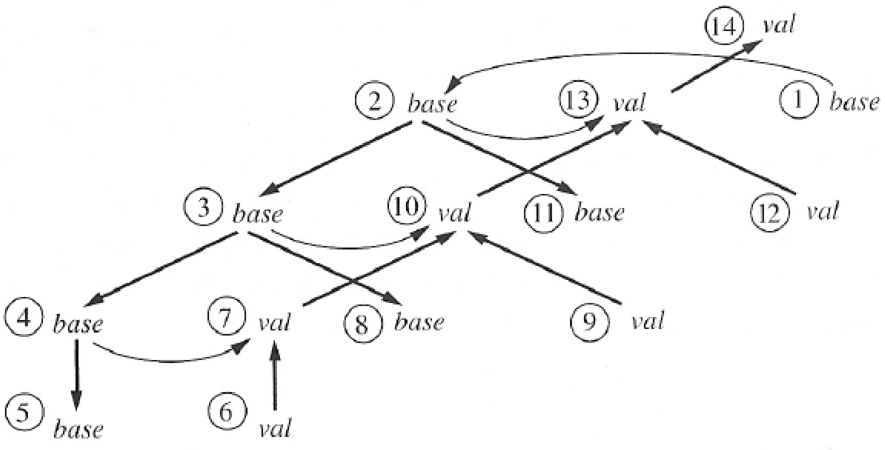
We will later have a look at what synthesized and inherited means. As we see in the example already here, being synthesized is (in its more general form) not as simplistic as "dependence only from attributes of children". In the example the synthesized attribute `val` depends on its inherited "sister attribute" `base` in most nodes. So, synthesized is not only "strictly bottom-up", it also goes "sideways" (from `base` to `val`). Now, this "sideways" dependence goes from inherited to synthesized only but never the other way around. That's fortunate, because in this way it's immediately clear that there are no *cycles* in the dependence graph. An evaluation (see later) following this form of dependence is **"down-up"**, i.e., first top-down, and afterwards bottom-up (but not then down again etc., the evaluation does not go into cycles).

Two-phase evaluation

Perhaps a too fine point concerning evaluation in the example. The above explanation highlighted that the evaluation is "phased" in first a top-down evaluation and afterwards a bottom-up phase. Conceptually, that is correct and gives a good intuition about the design of the dependencies of the attribute. Two "refinements" of that picture may be in order, though. First, as explained later, a dependence graph does not represent **one** possible evaluation (so it makes no real sense in speaking of "the" evaluation of the given graph, if we think of the edges as individual steps). The graph denotes which values need to be present *before* another value can be determined. Secondly, and relatd to that: If we take that view seriously, it's **not** strictly true that *all inherited depenencies are evaluated before all synthesized.* "Conceptually" they are, in a way, but there is an amount of "indepdendence" or "parallelism" possible. Looking at the following picture, which shows one of many possible evaluation orders shows, for example that step 8 is filling an inherited attribute, and that comes *after* 6 which deals with an synthesized one. But both steps are indepdedent, so they could as well be done the other way around.

So, the picture "first top-down, then bottom-up" is *conceptually correct* and a good intuition, it needs some fine-tuning when talking about when an indivdual step-by-step evaluation is done.

Possible evaluation order



The numbers in the picture give *one possible* evaluation order. As mentioned earlier, there is in general more than one possible way to evaluate dependency graph, in particular, when dealing with a syntax *tree*, and not with the generate case of a " syntax list" (considering lists as a degenerated form of trees). Generally, the rules that say when an AG is properly done assure that all possible evaluations give a unique value for all attributes, and the order of evaluation does not matter. Those conditions assure that each attribute instance gets a value *exactly once* (which also implies there are no cycles in the dependence graph).

Restricting dependencies

- general GAs allow bascially any kind of dependencies[9]
- complex/impossible to meaningfully evaluate (or understand)
- typically: restrictions, disallowing "mixtures" of dependencies
  - fine-grained: per attribute
  - or coarse-grained: for the whole attribute grammar

Synthesized attributes

bottom-up dependencies only (same-node dependency allowed).

Inherited attributes

top-down dependencies only (same-node and sibling dependencies allowed)

The classification in inherited = top-down and synthesized = bottom-up is a general guiding light. The discussion about the previous figures showed that there might be some refinements like that "sideways" dependencies are acceptable, not only strictly bottom-up dependencies.

[9] Apart from immediate cross-generation dependencies.

Synthesized attributes (simple)

Synthesized attribute

A **synthesized** attribute is defined wholly in terms of the node's *own* attributes, and those of its *children* (or constants).

Rule format for synth. attributes

For a **synthesized** attribute $\mathtt{s}$ of non-terminal A, *all* semantic rules with $A.\mathtt{s}$ on the left-hand side must be of the form

$$A.\mathtt{s} = f(X_1.\mathtt{b}_1, \ldots X_n.\mathtt{b}_k) \tag{5.1}$$

and where the semantic rule belongs to production $A \rightarrow X_1 \ldots X_n$

- Slight **simplification** in the formula.

The "simplification" here is that we ignore the fact that one symbol can have in general many attributes. So, we just write $X_1.b_1$ instead of $X_1.b_{1,1} \ldots X_1.b_{1.k_1}$ which would more "correctly" cover the situation in all generality, but doing so would not make the points more clear.

S-attributed grammar:

all attributes are synthesized

The simplification mentioned is to make the rules more readable, to avould all the subscript, while keeping the spirit. The simplification is that we consider only 1 attribute per symbol. In general, instead depend on $A.\mathtt{a}$ only, dependencies on $A.\mathtt{a}_1, \ldots A.\mathtt{a}_l$ possible. Similarly for the rest of the formula

Remarks on the definition of synthesized attributes

- Note the following aspects
  1. a synthesized attribute in a symbol: cannot *at the same time also* be "inherited".
  2. a synthesized attribute:
     - depends on attributes of children (and other attributes of the same node) only. However:
     - those attributes need *not* themselves be *synthesized* (see also next slide)
- in Louden:
  - he does not allow "intra-node" dependencies
  - he assumes (in his wordings): attributes are "globally unique"

Unfortunately, depending on the text-book the exact definitions (or the way it's formulated) of synthesized and inherited slightly deviate. But in spirit, of course, they all agree in principle. the lecture is not so much concerned with the super-fine print in definitions, more with questions like "given the following problem, write an AG", and the conceptual picture of synthesized (bottom-up and a bit of sideways), and inherited (top-down and perhaps a bit of sideways) helps in thinking about that problem. Of course, all books agree: *cycles* must be avoided and all attributes need to be uniquely defined. The concepts of synthesized and inherited attributes thereby helps to clarify thinking about those problems. For intance, by having this "phased" evaluation discussed earlier (first down with the inherited attributes, then up with the synthesized one) makes clear: there can't be a cycle.

Don't forget the purpose of the restriction

- ultimately: *calculate* values of the attributes
- thus: avoid **cyclic** dependencies
- one single synthesized attribute alone does not help much

S-attributed grammar

- restriction on the grammar, not just 1 attribute of one non-terminal
- simple form of grammar
- remember the expression evaluation example

S-attributed grammar:

all attributes are synthesized

Alternative, more complex variant

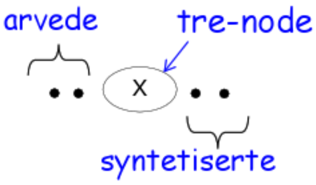
"Transitive" definition ($A \to X_1 \dots X_n$)

$$A.\mathtt{s} = f(A.\mathtt{i}_1, \dots, A.\mathtt{i}_m, X_1.\mathtt{s}_1, \dots X_n.\mathtt{s}_k)$$

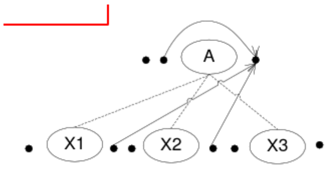
- in the rule: the $X_i.\mathtt{s}_j$'s synthesized, the $A_i.\mathtt{i}_j$'s inherited
- interpret the rule *carefully*: it says:
  - it's *allowed* to have synthesized & inherited attributes for A
  - it does **not** say: attributes in A *have to* be inherited
  - it says: in an A-node in the tree: a synthesized attribute
    - can depend on inherited att's in the same node and
    - on synthesized attributes of A-children-nodes

Pictorial representation

Conventional depiction



General synthesized attributes



Note that in the previous example discussing the dependence graph with attributes `base` and `val` was of this format and followed the convention: show the inherited `base` on the left, the synthesized `val` on the right.

Inherited attributes

- in *Louden's* simpler setting: inherited = non-synthesized

Inherited attribute

An **inherited** attribute is defined wholly in terms of the node's *own* attributes, and those of its *siblings* or its *parent* node (or constants).

Rule format

Rule format for inh. attributes

For an **inherited** attribute of a symbol X of X, *all* semantic rules mentioning $X.\mathtt{i}$ on the left-hand side must be of the form

$$X.\mathtt{i} = f(A.\mathtt{a}, X_1.\mathtt{b}_1, \dots, X, \dots X_n.\mathtt{b}_k)$$

and where the semantic rule belongs to production $A \to X_1 \dots X, \dots X_n$

- note: mentioning of "all rules", avoid conflicts.

Alternative definition ("transitive")

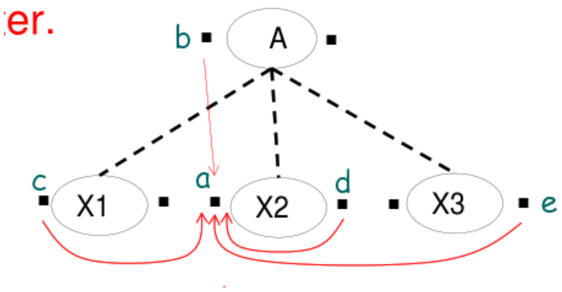
Rule format

For an **inherited** attribute `i` of a symbol X, *all* semantic rules mentioning $X.\mathtt{i}$ on the left-hand side must be of the form

$$X.\mathtt{i} = f(A.\mathtt{i}', X_1.\mathtt{b}_1, \dots, X.\mathtt{b}, \dots X_n.\mathtt{b}_k)$$

and where the semantic rule belongs to production $A \to X_1 \dots X \dots X_n$

- additional requirement: $A.\mathtt{i}'$ *inherited*
- rest of the attributes: inherited or synthesized



Simplistic example (normally done by the scanner)

CFG

$$
\begin{array}{rcl}
\textit{number} & \rightarrow & \textit{number}\,\textit{digit} \mid \textit{digit} \\
\textit{digit} & \rightarrow & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \mid
\end{array}
$$

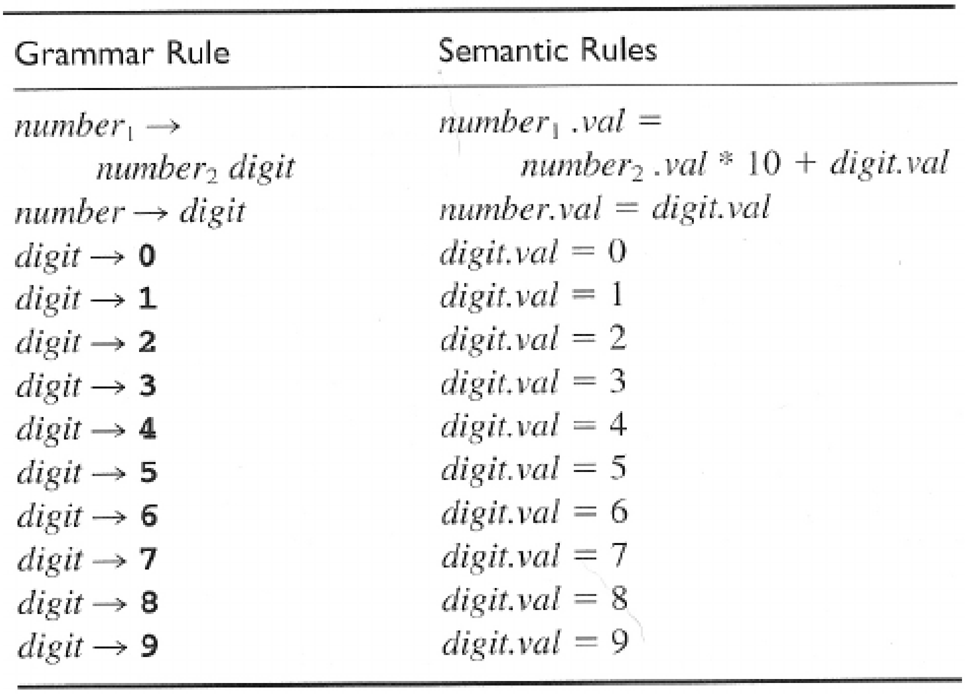
Attributes (just synthesized)

number	`val`
digit	`val`
terminals	[*none*]

We will look at an AG solution. In practice, this conversion is typically done by the scanner already, and the way it's normally done is relying on provide functions of the implementing programming language (all languages will support such conversion functions, either built-in or in some libraries). For instance in Java, one could use the method `valueOf(String s)`, for instance used as static method `Integer.valueOf("900")` of the class of integers. Obviously, not everything done by an AG can be done already by the scanner. But this particular example used as warm-up is so simple that it could be done by the scanner, and it typically is done there already.

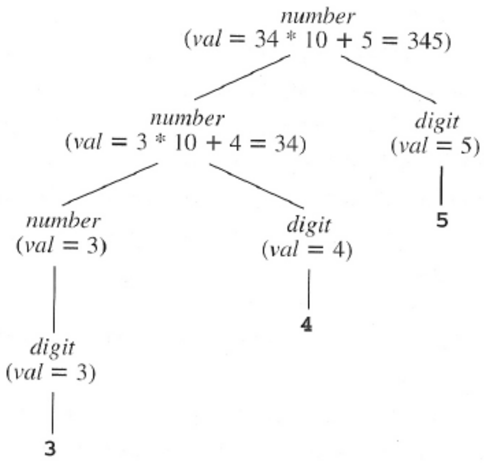
Numbers: Attribute grammar and attributed tree

A-grammar



Grammar Rule	Semantic Rules
$number_1 \rightarrow number_2\ digit$	$number_1.val = number_2.val * 10 + digit.val$
$number \rightarrow digit$	$number.val = digit.val$
$digit \rightarrow$ **0**	$digit.val = 0$
$digit \rightarrow$ **1**	$digit.val = 1$
$digit \rightarrow$ **2**	$digit.val = 2$
$digit \rightarrow$ **3**	$digit.val = 3$
$digit \rightarrow$ **4**	$digit.val = 4$
$digit \rightarrow$ **5**	$digit.val = 5$
$digit \rightarrow$ **6**	$digit.val = 6$
$digit \rightarrow$ **7**	$digit.val = 7$
$digit \rightarrow$ **8**	$digit.val = 8$
$digit \rightarrow$ **9**	$digit.val = 9$

attributed tree



Attribute evaluation: works on trees

i.e.: works equally well for

- *abstract syntax trees*

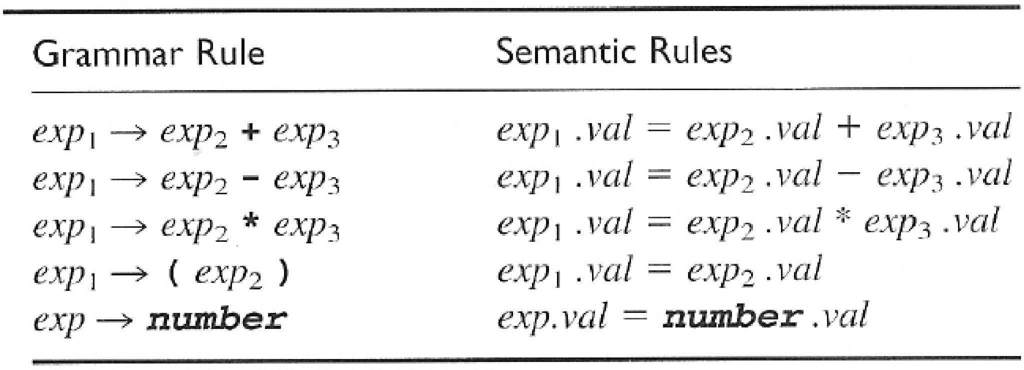
- *ambiguous* grammars

Seriously ambiguous expression grammar[10]

$$exp \rightarrow exp + exp \mid exp - exp \mid exp * exp \mid (\, exp\,) \mid \mathbf{n}$$

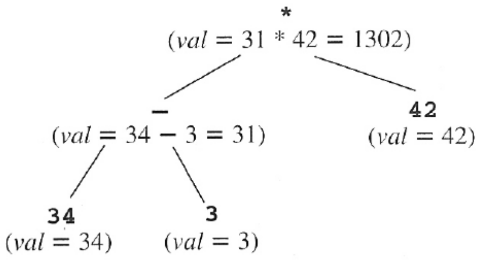
Evaluation: Attribute grammar and attributed tree

A-grammar



Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2 + exp_3$	$exp_1 .val = exp_2 .val + exp_3 .val$
$exp_1 \rightarrow exp_2 - exp_3$	$exp_1 .val = exp_2 .val - exp_3 .val$
$exp_1 \rightarrow exp_2 * exp_3$	$exp_1 .val = exp_2 .val * exp_3 .val$
$exp_1 \rightarrow (\ exp_2\ )$	$exp_1 .val = exp_2 .val$
$exp \rightarrow$ **number**	$exp.val =$ **number** $.val$

Attributed tree



Expressions: generating ASTs

Expression grammar with precedences & assoc.

$$\begin{array}{rcl} exp & \rightarrow & exp + term \mid exp - term \mid term \\ term & \rightarrow & term * factor \mid factor \\ factor & \rightarrow & (\, exp\,) \mid \mathbf{n} \end{array}$$

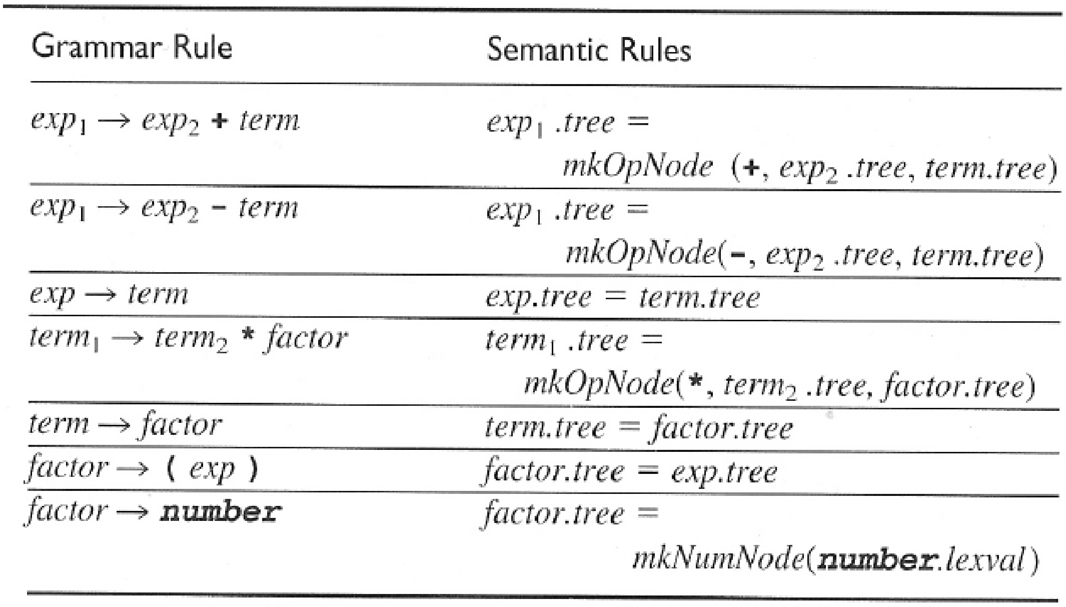
Attributes (just synthesized)

exp, *term*, *factor*	`tree`
n	`lexval`

[10] Alternatively: It's meant as grammar describing nice and clean ASTs for an underlying, potentially less nice grammar used for parsing.

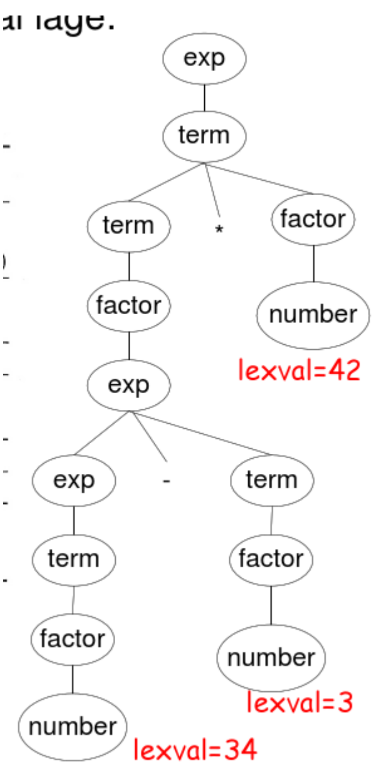
Expressions: Attribute grammar and attributed tree

A-grammar



Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2$ **+** $term$	$exp_1.tree =$ $mkOpNode\ (\mathbf{+}, exp_2.tree, term.tree)$
$exp_1 \rightarrow exp_2$ **-** $term$	$exp_1.tree =$ $mkOpNode(\mathbf{-}, exp_2.tree, term.tree)$
$exp \rightarrow term$	$exp.tree = term.tree$
$term_1 \rightarrow term_2$ ***** $factor$	$term_1.tree =$ $mkOpNode(\mathbf{*}, term_2.tree, factor.tree)$
$term \rightarrow factor$	$term.tree = factor.tree$
$factor \rightarrow$ **(** exp **)**	$factor.tree = exp.tree$
$factor \rightarrow$ `number`	$factor.tree =$ $mkNumNode($`number`$.lexval)$

A-tree



The AST looks a bit bloated. That's because the grammar was massaged in such a way that precedences and associativities during *parsing* are dealt with properly. The the grammar is describing more a parse tree rather than an AST, which often would be less verbose. But the AG formalisms itself does not care about what the grammar describes (a grammar used for parsing or a grammar describing the abstract syntax), it does especially not care if the grammar is ambiguous.

Example: type declarations for variable lists

CFG

$$\begin{array}{rcl}
\textit{decl} & \to & \textit{type}\ \textit{var-list} \\
\textit{type} & \to & \textbf{int} \\
\textit{type} & \to & \textbf{float} \\
\textit{var-list}_1 & \to & \textbf{id}, \textit{var-list}_2 \\
\textit{var-list} & \to & \textbf{id}
\end{array}$$

- Goal: attribute type information to the syntax tree
- *attribute*: `dtype` (with values *integer* and *real*)[11]
- complication: "top-down" information flow: type declared for a list of vars ⇒ **inherited** to the elements of the list

Types and variable lists: inherited attributes

grammar productions			semantic rules		
decl	→	*type var-list*	*var-list* .`dtype`	=	*type* .`dtype`
type	→	**int**	*type* .`dtype`	=	*integer*
type	→	**float**	*type* .`dtype`	=	*real*
$\textit{var-list}_1$	→	**id**, $\textit{var-list}_2$	**id**.`dtype`	=	$\textit{var-list}_1$.`dtype`
			$\textit{var-list}_2$.`dtype`	=	$\textit{var-list}_1$.`dtype`
var-list	→	**id**	**id**.`dtype`	=	*var-list* .`dtype`

- **inherited**: attribute for **id** and *var-list*
- but also *synthesized* use of attribute `dtype`: for *type* .`dtype`[12]

The dependencies are (especially for the variable lists) in such a way that the attribute of a later element depends on an ealier; in other words, the type information propagates from left to right through the "list". Seen as a tree, that means, the information propagates top-down in the tree. That can be seen in the next (quite small) example: the type information (there **float**) propagates down the right-branch of the tree, which corresponds to the list of two variables x and y.

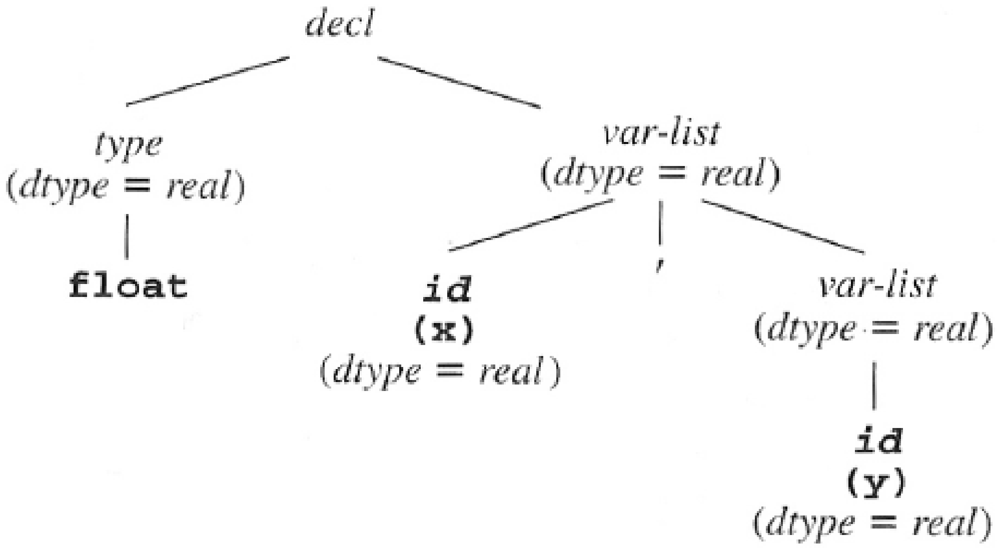
Types & var lists: after evaluating the semantic rules

float id(x),**id**(y)

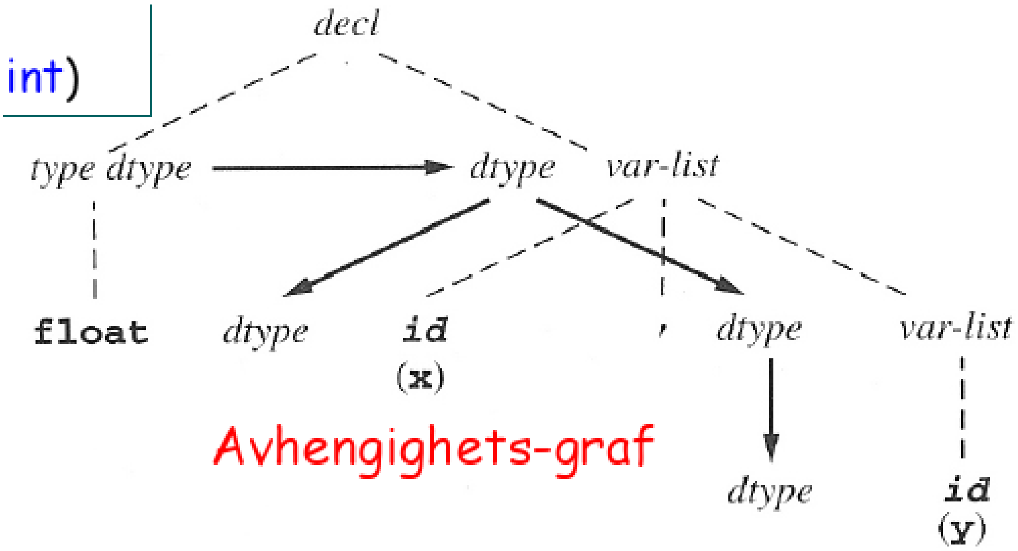
[11] There are thus 2 different attribute values. We don't mean "the attribute `dtype` has integer values", like $0, 1, 2, \ldots$

[12] Actually, it's conceptually better not to think of it as "the attribute `dtype`", it's better as "the attribute `dtype` of non-terminal *type*" (written *type* .`dtype`) etc. Note further: *type* .`dtype` is *not* yet what we called *instance* of an attribute.

Attributed parse tree



Dependence graph



Example: Based numbers (octal & decimal)

- remember: grammar for numbers (in decimal notation)
- evaluation: synthesized attributes
- now: *generalization* to numbers with decimal and octal notation

CFG

$$
\begin{array}{rcl}
\textit{based-num} & \rightarrow & \textit{num}\ \textit{base-char} \\
\textit{base-char} & \rightarrow & \mathbf{o} \\
\textit{base-char} & \rightarrow & \mathbf{d} \\
\textit{num} & \rightarrow & \textit{num}\ \textit{digit} \\
\textit{num} & \rightarrow & \textit{digit} \\
\textit{digit} & \rightarrow & \mathbf{0} \\
\textit{digit} & \rightarrow & \mathbf{1} \\
 & \dots & \\
\textit{digit} & \rightarrow & \mathbf{7} \\
\textit{digit} & \rightarrow & \mathbf{8} \\
\textit{digit} & \rightarrow & \mathbf{9}
\end{array}
$$

Based numbers: attributes

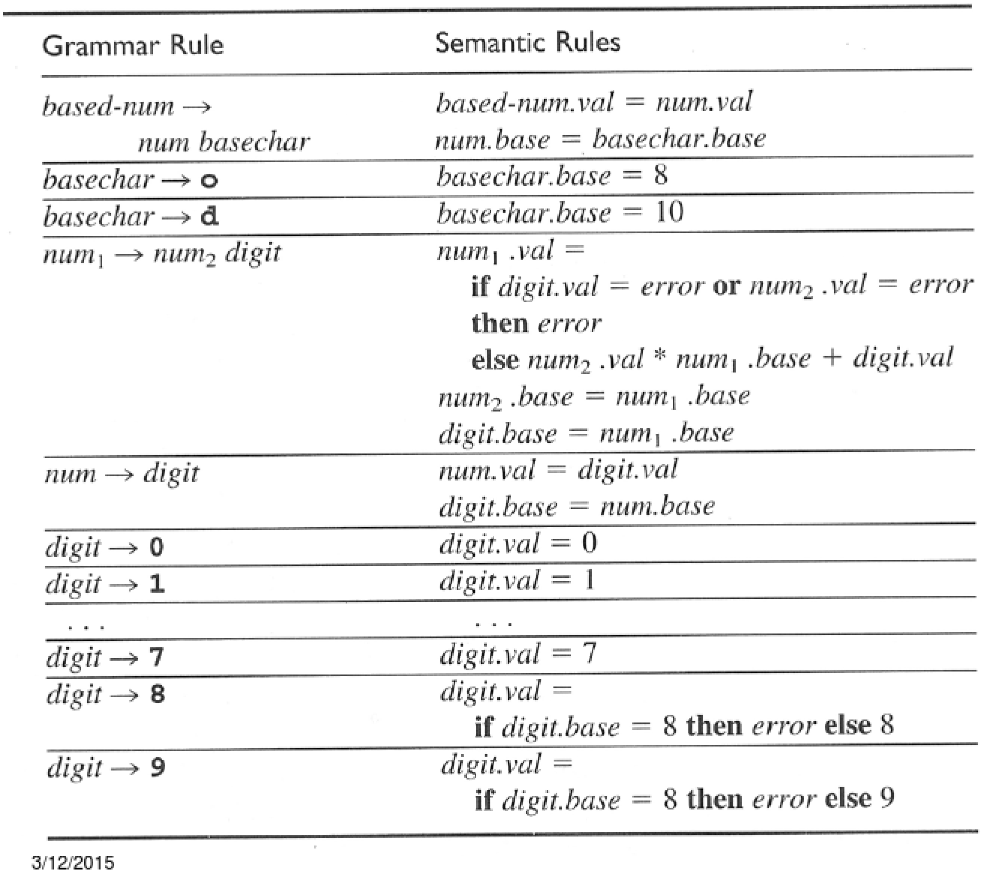
Attributes

- *based-num* .`val`: synthesized
- *base-char* .`base`: synthesized
- for *num*:
  - *num* .`val`: synthesized
  - *num* .`base`: **inherited**
- *digit* .`val`: synthesized

- **9** is not an octal character

⇒ attribute `val` may get value "*error*"!

Based numbers: a-grammar



Grammar Rule	Semantic Rules
$based\text{-}num \rightarrow$ $num\ basechar$	$based\text{-}num.val = num.val$ $num.base = basechar.base$
$basechar \rightarrow$ `o`	$basechar.base = 8$
$basechar \rightarrow$ `d`	$basechar.base = 10$
$num_1 \rightarrow num_2\ digit$	$num_1\ .val =$ **if** $digit.val = error$ **or** $num_2\ .val = error$ **then** $error$ **else** $num_2\ .val * num_1\ .base + digit.val$ $num_2\ .base = num_1\ .base$ $digit.base = num_1\ .base$
$num \rightarrow digit$	$num.val = digit.val$ $digit.base = num.base$
$digit \rightarrow$ **0**	$digit.val = 0$
$digit \rightarrow$ **1**	$digit.val = 1$
. . .	. . .
$digit \rightarrow$ **7**	$digit.val = 7$
$digit \rightarrow$ **8**	$digit.val =$ **if** $digit.base = 8$ **then** $error$ **else** 8
$digit \rightarrow$ **9**	$digit.val =$ **if** $digit.base = 8$ **then** $error$ **else** 9

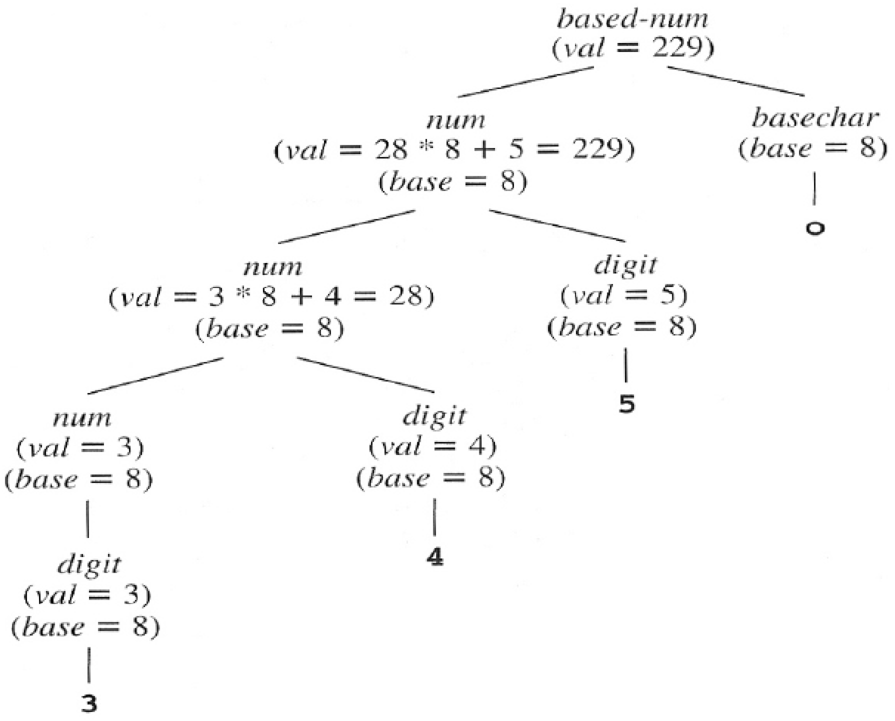
3/12/2015

The attribute grammar should rather be straightforward and the next slides will shed light on the dependencies and the evaluation. That illustrates the synthesized vs. the inherited parts perhaps more clearly than the equations of the semantic rules. As mentioned in the slides: the evaluation can lead to *errors* insofar that for base-8 numbers, the characters `8` and `9` are not allowed. Technically, to be a proper attribute grammar, a value need to be attached to each attribute instance for each tree. If we would take that serious, it required that we had to give back an "error" value, as can be seen in the code of the semantic rules. If we take that even more seriously, it would mean that the "type" of the `val` attribute is not just integers, but integers *or* an error value.

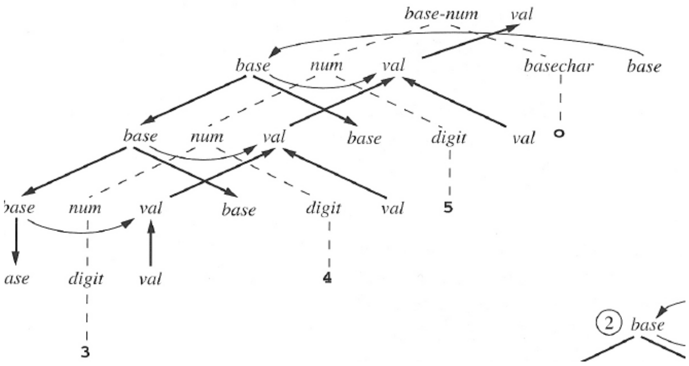
In a practical implementation, one would probably rather operate with *exceptions*, to achieve the same. Technically, an exception is not a ordinary *value* which is given back, but interrupts the standard control-flow as well. That kind of programming convenience is outside the (purely functional/equational) framework of AGs, and therefore, the given semantic rules deal the extra error value explicitly and evaluation propagate errors explicitly; since the errors occur during the "calculation phase", i.e., when dealing with the synthesized attribute, an error is propagated upwards the tree.

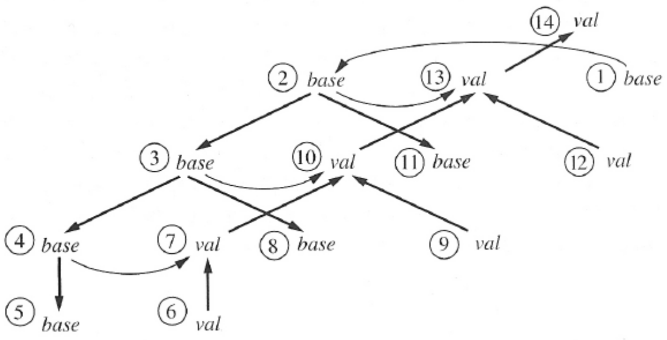
Based numbers: after eval of the semantic rules

Attributed syntax tree



Based nums: Dependence graph & possible evaluation order





Dependence graph & evaluation

- **evaluation order** must respect the edges in the *dependence graph*
- *cycles* must be avoided!
- directed acyclic graph (DAG)
- dependence graph ~ partial order
- *topological sorting*: turning a partial order to a total/linear order (which is consistent with the PO)
- *roots* in the dependence graph (**not** *the* root of the syntax tree): their values must come "from outside" (or constant)
- often (and sometimes required): terminals in the syntax tree:
  - terminals *synthesized* / *not inherited*
  - ⇒ terminals: *roots* of dependence graph
  - ⇒ get their value from the parser (token value)

A DAG is not a tree, but a generalization thereof. It may have more than one "root" (like a forest). Also: "shared descendents" are allowed. But no cycles.

As for the treatment of terminals, resp. restrictions some books require: An alternative view is that terminals get token values "from outside", the lexer. They are as if they were synthesized, except that it comes "from outside" the grammar.

Evaluation: parse tree method

For acyclic dependence graphs: possible "naive" approach

Parse tree method

Linearize the given partial order into a total order (topological sorting), and then simply evaluate the equations following that.

Rest

- works only if *all* dependence graphs of the AG are acyclic
- acyclicity of the dependence graphs?

  - decidable for given AG, but computationally expensive[13]
  - don't use general AGs but: restrict yourself to subclasses

- disadvantage of parse tree method: also not very efficient check per parse tree

Observation on the example: Is evalution (uniquely) possible?

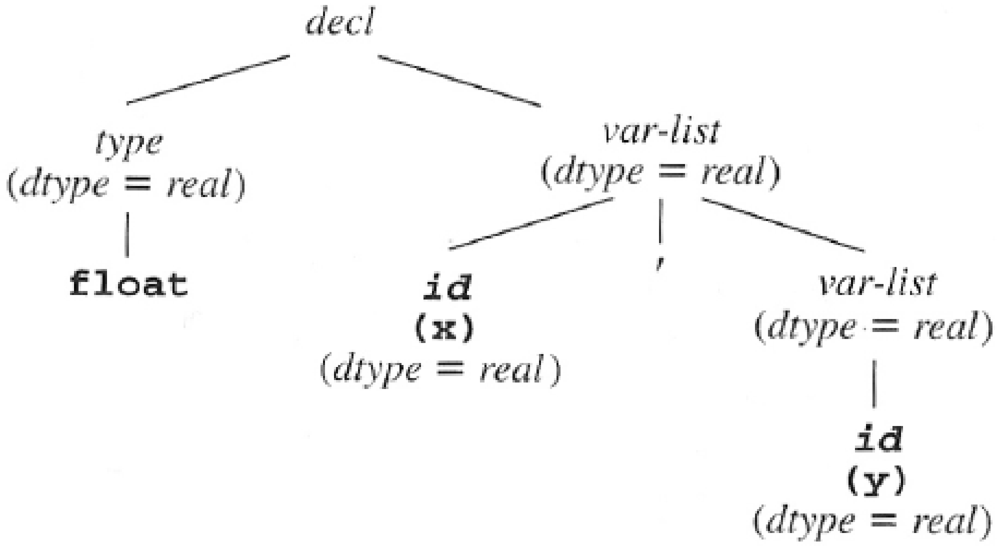
- all attributes: *either* inherited *or* synthesized[14]
- all attributes: must actually be *defined* (by some rule)
- guaranteed in that for every production:
  - all *synthesized* attributes (on the left) are defined
  - all *inherited* attributes (on the right) are defined
  - local loops forbidden
- since all attributes are either inherited or synthesized: each attribute in any parse tree: defined, and defined only *one* time (i.e., **uniquely defined**)

Loops

- loops intolerable for *evaluation*
- difficult to check (exponential complexity).[15]

Variable lists (repeated)

Attributed parse tree

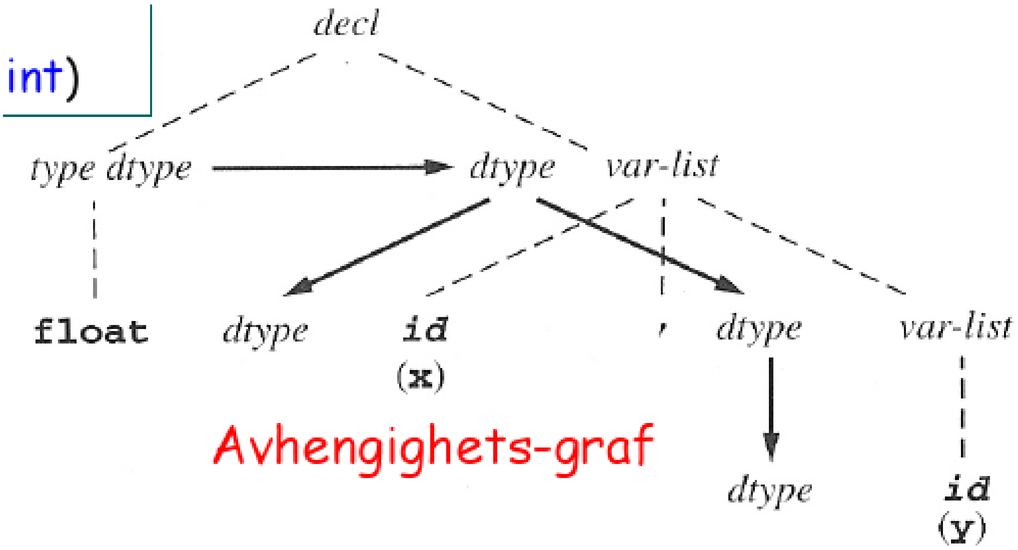


[13] On the other hand: the check needs to be done only once.

[14] *base-char* .base (synthesized) considered different from *num* .base (inherited)

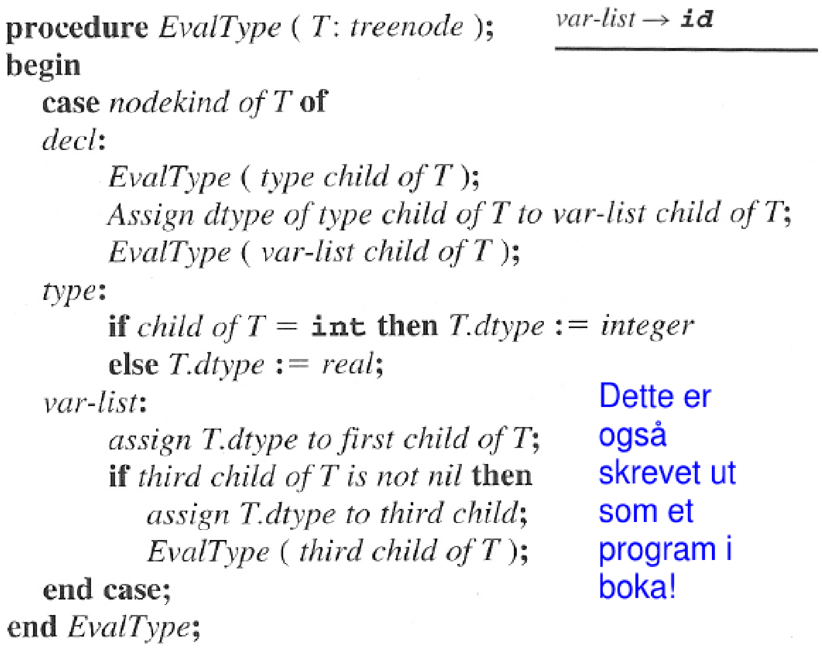
[15] acyclicity checking for a *given* dependence graph: not so hard (e.g., using topological sorting). Here: for *all* syntax trees.

Dependence graph



Typing for variable lists

- code assume: tree given



var-list → id

```
procedure EvalType ( T: treenode );
begin
   case nodekind of T of
   decl:
         EvalType ( type child of T );
         Assign dtype of type child of T to var-list child of T;
         EvalType ( var-list child of T );
   type:
         if child of T = int then T.dtype := integer
         else T.dtype := real;
   var-list:
         assign T.dtype to first child of T;
         if third child of T is not nil then
            assign T.dtype to third child;
            EvalType ( third child of T );
   end case;
end EvalType;
```

Dette er også skrevet ut som et program i boka!

The assumption that the tree is *given* is reasonable, if dealing with ASTs. For parse-tree, the attribution of types must deal with the fact that the parse tree is being built during parsing. It also means: it “blurs” typically the border between context-free and context-sensitive analysis.

L-attributed grammars

- goal: AG suitable for "on-the-fly" attribution
- all parsing works left-to-right.

L-attributed grammar

An attribute grammar for attributes $\mathtt{a}_1, \ldots, \mathtt{a}_k$ is *L-attributed*, if for each inherited attribute $\mathtt{a}_j$ and each grammar rule

$$X_0 \to X_1 X_2 \ldots X_n \ ,$$

the associated equations for $\mathtt{a}_j$ are all of the form

$$X_i.\mathtt{a}_j = f_{ij}(X_0.\vec{\mathtt{a}}, X_1.\vec{\mathtt{a}} \ldots X_{i-1}.\vec{\mathtt{a}}) \ .$$

where additionally for $X_0.\vec{\mathtt{a}}$, only *inherited* attributes are allowed.

Rest

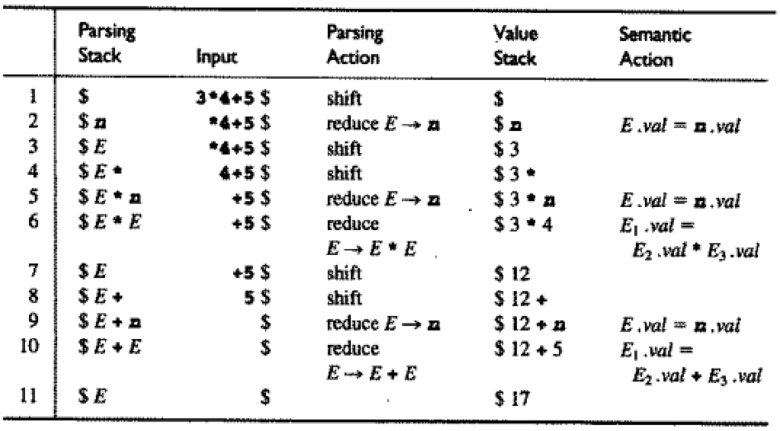
- $X.\vec{\mathtt{a}}$: short-hand for $X.\mathtt{a}_1 \ldots X.\mathtt{a}_k$
- Note S-attributed grammar ⇒ L-attributed grammar

Nowadays, doing it on-the-fly is perhaps not the most important design criterion.

"Attribution" and LR-parsing

- easy (and typical) case: synthesized attributes
- for *inherited* attributes
  - not quite so easy
  - perhaps better: *not* "on-the-fly", i.e.,
  - better *postponed* for later phase, when AST available.
- implementation: additional *value stack* for synthesized attributes, maintained "besides" the parse stack

Example: value stack for synth. attributes



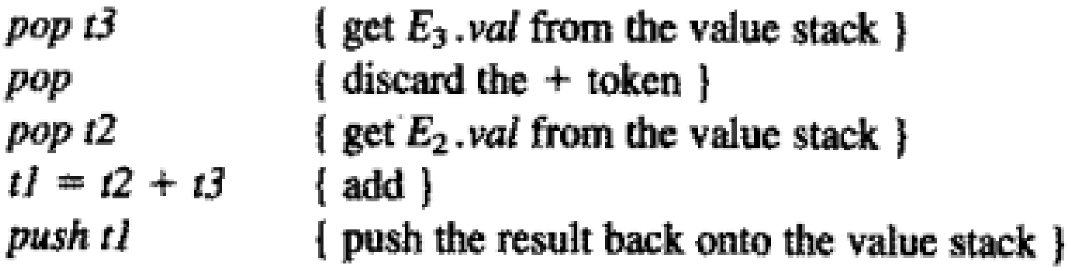
	Parsing Stack	Input	Parsing Action	Value Stack	Semantic Action
1	\$	3*4+5 \$	shift	\$	
2	\$ **n**	*4+5 \$	reduce $E \to$ **n**	\$ **n**	$E.val =$ **n**$.val$
3	\$ E	*4+5 \$	shift	\$ 3	
4	\$ E *	4+5 \$	shift	\$ 3 *	
5	\$ E * **n**	+5 \$	reduce $E \to$ **n**	\$ 3 * **n**	$E.val =$ **n**$.val$
6	\$ E * E	+5 \$	reduce $E \to E * E$	\$ 3 * 4	$E_1.val =$ $E_2.val * E_3.val$
7	\$ E	+5 \$	shift	\$ 12	
8	\$ E +	5 \$	shift	\$ 12 +	
9	\$ E + **n**	\$	reduce $E \to$ **n**	\$ 12 + **n**	$E.val =$ **n**$.val$
10	\$ E + E	\$	reduce $E \to E + E$	\$ 12 + 5	$E_1.val =$ $E_2.val + E_3.val$
11	\$ E	\$		\$ 17	

Sample action

```
E : E + E   { $$ = $1 + $3; }
```

in (classic) yacc notation

Value stack manipulation: that's what's going on behind the scene



```
pop t3          { get E₃.val from the value stack }
pop             { discard the + token }
pop t2          { get E₂.val from the value stack }
t1 = t2 + t3    { add }
push t1         { push the result back onto the value stack }
```

5.3 Signed binary numbers (SBN)

SBN grammar

$$\begin{array}{rcl} number & \to & sign\,list \\ sign & \to & + \mid - \\ list & \to & list\,bit \mid bit \\ bit & \to & \mathbf{0} \mid \mathbf{1} \end{array}$$

Intended attributes

symbol	attributes
number	`value`
sign	`negative`
list	`position`, `value`
bit	`position`, `value`

- here: attributes for non-terminals (in general: terminals can also be included)

5.4 Attribute grammar SBN

	production			attribution rules
1	*number*	→	*sign list*	$list.\mathtt{position} = 0$
				if $sign.\mathtt{negative}$
				then $number.\mathtt{value} = -LIST.\mathtt{value}$
				else $number.\mathtt{value} = LIST.\mathtt{value}$
2	*sign*	→	`+`	$sign.\mathtt{negative} = false$
3	*sign*	→	`-`	$sign.\mathtt{negative} = true$
4	*list*	→	*bit*	$bit.\mathtt{position} = list.\mathtt{position}$
				$list.\mathtt{value} = bit.\mathtt{value}$
5	$list_0$	→	$list_1\ bit$	$list_1.\mathtt{position} = list_0.\mathtt{position} + 1$
				$bit.\mathtt{position} = list_0.\mathtt{position}$
				$list_0.\mathtt{position} = list_1.\mathtt{value} + bit.\mathtt{value}$
6	*bit*	→	**0**	$bit.\mathtt{value} = 0$
7	*bit*	→	**1**	$bit.\mathtt{value} = 2^{bit.\mathtt{position}}$

Chapter
Symbol tables

What is it about?

Learning Targets of this Chapter

1. symbol table data structure
2. design and implementation choices
3. how to deal with scopes
4. connection to attribute grammars

Contents

6.1 Introduction

Symbol tables, in general

- **central** data structure
- "data base" or repository associating properties with "names" (identifiers, symbols)[1]
- **declarations**
  - constants
  - type declarationss
  - variable declarations
  - procedure declarations
  - class declarations
  - . . .
- *declaring* occurrences vs. *use* occurrences of names (e.g. variables)

- goal: associate attributes (properties) to syntactic elements (names/symbols)
- storing once calculated: (costs memory) ↔ recalculating on demand (costs time)
- most often: **storing** preferred
- but: can't one store it in the nodes of the *AST*?
  - remember: attribute grammar
  - however, fancy attribute grammars with many rules and complex synthesized/inherited attribute (whose evaluation traverses up and down and across the tree):
    - ∗ might be intransparent
    - ∗ storing info *in* the tree: might not be efficient

⇒ central repository (= **symbol table**) better

[1] Remember the (general) notion of "attribute".

So: do I need a symbol table?

In theory, alternatives exists; in practice, yes, symbol tables is the way to go; most compilers do use symbol tables.

Most often (and in our course), the symbol table is set up once, containing all the symbols that occur in a given program, and then, the semantic analyses (type checking, etc.) update the table accordingly. Implicit in that is that the symbol table is "static" (i.e., part of the static phase of the compiler). There are also some languages, which allow "manipulation" of symbol tables at *run time* (Racket is one (formally PLT scheme)).

In the slides, a point was made that basically every compiler has a symbol table (or even more than one). You find statements in the internet that symbol tables are not needed or even to be avoided. For instance, the stack overflow wisdom "no symbol tables in Go" claims that there are no symbol tables in Go (and in functional languages). It's not clear how reliable that information is, because here's a link `https://golang.org/pkg/debug/gosym/` to the official go implementation, referring to symbol tables.

6.2 Symbol table design and interface

Symbol table as abstract data type

- separate **interface** from implementation
- ST: "nothing else" than a lookup-table or *dictionary*
- associating "keys" with "values"
- here: keys = names (id's, symbols), values the attribute(s)

Schematic interface: two core functions (+ more)

- *insert*: add new binding
- *lookup*: retrieve

besides the core functionality:

- structure of (different?) *name spaces* in the implemented language, *scoping* rules
- typically: not one single "flat" namespace ⇒ typically not one big *flat* look-up table

⇒ influence on the design/interface of the ST (and indirectly the choice of implementation)

- necessary to "delete" or "hide" information (*delete*)

A symbol table is, typically, not just a "flat" dictionary, neither conceptually nor the way it's implemented. *Scoping* typically is something that often complicates the design of the symbol table.

It should also be clear from the context of discussion: when we speak of the *value* of an attribute we typically don't mean the semantic value of the symbol, like the integer value of an expression. The value of an attribute is meant in the "meta"-way, the value that the analysis attaches to the entity, for instance its type, its address, etc. (and only in rather rare cases, its programming language level value). The situation is the same as for attribute grammars and indeed, symbol tables can be seen as a data structure realizing "attributes". See also the next slide, contrasting two ways of attaching "attributes" to entities in a (syntax) tree: "internal", as part of the nodes, or external, in a separate repository (known as symbol table).

Two main philosophies

traditional table(s)

- central repository, separate from AST
- interface
  - $lookup(name)$,
  - $insert(name, decl)$,
  - $delete(name)$
- last 2: update ST for declarations *and* when entering/exiting *blocks*

decls. in the AST nodes

- do look-up ⇒ tree-*search*
- insert/delete: implicit, depending on relative positioning in the tree
- look-up:
  - efficiency?
  - however: optimizations exist, e.g. "redundant" extra table (similar to the traditional ST)

Here, for concreteness, *declarations* are the attributes stored in the ST. In general, it is not the only possible stored attribute. Also, there may be more than one ST.

Language often have different "name spaces". Even a relatively old-school language like C has 4 different name spaces for identifiers. There are different kinds of identifiers, and different rules (for instance wrt. scoping) apply to them. One way to arrange them could be to have different symbol tables, one specially for each name space. Later we will have also situation (but not caused by different kinds of identifiers), where the symbol table is arrange in a way that smaller symbol tables (per scope) are linked together where a symbol table of a "surrounding" scope points to a symbol table representing a scope nested deeper. One might see that as having "many" symbol tables, but maybe that's misleading. It's more an internal representation which a linked structure, but that data structure containing many individiual table is better seen conceptually as one symbol table (*the* symbol table of the language) but a complex behavior reflecting the lexical scoping of the language. Actually, whether or not one implments it in chaining up a bunch of individual hash table or similar structure or doing a different representation is a design choice one can make, both realizing the same external at the interface. In that spirit, also the remark that C has 4 different name spaces (which is true) and therefore maybe 4 symbol tables is a matter of how one seens (and implements) it: one may as well see and implement it as one symbol table (with 4 kinds of identifiers which are treated differently).

A cautionary note: You may find the statement that C (being old fashioned) does not feature name spaces. The discussion here was about the internal organization and scoping rules for identifers in C, which form internally 4 different name spaces. But C does not have elabore user-level mechanisms to introduce name spaces; therefore, one may stumble upon statements like "C does not support name spaces"...

6.3 Implementing symbol tables

Data structures to implement a symbol table

- different ways to implement *dictionaries* (or look-up tables etc.)
  - simple (association) lists

    - trees
        - balanced (AVL, B, red-black, binary-search trees)
    - **hash** tables, often method of choice
    - functional vs. imperative implementation
- careful choice influences efficiency
- influenced also by the language being implemented
- in particular, by its **scoping** rules (or the structure of the name space in general) etc.[2]

Nested block / lexical scope

for instance: *C*

```
{ int i; ... ; double d;
  void p(...);
  {
     int i;
     ...
  }
  int j;
  ...
```

more later

Blocks in other languages

TeX

```
\def\x{a}
{
  \def\x{b}
  \x
}
\x
\bye
```

LaTeX

```
\documentclass{article}
\newcommand{\x}{a}
\begin{document}
\x
{\renewcommand{\x}{b}
  \x
}
\x
\end{document}
```

But: static vs. dynamic binding (see later)

LaTeX and TeX are chosen for easy trying out the result oneself (assuming that most people have access to LaTeX and by implication, TeX). TeX is the underlying "core" on which LaTeX is put on

[2] Also the language used for implementation (and the availability of libraries therein) may play a role (but remember "bootstrapping")

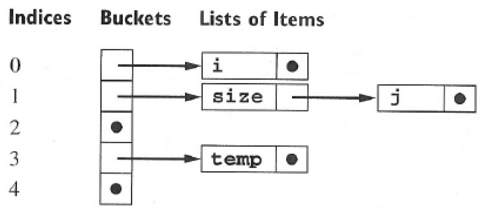
top. There are other formats in top of TeX (`texi` is another one; `texi` is involved, for instance, type setting the pdf version of the Compila language specification)

Hash tables

- classical and common implementation for STs
- "hash table":
  - generic term itself, different general forms of HTs exists
  - e.g. *separate chaining* vs. *open addressing*

There exists alternative terminology (cf. INF2220 in the older numbering scheme, it's the algo & data structures lecture), under which separate chaining is also known as *open hashing*. The *open addressing* methods are also called *closed hashing*. It's confusing, but that's how it is, and it's just words.

Separate chaining



Code snippet

```
{
  int temp;
  int j;
  real i;
  void size (....) {
    {
       ....
    }
  }
}
```

Block structures in programming languages

- almost no language has one global namespace (at least not for variables)
- pretty old concept, seriously started with ALGOL60

Block

- "region" in the program code
- delimited often by `{` and `}` or `BEGIN` and `END` or similar
- organizes the **scope** of declarations (i.e., the name space)
- can be **nested**

Block-structured scopes (in C)

```
int i, j;

int f(int size)
{ char i, temp;
  ...
  { double j;
    ..
  }
  ...
  { char * j;
    ...
  }
}
```

Nested procedures in Pascal

```
program Ex;
var i,j : integer

function f(size : integer) : integer;
var i, temp : char;
   procedure g;
   var j : real;
   begin
      ...
   end;
   procedure h;
   var j : ^char;
   begin
      ...
   end;

begin (* f's body *)
 ...
end;
begin  (* main program *)
   ...
end.
```

The Pascal-example shows a feature of Pascal, which is *not* supported by C, namely nested declarations of functions or procedures. As far as scoping and the discussion in the lecture is concerned, that's not a big issue: just that concerning names for variables, C and Pascal allow nested blocks, but for names representing functions or procedures, Pascal offers more freedom.

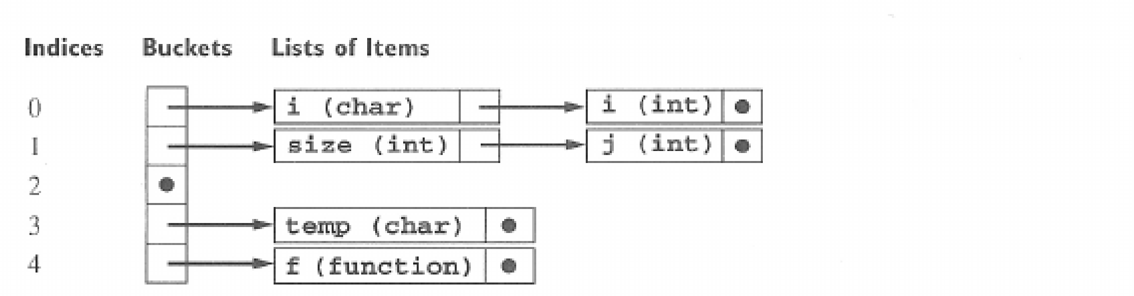
Block-strucured via stack-organized separate chaining

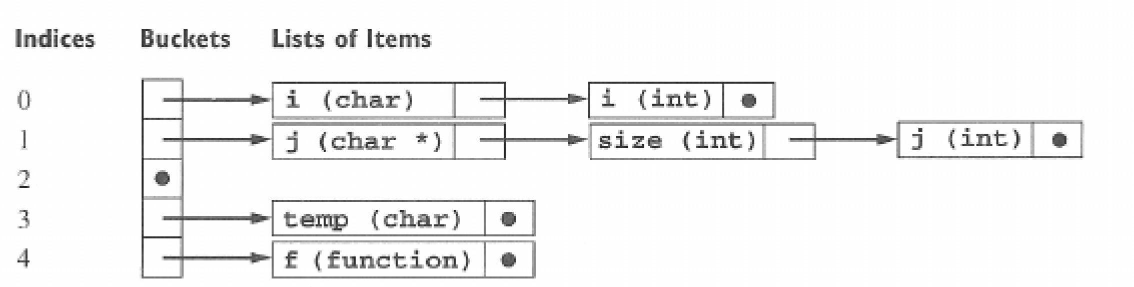
C code snippet

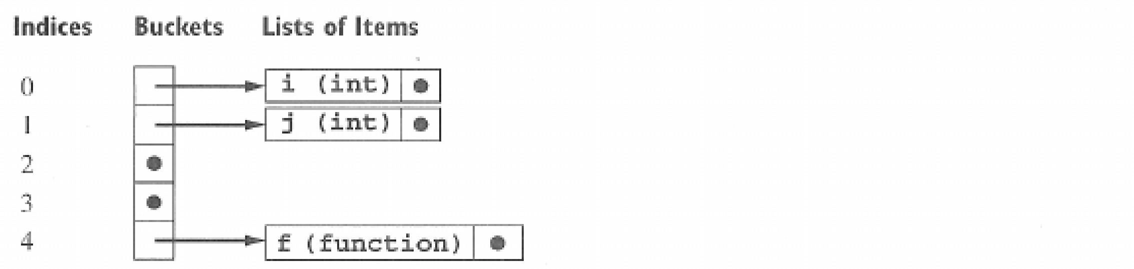
```
int i, j;

int f(int size)
```

```
{ char i , temp;
  ...
  { double j;
     ..
  }
  ...
  { char * j;
     ...
  }
}
```

“Evolution” of the hash table





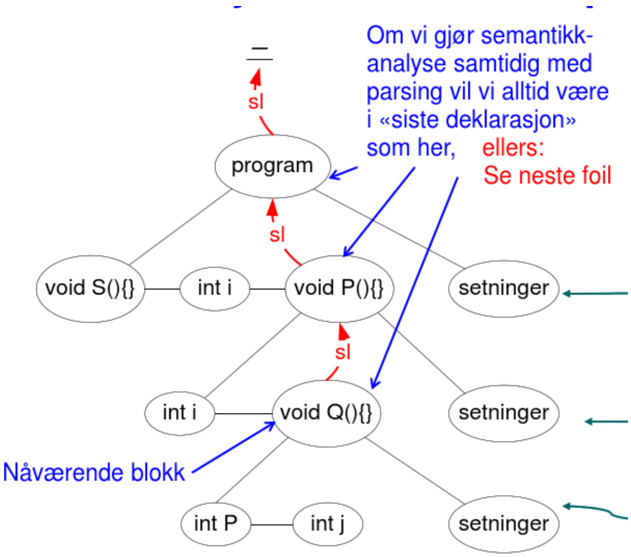


The 3 pictures (shown on the right-hand side of the slide version) correpond to three “points” inside the C program. The first one after entering the scope of function `f`. Inside the body of the function (immediately after entering), the two local variables are available, and of course also the formal parameter `temp`, which can be seen as a local variable, as well. At that point, the global variable `i` of type `int` is no longer “visible” or accessible, any reference to `i` will refer to the local variable `i` at that point.

Upon entering the first nested local scope, a second variable `j` is entered (making the global variable `j` unaccessible). That situation is *not* shown in the pictures. New, when *leaving* the mentioned scope, one way of dealing with the situation is that the additional second `j` of type `double` is *removed* from the hash-table again (shortering the corresponding linked chain). What is shown is a situation inside the *second* nested scope with another variable `j` (now a char pointer). Since the first nested local scope has been left at that point, the corresponding `j` "has become history", and the hash table of the third picture only contains the global `j` variable (which is unaccessible) and the now relevant second local `j` variable.

Using the syntax tree for lookup following (static links)

```
lookup (string n) {
   k = current, surrounding block
   do            // search for n in  decl for block k;
      k = k.sl   // one nesting level  up
   until found or k == none
}
```



Om vi gjør semantikk-analyse samtidig med parsing vil vi alltid være i «siste deklarasjon» som her, ellers: Se neste foil

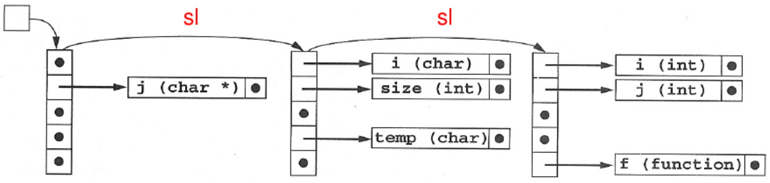
The notion of *static link* will be discussed later, in connection with the so-called run-time system and the run-time *stack*. There we go into more details, but the idea is the same as here: find a way to "locate" the relevant scope. If they are nested, connect them via some "parent pointer", and that pointer is known as static links (again, different names exists for that, unfortunately).

Alternative representation:

- arrangement different from 1 table with stack-organized external chaining
- each *block* with its **own** hash table.
- standard hashing within each block
- **static links** to link the block levels

⇒ "tree-of-hashtables"

- AKA: *sheaf-of-tables* or *chained symbol tables* representation



Note that the top-most scope is at the right-hand side of the table, and the static-link always points to the (uniquely determined) surrounding scope.

One may more generally say: one *symbol table* per block, as this form of organization can generally be done for symbol tables data structures (where hash tables is just one of many possible data structure to implement look-up tables).

6.4 Block-structure, scoping, binding, name-space organization

Block-structured scoping with chained symbol tables

- remember the *interface*
- look-up: following the static link (as seen)[3]
- **Enter** a block
  - create new (empty) symbol table
  - set static link from there to the “old” (= previously current) one
  - set the current block to the newly created one
- at **exit**
  - move the *current block* one level up
  - note: no *deletion* of bindings, just made *inaccessible*

Lexical scoping & beyond

- block-structured lexical scoping: **central** in programming languages (ever since ALGOL60 ...)
- but: other scoping mechanism exists (and exist side-by-side)
- example: C^{++}
  - member functions *declared* inside a class
  - *defined* outside
- still: method supposed to be able to access names defined in the *scope of the class* definition (i.e., other members, e.g. using `this`)

C^{++} class and member function

```
class A {
  ... int f(); ... // member function
}

A::f() {}     // def. of f ``in'' A
```

[3] The notion of static links will be encountered later again when dealing with *run-time* environments (and for analogous purposes: identfying scopes in “block-stuctured” languages).

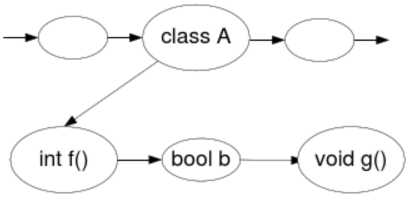
Java analogon

```
class A {
    int f() {...};
    boolean b;
    void h() {...};
}
```

Scope resolution in C^{++}

- class *name* introduces a **name for the scope**[4] (not only in C^{++})
- scope resolution operator `::`
- allows to explicitly refer to a “scope”’

- to implement
  - such flexibility,
  - also for *remote access* like `a.f()`
- declarations must be kept separately for each block (e.g. one hash table per class, record, etc., appropriately chained up)



Same-level declarations

Same level

```
typedef int i
int i;
```

- often forbidden (e.g. in C)
- *insert*: requires check (= *lookup*) first

Sequential vs. “collateral” declarations

1. Sequential in C

```
int i = 1;
void f(void)
  { int i = 2, j = i+1,
    ...
  }
```

[4] Besides that, class names themselves are subject to scoping themselves, of course ...

2. Collateral in ocaml/ML/Lisp

```
let i = 1;;
let i = 2 and y = i+1;;

print_int(y);;
```

I think the name “collateral” is unfortunate. A better word, in my eyes, would be *simultaneous* (or *parallel)*.

Recursive declarations/definitions

- for instance for functions/procedures
- also classes and their members

Direct recursion

```
int gcd(int n, int m) {
  if   (m == 0) return n;
  else return gcd(m,n % m);
}
```

Indirect recursion/mutual recursive def’s

```
void f(void) {
    ... g() ... }
void g(void) {
  ... f() ...}
```

Before treating the body, parser must add `gcd` into the symbol table (similar for the other example).

Mutual recursive definitions

```
void g(void);  /* function prototype decl. */

void f(void) {
    ... g() ... }
void g(void) {
  ... f() ...}
```

- different solutions possible
- Pascal: *forward declarations*
- or: treat all function definitions (within a block or similar) as mutually recursive
- or: special grouping syntax

Example syntax-es for mutual recursion

ocaml

```
let rec  f (x:int): int  =
    g(x+1)
and g(x:int) : int =
    f(x+1);;
```

Go

```
func f(x int) (int) {
        return g(x) +1
}

func g(x int) (int) {
        return f(x) -1
}
```

Static vs dynamic scope

- concentration so far on:
  - lexical scoping/block structure, static binding
  - some minor complications/adaptations (recursion, duplicate declarations, ...)
- **big** variation: **dynamic** binding / **dynamic scope**
- for variables: *static* binding/ *lexical scoping* the norm
- however: cf. late-bound methods in OO

Static scoping in C

Code snippet

```
#include <stdio.h>

int i = 1;
void f(void) {
  printf("%d\n",i);
}


void main(void) {
  int  i = 2;
  f();
  return 0;
}
```

which value of `i` is printed then?

Dynamic binding example

```
 1  void Y () {
 2    int i;
 3    void P() {
 4      int i;
 5      ...;
 6      Q();
 7    }
 8    void Q(){
 9      ...;
10      i = 5;   // which i is meant?
11    }
12    ...;
13
14    P();
15    ...;
16  }
```

for dynamic binding: the one from line 4

Static or dynamic?

TeX

```
\def\astring{a1}
\def\x{\astring}
\x
{
  \def\astring{a2}
  \x
}
\x
\bye
```

LaTeX

```
\documentclass{article}
\newcommand{\astring}{a1}
\newcommand{\x}{\astring}
\begin{document}
\x
{
  \renewcommand{\astring}{a2}
  \x
}
\x
\end{document}
```

emacs lisp (not Scheme)

```
(setq astring "a1")  ;; ``assignment''
(defun x() astring)  ;; define ``variable x''
(x)                  ;; read value
(let ((astring "a2"))
      (x))
```

Again, it's very easy to check by invoking TEX or LATEX, or firing off emacs and evaluate the lisp snippet in a buffer, for instance.

Code first

```
package main
import ("fmt")

var f = func ()   {
  var x = 0
  var g = func() {fmt.Printf(" x = %v", x)}
  x = x + 1
    {
      var x = 40                  // local variable
      g()
      fmt.Printf(" x = %v", x)}
}
func main() {
  f()
}
```

Static binding is not about "value"

- the "static" in static binding is about
  - binding to the declaration / memory location,
  - not about the *value*
- nested functions used in the example (Go)
- `g` declared inside `f`

```
package main
import ("fmt")

var f = func ()   {
  var x = 0
  var g = func() {fmt.Printf(" x = %v", x)}
  x = x + 1
    {
      var x = 40                  // local variable
      g()
      fmt.Printf(" x = %v", x)}
}
func main() {
  f()
}
```

Static binding can become tricky

```
package main
import ("fmt")

var f = func () (func (int) int)  {
        var x = 40                  // local variable
        var g = func (y int) int { // nested function
                return x + 1
        }
        x = x+1                     // update x
        return g                    // function as return value
}

func main() {
        var x = 0
        var h = f()
        fmt.Println(x)
        var r = h (1)
        fmt.Printf(" r = %v", r)
}
```

- example uses *higher-order* functions

As said, the example uses higher-order functions. In particular, the function `f` gives back some function, namely the function `g`, and not only that: function `g` is defined *inside* `f`, in particular, `g` is defined inside the scope of `f`. And finally, the nested function `g` refers to `x`, which is also defined inside `f`. Now the problem is that the scope of `f` lives longer than the body of `f` itself. We come to that problem also later, when dealing with run-time environment. In many languages, one important part of the RTE is the run-time stack, or call stack. It turns out, that in situations like the ones illustrated here, a stack is no longer good enough for providing lexical scoping.

6.5 Symbol tables as attributes in an AG

Nested lets in ocaml

```
let x = 2 and y = 3 in
  (let x = x+2 and y =
       (let z = 4 in x+y+z)
   in print_int (x+y))
```

- simple grammar (using **,** for "collateral" = simultaneous declarations)

$$\begin{array}{rcl}
S & \rightarrow & \textit{exp} \\
\textit{exp} & \rightarrow & \textbf{(}\,\textit{exp}\,\textbf{)} \mid \textit{exp} + \textit{exp} \mid \textbf{id} \mid \textit{num} \mid \textbf{let}\ \textit{dec-list}\ \textbf{in}\ \textit{exp} \\
\textit{dec-list} & \rightarrow & \textit{dec-list}\,\textbf{,}\ \textit{decl} \mid \textit{decl} \\
\textit{decl} & \rightarrow & \textbf{id} = \textit{exp}
\end{array}$$

1. no identical names in the same let-block
2. used names must be declared
3. most-closely nested binding counts
4. sequential (non-simultaneous) declaration (≠ ocaml/ML/Haskell . . .)

```
let x = 2, x = 3 in x + 1        (* no, duplicate  *)

let x = 2  in x+y                (* no, y unbound *)

let x = 2 in (let x = 3 in x)    (* decl. with 3 counts *)

let x = 2, y = x+1               (* one after the other *)
in   (let x = x+y,
          y = x+y
      in y)
```

Goal

Design an *attribute grammar* (using a *symbol table*) specifying those rules. Focus on: error attribute.

Attributes and ST interface

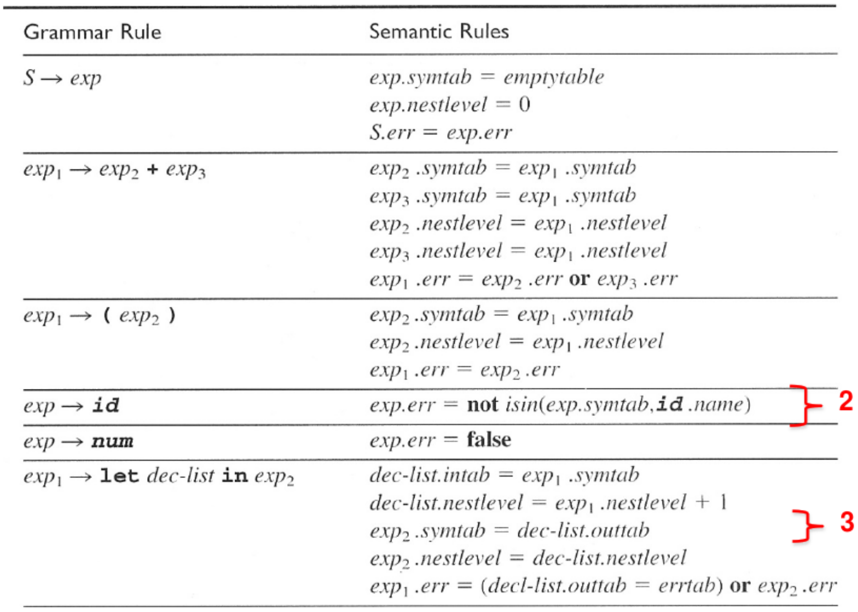
symbol	attributes	kind
exp	`symtab`	inherited
	`nestlevel`	inherited
	err	synthesis
dec-list, *decl*	`intab`	inherited
	`outtab`	synthesized
	`nestlevel`	inherited
id	`name`	injected by scanner

Symbol table functions

- `insert(tab,name,lev)`: returns a new table
- `isin(tab,name)`: boolean check
- `lookup(tab,name)`: gives back *level*
- `emptytable`: you have to start somewhere
- `errtab`: error from declaration (but not stored as attribute)

As for the information stored and especially for the look-up function: Realistically, more info would be stored, as well, for instance types etc.

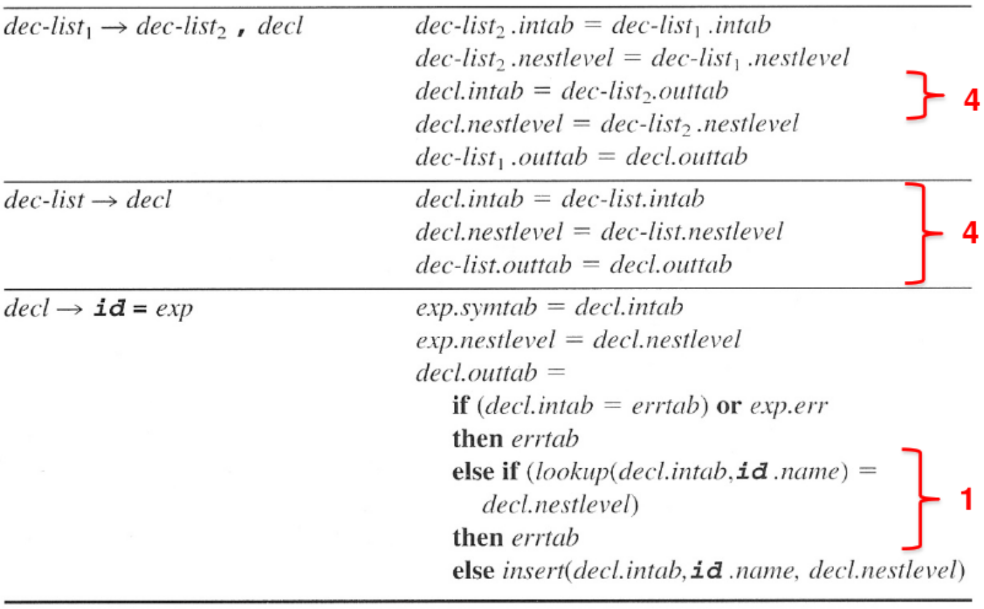
Attribute grammar (1): expressions



Grammar Rule	Semantic Rules	
$S \rightarrow exp$	$exp.symtab = emptytable$ $exp.nestlevel = 0$ $S.err = exp.err$	
$exp_1 \rightarrow exp_2$ **+** exp_3	$exp_2.symtab = exp_1.symtab$ $exp_3.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_3.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$ **or** $exp_3.err$	
$exp_1 \rightarrow$ **(** exp_2 **)**	$exp_2.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$	
$exp \rightarrow$ **id**	$exp.err =$ **not** $isin(exp.symtab,$**id**$.name)$	2
$exp \rightarrow$ **num**	$exp.err =$ **false**	
$exp_1 \rightarrow$ **let** *dec-list* **in** exp_2	$dec\text{-}list.intab = exp_1.symtab$ $dec\text{-}list.nestlevel = exp_1.nestlevel + 1$ $exp_2.symtab = dec\text{-}list.outtab$ $exp_2.nestlevel = dec\text{-}list.nestlevel$ $exp_1.err = (decl\text{-}list.outtab = errtab)$ **or** $exp_2.err$	3

- note: expressions in let's can introduce scopes themselves!
- interpretation of nesting level: expressions vs. declarations[5]

Attribute grammar (2): declarations



$dec\text{-}list_1 \rightarrow dec\text{-}list_2$ **,** $decl$	$dec\text{-}list_2.intab = dec\text{-}list_1.intab$ $dec\text{-}list_2.nestlevel = dec\text{-}list_1.nestlevel$ $decl.intab = dec\text{-}list_2.outtab$ $decl.nestlevel = dec\text{-}list_2.nestlevel$ $dec\text{-}list_1.outtab = decl.outtab$
$dec\text{-}list \rightarrow decl$	$decl.intab = dec\text{-}list.intab$ $decl.nestlevel = dec\text{-}list.nestlevel$ $dec\text{-}list.outtab = decl.outtab$
$decl \rightarrow$ **`id`** $=$ exp	$exp.symtab = decl.intab$ $exp.nestlevel = decl.nestlevel$ $decl.outtab =$ **if** ($decl.intab = errtab$) **or** $exp.err$ **then** $errtab$ **else if** ($lookup(decl.intab,$**`id`**$.name) =$ $decl.nestlevel$) **then** $errtab$ **else** $insert(decl.intab,$**`id`**$.name, decl.nestlevel)$

Final remarks concerning symbol tables

- *strings* as symbols i.e., as keys in the ST: might be improved
- name spaces can get complex in modern languages,
- more than one "hierarchy"
  - lexical blocks
  - inheritance or similar
  - (nested) modules
- not all bindings (of course) can be solved at compile time: *dynamic binding*
- can e.g. variables and types have same name (and still be distinguished)
- *overloading* (see next slide)

Final remarks: name resolution via overloading

- corresponds to "in abuse of notation" in textbooks
- disambiguation not by name, but differently especially by "argument types" etc.
- variants :
  - method or function overloading
  - operator overloading
  - user defined?

[5] I would not have recommended doing it like that (though it works)

```
i + j    // integer addition
r + s    // real-addition

void f(int i)
void f(int i,int j)
void f(double r)
```

Chapter
Types and type checking

What is it about?

Learning Targets of this Chapter

1. the concept of types
2. specific common types
3. type safety
4. type checking
5. polymorphism, subtyping and other complications

Contents

7.1 Introduction

This chapter deals with “types”. Since the material is presented as part of the static analysis (or semantic analysis) phase of the compiler, we are dealing mostly with *static* aspects of types (i.e., static typing).

The notion of “type” is **very** broad and has many different aspects. The study of “types” is a research field in itself (“type theory”). In some way, types and type checking is the very essence of semantic analysis, insofar that types can be very “expressive” and can be used to represent vastly many different aspects of the behavior of a program. By “more expressive” I mean types that express much more complex properties or attributes than the ones standard programmers are familiar with: booleans, integers, structured types, etc. When increasing the “expressivity”, types might not only capture more complex situations (like types for higher-order functions), but also unusual aspects, not normally connected with types, like for instance: bounds on memory usage, guarantees of termination, assertions about secure information flow (like no information leakage), and many more.

As a final random example: a language like *Rust* is known for its non-standard form of memory management based on the notion of *ownership* to a piece of data. Ownership tells who has the right to access the data when and how, and that’s important to know as as simultaneous write access leads to trouble. Regulating ownership can and has been formulated by corresponding “ownership type systems” where the type expresses properties concerning ownership.

That should give a feeling that, with the notion of types such general, the situation is a bit as with “attributes” and attribute grammars: “everything” may be an attribute since an attribute is nothing else than a “property”. The same holds for types. With a loose interpretation like that, types may represent basically all kinds of concepts: like, when interested in property “A”, let’s intoduce the notion of “A”-types (with “A” standing for memory consumption, ownership, and what not). But still: studying type systems and their expressivity and application to programming languages seems a much *broader* and *deeper* (and more practical) field than the study of attribute grammars. By more practical, I mean: while attribute grammars certainly have useful applications, stretching them to new "non-standard" applications may be possible, but it's, well, stretching it.[1] Type systems, on the other hand, span more easily form very simple and practical usages to very expressive and foundational logical system.

In this lecture, we keep it more grounded and mostly deal with concrete, standard (i.e., not very esoteric) types. Simple or "complicated" types, there are at least two aspects of a type. One is, what a user or programmer sees or is exposed to. The second one is the inside view of the compiler writer. The user may be informed that it's allowed to write `x + y` where `x` and `y` are both integers (carrying the type `int`), or both strings, in which case + represents string addition. Or perhaps the language even allows that one variable contains a string and the other an integer, in which case the + is still string concatenation, where the integer valued operand has to be converted to its string representation. The compiler writer needs then to find representations in memory for those data types (ultimately in binary form) that actually *realize* the operations described above on an abstract level. That means choosing an appropriate encoding, choosing the right amount of memory (long ints need more space than short ints, etc, perhaps even depending on the platform), and making sure that needed conversions (like from integers to string) actually are done in the compiled code (most likely arranged statically). Of course, the programmer does not want to know those details, he typically could not care less, for instance, whether the machine architecture is "little-endian" or "big-endian" (see `https://en.wikipedia.org/wiki/Endianness`). But the compiler writer will have to care when writing the compiler itself to *represent* or *encode* what the programmer calls "an integer" or "a string". So, apart from the more esoteric and advanced roles types play in programming languages, perhaps the most fundamental role is that of **abstraction**: to shield the programmer from the dirty details of the actual representation.

> Types are a central abstraction for programmers.

Abstraction in the sense of hiding underlying representional details.[2]

The lecture will have some look at both aspects of type systems. One is the representational aspect. That one is more felt in languages like C, which is closer to the operating system and to memory in hardware than languages that came later. Besides that, we will also more look at type system as *specification* of what is allowed at the programmer's level ("is it allowed to do a + on an a value of `integer` type and of `string` type?"), i.e., how to specify a type system in a programming language independent from the question how to choose proper lower-level encodings that the abstraction specified in the type system.

General remarks and overview

- Goal here:
  - what are *types*?
  - static vs. dynamic typing
  - how to describe types *syntactically?*
  - how to *represent* and use types in a compiler?
- coverage of various types
  - basic types (often predefined/built-in)
  - type constructors

[1] That's at least my slightly biased opinion.

[2] Beside that practical representational aspect, types are also an abstraction in the sense that they can be viewed as the "set" of all the values of that given type. Like `int` represents the set of all integers. Both views are consistent as all members of the "set" `int` are consistently represented in memory and consistently treated by functions operating on them. That "consistency" allows us as programmers to think of them as integers, and forget about details of their representation, and it's the task of the compiler writer, to reconcile those two views: *the low-level encoding must maintain the high-level abstraction.*

  - values of a type
  - type operators
  - representation at run-time
  - run-time tests and special problems (array, union, record, pointers)
- specification and implementation of type systems/type checkers
- advanced concepts

Why types?

- crucial, user-visible **abstraction** describing program behavior
- one view: type describes a set of (mostly related) *values*
- static typing: checking/enforcing a type discipline at compile time
- dynamic typing: same at run-time, mixtures possible
- completely untyped languages: very rare, types were part of PLs from the start.

Milner's dictum ("type safety")

Well-typed programs cannot go wrong!

- *strong* typing:[3] rigorously prevent "misuse" of data
- types useful for later phases and optimizations
- documentation and partial specification

In contrast to (standard) types: many other abstractions in SA (like the control-flow graph or data flow analysis and others) are not directly visible in the source code. However, in the light of the introductory remarks that "types" can capture a very broad spektrum of semantic properties of a language if one just makes the notion of type general enough ("ownership", "memory consumption"), it should come as no surprise that one can capture *data flow* in appropriately complex type systems, as well...

Besides that: there are not really any *truly* untyped languages around, there is always some discipline (beyond syntax) on what a programmer is allowed to do and what not. Probably the anarchistic recipe of "anything (syntactically correct) goes" tends to lead to disaster. Note that "dynamically typed" or "weakly typed" is not the same as "untyped".

Types: in first approximation

Conceptually

- semantic view: set of values *plus* a set of corresponding operations
- syntactic view: notation to *construct* basic elements of the type (its values) *plus* "procedures" operating on them
- compiler implementor's view: data of the same type have same underlying memory representation

further classification:

- built-in/predefined vs. *user-defined* types
- basic/base/elementary/primitive types vs. compound types
- type constructors: building more compex types from simpler ones
- reference vs. value types

[3] Terminology rather fuzzy, and perhaps changed a bit over time.

7.2 Various types and their representation

Some typical base types

base types			
int	0, 1, ...	+, −, *, /	integers
real	5.05E4 ...	+, −, *	real numbers
bool	true, false	and or (|) ...	booleans
char	'a'		characters
⋮			

- often HW support for some of those (including some of the op's)
- mostly: elements of int are not exactly mathematical *integers*, same for real
- often variations offered: int32, int64
- often implicit *conversions* and relations between basic types
  - which the type system has to specify/check for legality
  - which the compiler has to implement

Some compound types

compound types		
array[0..9] of real		a[i+1]
list	[], [1;2;3]	concat
string	"text"	concat ...
struct / record		r.x
...		

- mostly reference types
- when built in, special "easy syntax" (same for basic built-in types)
  - 4 + 5 as opposed to plus(4,5)
  - a[6] as opposed to array_access(a, 6) ...
- parser/lexer aware of built-in types/operators (special precedences, associativity, etc.)
- cf. functionality "built-in/predefined" via libraries

Being a "conceptual" view means, it's about the "interface", it's an abstract view of how one can make *use* of members of a type. It not about implementation details, like "integers are 2 byte words in such-and-such representation". See also the notion of abstract data type on the next slide.

Abstract data types

- unit of *data* together with *functions/procedures/operations* ... operating on them
- encapsulation + interface
- often: separation between exported and internal operations
  - for instance public, private ...
  - or via separate interfaces
- (static) classes in Java: may be used/seen as ADTs, methods are then the "operations"

```
ADT  begin
    integer i;
    real x;
    int proc total(int a) {
       return i * x + a   // or: ``total = i * x + a''
    }
end
```

Type constructors: building new types

- array type
- record type (also known as struct-types)
- union type
- pair/tuple type
- pointer type
  - explict as in C
  - implict distinction between reference and value types, hidden from programmers (e.g. Java)
- *signatures* (specifying methods / procedures / subroutines / functions) as type
- function type constructor, incl. higher-order types (in functional languages)
- (names of) classes and subclasses
- ...

Basically all languages support to build more complex types from the basic one and ways to use and check them. Sometimes it's not even very visible, for instance, one may already see strings as compound. For instance in C, which takes a very implementation-centric view on types, explains strings as

> one-dimensional array of characters terminated by a null character '\0'

Of course, there is special syntax to build values of type string, writing `"abc"` as opposed to `string-cons('a, string_cons('b, ...))` or similar... This smooth support of working with strings may make them feel as if being primitive.

In the following we will have a look at a few of composed types in programming languages. The Compila language of this year's oblig supports records but also "names" of records. We will also discuss the issue of "types as such" vs. "names of types" later (for instance in connection with the question how to "compare types: when are they equal or compatible, what about subtping? etc.).

Arrays

Array type

```
array [<indextype>] of <component type>
```

- elements (arrays) = (finite) functions from index-type to component type
- allowed index-types:
  - non-negative (unsigned) integers?, `from ... to ...`?
  - other types?: enumerated types, characters
- things to keep in mind:
  - indexing outside the array bounds?

   - are the array bounds (statically) known to the compiler?
   - *dynamic* arrays (extensible at run-time)?

Integer-indexed arrays are typically a very efficent data structure, as they mirror the layout of standard random access memory and customary hardware.[4] Indeed, contiguous random-access memory can be seen as one big array of "cells" or "words" and standard hardware *supports* fast access to to those cells by indirect addressing modes (like making use of an off-set from a base address, even offset multiplied by a factor (which represents the size of the entries)). In the later chapters about code generation, we will look a bit into different addressing modes of machine instructions.

One and more-dimensional arrays

- one-dimensional: efficiently implementable in standard hardware (relative memory addressing, known offset)
- two or more dimensions

```
array [1..4] of array [1..3] of real
array [1..4, 1..3]  of real
```

- one can see it as "array of arrays" (Java), an array is typically a reference type
- conceptually "two-dimensional"- *linear layout* in memory (language dependent)

Records ("structs")

```
struct {
  real r;
  int  i;
}
```

- values: "labelled tuples" (`real`× `int`)
- constructing elements, e.g.

```
struct point {int x; int y;};
struct point pt = { 300, 42 };
```

struct point

- access (read or update): *dot-notation* `x.i`
- implemenation: linear memory layout given by the (types of the) attributes
- attributes accessible by statically fixed *offsets*
- *fast* access
- cf. objects as in Java

[4] There exists unconventional hardware memory architectures which are *not* accessed via addresses, like content-addressable memory. Those don't resemble "arrays". They are a specialist niche, but have applications.

Structs in C

The following is not too important, just some side remarks on a bit esoteric aspects of structs in C: The definition, declaration etc. of struct types and structs in C is slightly confusing.

```
struct Foo { // Foo is called a ``tag''
   real r;
   int  i
```

The `foo` is a *tag*, which is almost like a type, but not quite, at least as far as C is concerned (i.e. the definition of C distinguishes even it is not so clear why). Technically, for instance, the name space for tags is different from that for types. Ignoring details, one can make use of the tag almost as if it were a type, for instance,

```
struct foo b
```

declares the structure `b` to adhere to the struct type tagged by `foo`. Since `foo` is not a proper type, what is illegal is a declaration such as `foo b`. In general the question whether one should use `typedef` in commbination with struct tags (or *only* `typedef`, leaving out the tag), seems a matter of debate. In general, the separation between tags and types (resp. type names) is a messy, ill-considered design. One should do better these days.

Tuple/product types

- $T_1 \times T_2$ (or in ascii `T_1 * T_2`)
- elements are *tuples*: for instance: `(1, "text")` is element of `int * string`
- generalization to n-tuples:

value	type
`(1, "text", true)`	`int * string * bool`
`(1, ("text", true))`	`int * (string * bool)`

- structs can be seen as "labeled tuples", resp. tuples as "anonymous structs"
- tuple types: common in functional languages,
- in C/Java-like languages: n-ary tuple types often only implicit as *input* types for procedures/methods (part of the "signature")

The two "triples" and their types touches upon an issue discussed later, namely when are two types equal (and related to that, whether or not the corresponding values (here the "triples") are equal.

Union types (C-style again)

```
union {
   real r;
   int  i
}
```

- related to *sum types* (outside C)
- (more or less) represents *disjoint union* of values of "participating" types
- access in C (confusingly enough): dot-notation `u.i`

Union types in C and type safety

- union types is C: bad example for (safe) type disciplines, as it's simply type-unsafe, basically an *unsafe* hack ...

Union type (in C):

- nothing much more than a directive to allocate enough memory to hold largest member of the union.
- in example: `real` takes more space than `int`

- implementor's (= low level) focus and memory allocation, not "proper usage focus" or assuring strong typing

⇒ bad example of modern use of types

- better (type-safe) implementations known since

⇒ *variant record* ("tagged"/"discriminated" union) or even inductive data types

Inductive types are basically: union types done right plus possibility of "recursion". On the next slide, we discuss variant records from Pascal. They try to remedy the deficiency of C-style records by adding as additional component some "discriminator". This possibility for enhanced security goes only half way, it's still possible to subvert the type system. Inductive data types also allow recursive definitions, and can be used for pattern matching, an elegant form of "case-switching".

Variant records from Pascal

```
record case isReal: boolean of
  true:  (r:real);
  false: (i:integer);
```

- "variant record"
- non-overlapping memory layout[5]
- programmer responsible to set and check the "discriminator" self
- enforcing type-safety-wise: not really an improvement :-(

```
record case  boolean of
  true:  (r:real);
  false: (i:integer);
```

Inductive types in ML and similar

- *type-safe* and powerful
- allows *pattern matching*

```
IsReal of real | IsInteger of int
```

- allows *recursive* definitions ⇒ inductive data types:

[5] Again, that's an implementor-centric view, not a user-centric one.

```
type int_bintree =
    Node of int * int_bintree * bintree
|   Nil
```

- Node, Leaf, IsReal: *constructors* (cf. languages like Java)
- constructors used as discriminators in “union” types

```
type exp =
     Plus of exp * exp
   | Minus of exp * exp
   | Number of int
   | Var of string
```

Recursive data types in C

does not work

```
struct intBST  {
  int val;
  int isNull;
  struct intBST left, right;
}
```

“indirect” recursion

```
struct intBST {
  int val;
  struct intBST *left, *right;
};
typedef struct intBST * intBST;
```

In Java: references implicit

```
class BSTnode {
 int val;
 BSTnode left, right;
```

- note: *implementation* in ML: also uses “pointers” (but hidden from the user)
- no nil-pointers in ML (and NIL is not a nil-pointer, it’s a constructor)

Pointer types

- *pointer* type: notation in C: int*
- “ * ”: can be seen as type constructor

```
int* p;
```

- random other languages: ^integer in Pascal, int ref in ML
- value: *address* of (or reference/pointer to) values of the underlying type

- operations: *dereferencing* and determining the address of an data item (and C allows “ *pointer arithmetic* ”)

```
var a:   ^integer   (* pointer to an integer     *)
var b:    integer
...
a := &i             (* i an int var              *)
                    (* a :=  new integer ok too *)
b:= ^a + b
```

Implicit dereferencing

- many languages: more or less hide existence of pointers
- cf. reference vs. value types often: automatic/implicit dereferencing

```
C r;
C r = new C();
```

- “sloppy” speaking: “ `r` is an object (which is an instance of class `C` /which is of type `C`)”,
- slightly more precise: variable “ `r` contains an object. . . ”
- precise: “variable `r` will contain a reference to an object”
- `r.field` corresponds to something like “ `(*r).field`, similar in Simula

Programming with pointers

- “popular” source of errors
- test for non-null-ness often required
- explicit pointers: can lead to problems in block-structured language (when handled non-expertly)
- watch out for parameter passing
- aliasing
- null-pointers: “the billion-dollar-mistake”
- take care of concurrency

Null pointer are generally attributed (actually including self-attributed) to Tony Hoare, famous for many landmark contributions. He himself refers to the introduction of null pointers or null references (1965 for ALGOL-W) as his billion dollar mistake. See also here, but the video seems no longer to work, but there is some notes or rudimentary transscript. One can also consult Hoare’s Turing Award lecture (1980), where he talks about similar topics. Also the text of the lecture is available on the net. In the lecture, he interestingly mentions as the *first* and foremost design principle for the design of ALGOL resp. the corresponding compiler: *security*. So it’s not that the intention was to say “to hell with security, speed comes first”. From the text, though, it seems that he speaks about “security” of the compiler itself, in that it should never crash (= “. . . no core dumps should ever be nessessary”).

Function variables

```
program Funcvar;
var pv : Procedure (x: integer);  (* procedur var     *)

   Procedure Q();
   var
      a : integer;
      Procedure P(i : integer);
      begin
         a:= a+i;    (* a def'ed outside              *)
      end;
   begin
      pv := @P;      (* ``return'' P (as side effect) *)
   end;              (* "@" dependent on dialect       *)
begin                (* here: free Pascal              *)
   Q();
   pv(1);
end.
```

Function variables and nested scopes

- tricky part here: nested scope + function definition *escaping* surrounding function/scope.
- here: inner procedure "returned" via assignment to function variable
- think about *stack discipline* of dynamic memory management?
- related also: functions allowed as return value?
  - Pascal: not directly possible (unless one "returns" them via function-typed reference variables like here)
  - C: possible, but *nested* function definitions not allowed
- combination of nested function definitions and functions as official return values (and arguments): *higher-order functions*
- Note: functions as arguments less problematic than as return values.

For the sake of the lecture: Let's not distinguish conceptually between functions and procedures. But in Pascal, a procedure does not return a value, functions do.

Function signatures

- define the "header" (also "signature") of a function[6]
- in the discussion: we don't distinguish mostly: functions, procedures, methods, subroutines.
- functional type (independent of the name f): `int→int`

Modula-2

```
var f: procedure (integer): integer;
```

C

```
int (*f) (int)
```

- *values*: all functions (procedures ...) with the given signature
- problems with block structure and free use of procedure variables.

[6] Actually, an identfier of the function is mentioned as well.

Escaping

```
 1 program Funcvar;
 2 var pv : Procedure (x: integer);   (* procedur var    *)
 3
 4    Procedure Q();
 5    var
 6       a : integer;
 7       Procedure P(i : integer);
 8       begin
 9          a:= a+i;    (* a def'ed outside              *)
10       end;
11    begin
12       pv := @P;      (* ``return'' P (as side effect) *)
13    end;              (* "@" dependent on dialect      *)
14 begin                (* here: free Pascal             *)
15    Q();
16    pv(1);
17 end.
```

- at the end of line 15: variable a no longer exists
- possible safe usage: only assign to such variables (here pv) a new value (= function) at the same blocklevel the variable is declared

As mentioned before function *parameters* less problematic than returning them (as with function variable), and the reason is that the stack-discipline in that case is still doable.

Classes and subclasses

Parent class

```
class A {
  int i;
  void f() {...}
}
```

Subclass B

```
class B extends A {
  int i
  void f() {...}
}
```

Subclass C

```
class C extends A {
  int i
  void f() {...}
}
```

- classes resemble records, and subclasses variant types, but additionally
  - visibility: local methods possible (besides fields)
  - subclasses
  - objects mostly created dynamically, *no* references into the stack

  - subtyping and polymorphism (subtype polymorphism): a reference typed by `A` can also point to `B` or `C` objects

- special problems: not really many, nil-pointer still possible

The three classes from above illustrate subclassing (and in many object-oriented languages, connected to that, subtyping). Note that the classes are *also* names of types. What is also is illustrated is *overriding* as far as `f` is concerned. Inheritance is actually not illustrated, insofar that `f` as only method involved is overridden, not inherited both in `B` and `C`. The methods `f` and the instance variables `i` are treated differently as far as binding is concerned. That will be discussed next. In the slides we use `rA` to refer to a variable of *static* type/class `A`.

Access to object members: late binding

- notation `rA.i` or `rA.f()`
- dynamic binding, late-binding, virtual access, dynamic dispatch ...: all mean roughly the same
- central mechanism in many OO language, in connection with inheritance

Virtual access `rA.f()` (methods)

"deepest" `f` in the run-time class of the *object*, `rA` points to

- remember: "most-closely nested" access of variables in nested lexical block
- Java:
  - methods "in" objects are only dynamically bound (but there are class methods too)
  - instance variables not, neither static methods "in" classes.

Example: fields and methods

```
public class Shadow {
    public static void main(String[] args){
        C2 c2 = new C2();
        c2.n();
    }
}

class C1 {
    String s = "C1";
    void m () {System.out.print(this.s);}
}


class C2 extends C1 {
    String s = "C2";
    void n () {this.m();}
}
```

The code is compilable Java code and can thus be tested. It is supposed to illustrated the discussed difference in the treatment of fields and methods, as far as binding is concerned. While the mechanism for methods (which are late or dynamically bound) is called overriding, the similar (but of course not same) situation for fields (which are statically bound) is called *shadowing*. One may also see it like that: fields are treated as if they were *static* methods.

Diverse notions

- *Overloading*
  - common for (at least) standard, built-in operations
  - also possible for user defined functions/methods . . .
  - disambiguation via (static) types of arguments
  - "ad-hoc" polymorphism
  - implementation:
    * put types of parameters as "part" of the name
    * look-up gives back a set of alternatives
- type-conversions: can be problematic in connection with overloading
- (generic) polymporphism
  `swap(var x,y:  anytype)`

7.3 Equality of types

Classes as types

- classes = types? Not so fast
- more precise view:
  - design decision in Java and similar languages (but not all/even not all class-based OOLs): that class *names* are used in the role of (names of) types.
- other roles of classes (in class-based OOLs)
  - generator of objects (via constructor, again with the same name)[7]
  - containing *code* that implements the instances

```
C x = new C()
```

Example with interfaces

```
interface I1 { int m (int x); }
interface I2 { int m (int x); }
class C1 implements I1 {
    public int m(int y) {return y++;  }
}
class C2 implements I2 {
    public int m(int y) {return y++;  }
}

public class Noduck1 {
    public static void main(String[] arg) {
        I1 x1 = new C1();        // I2 not possible
        I2 x2 = new C2();
        x1 = x2;                 // ???
    }
}
```

Analogous when using classes in their roles as types

[7] Not for Java's *static* classes etc, obviously.

When are 2 types "equal"?

- *type equivalence*
- surprisingly *many* different answers possible
- implementor's focus (deprecated): type `int` and `short` are equal, because they "are" both 2 byte
- type checker must often decide such equivalences
- related to a more fundamental question: what's a type?

Example: pairs of integers

```
type pair_of_ints = int * int;;
let x : pair_of_ints = (1,4);;
```

Questions

- Is "the" type of (values of) x `pair_of_ints`, or
- the product type `int * int` , or
- both, as they are equal, i.e., `pair_of_int` is an abbreviation of the product type (*type synonym*)?

For this particular language (ocaml), the piece of code is correct: the pair `(1,4)` is of type `int * int` and of type `pair_of_ints`.

Structural vs. nominal equality

a, b

```
var a, b: record
   int i;
   double d
 end
```

c

```
var c: record
   int i;
   double d
 end
```

typedef

```
typedef idRecord: record
    int i;
    double d
 end
```

```
var d: idRecord;
var e: idRecord;;
```

what's possible?

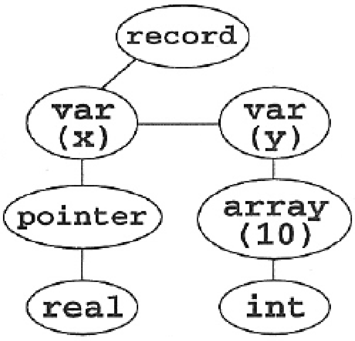
```

a := c;
a := d;

a := b;
d := e;
```

Types in the AST

- types are part of the syntax, as well
- represent: either in a separate symbol table, or part of the AST

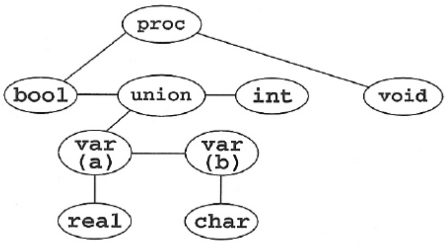
Record type

```
record
  x: pointer to real;
  y: array [10] of int
 end
```



Procedure header

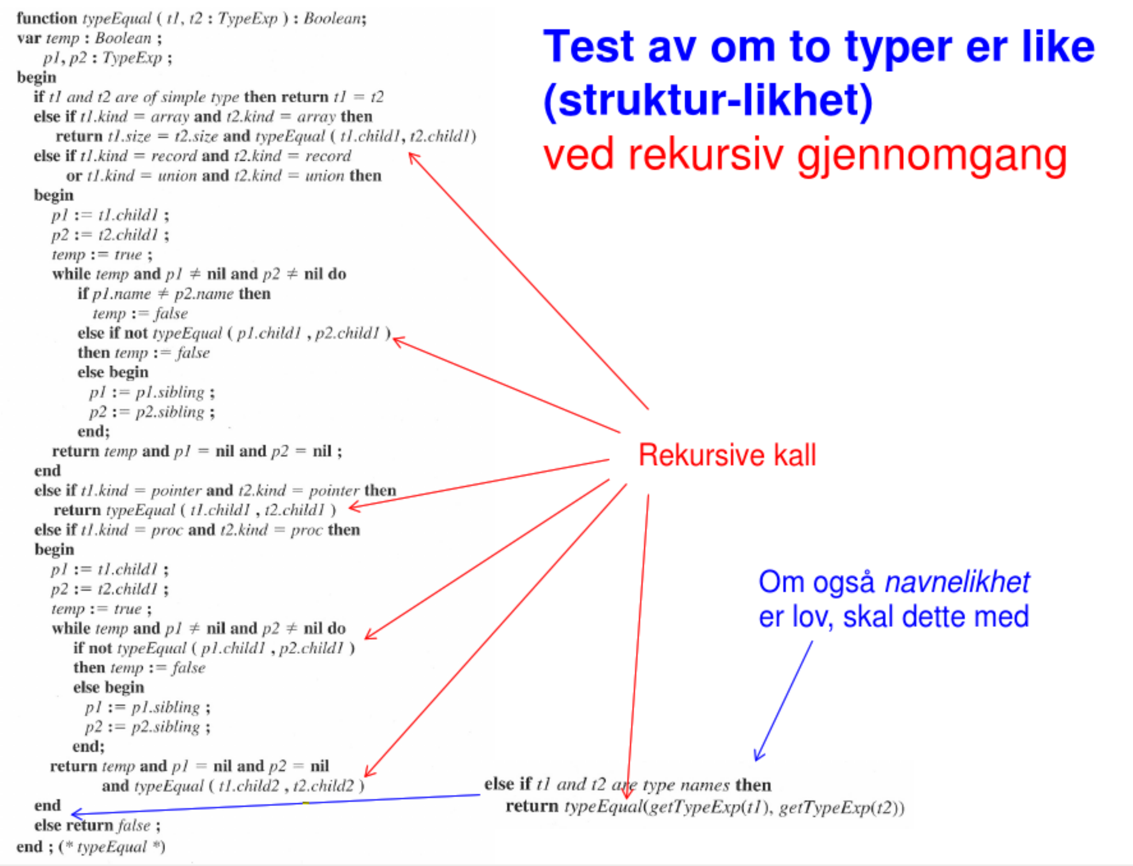
```
proc(bool,
      union a: real; b:char end,
      int):void
 end
```



Structured types without names

$$
\begin{array}{rcl}
\textit{var-decls} & \rightarrow & \textit{var-decls}\,\mathbf{;}\,\textit{var-decl} \mid \textit{var-decl} \\
\textit{var-decl} & \rightarrow & \mathbf{id:}\,\textit{type-exp} \\
\textit{type-exp} & \rightarrow & \textit{simple-type} \mid \textit{structured-type} \\
\textit{simple-type} & \rightarrow & \mathbf{int} \mid \mathbf{bool} \mid \mathbf{real} \mid \mathbf{char} \mid \mathbf{void} \\
\textit{structured-type} & \rightarrow & \mathbf{array}\,[\,\textit{num}\,] : \textit{type-exp} \\
 & \mid & \mathbf{record}\,\textit{var-decls}\,\mathbf{end} \\
 & \mid & \mathbf{union}\,\textit{var-decls}\,\mathbf{end} \\
 & \mid & \mathbf{pointerto}\,\textit{type-exp} \\
 & \mid & \mathbf{proc}\,\mathbf{(}\,\textit{type-exps}\,\mathbf{)}\,\textit{type-exp} \\
\textit{type-exps} & \rightarrow & \textit{type-exps}\,\mathbf{,}\,\textit{type-exp} \mid \textit{type-exp}
\end{array}
$$

Structural equality



```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;
var temp : Boolean ;
    p1, p2 : TypeExp ;
begin
  if t1 and t2 are of simple type then return t1 = t2
  else if t1.kind = array and t2.kind = array then
     return t1.size = t2.size and typeEqual ( t1.child1, t2.child1)
  else if t1.kind = record and t2.kind = record
       or t1.kind = union and t2.kind = union then
  begin
    p1 := t1.child1 ;
    p2 := t2.child1 ;
    temp := true ;
    while temp and p1 ≠ nil and p2 ≠ nil do
       if p1.name ≠ p2.name then
         temp := false
       else if not typeEqual ( p1.child1 , p2.child1 )
       then temp := false
       else begin
         p1 := p1.sibling ;
         p2 := p2.sibling ;
       end;
    return temp and p1 = nil and p2 = nil ;
  end
  else if t1.kind = pointer and t2.kind = pointer then
    return typeEqual ( t1.child1 , t2.child1 )
  else if t1.kind = proc and t2.kind = proc then
  begin
    p1 := t1.child1 ;
    p2 := t2.child1 ;
    temp := true ;
    while temp and p1 ≠ nil and p2 ≠ nil do
       if not typeEqual ( p1.child1 , p2.child1 )
       then temp := false
       else begin
         p1 := p1.sibling ;
         p2 := p2.sibling ;
       end;
    return temp and p1 = nil and p2 = nil
             and typeEqual ( t1.child2 , t2.child2 )
  end
  else return false ;
end ; (* typeEqual *)
```

```
else if t1 and t2 are type names then
  return typeEqual(getTypeExp(t1), getTypeExp(t2))
```

Test av om to typer er like (struktur-likhet)
ved rekursiv gjennomgang

Rekursive kall

Om også *navnelikhet* er lov, skal dette med

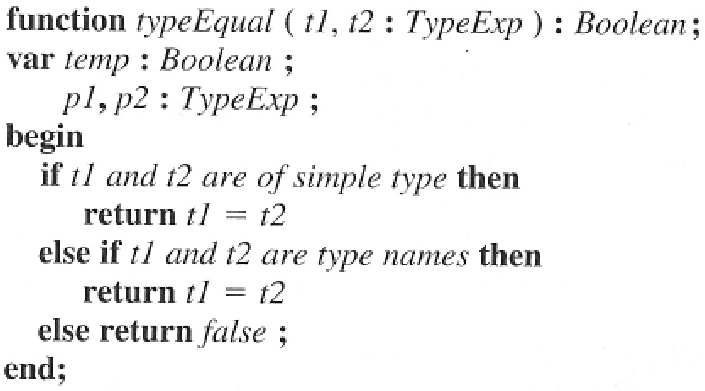
Types with names

```
         var-decls  →  var-decls ; var-decl  |  var-decl
          var-decl  →  id : simple-type-exp
        type-decls  →  type-decls ; type-decl  |  type-decl
         type-decl  →  id = type-exp
          type-exp  →  simple-type-exp  |  structured-type
   simple-type-exp  →  simple-type  |  id                  identifiers
       simple-type  →  int  |  bool  |  real  |  char  |  void
   structured-type  →  array [ num ] : simple-type-exp
                    |  record var-decls end
                    |  union var-decls end
                    |  pointerto simple-type-exp
                    |  proc ( type-exps ) simple-type-exp
         type-exps  →  type-exps , simple-type-exp
                    |  simple-type-exp
```

Name equality

- all types have “names”, and two types are equal iff their names are equal
- type equality checking: obviously simpler

- of course: type names may have *scopes*. . . .



```
function typeEqual ( t1, t2 : TypeExp ) : Boolean;
var temp : Boolean ;
      p1, p2 : TypeExp ;
begin
   if t1 and t2 are of simple type then
       return t1 = t2
   else if t1 and t2 are type names then
       return t1 = t2
   else return false ;
end;
```

Type aliases

- languages with type aliases (type synonyms): C, Pascal, ML . . . .
- often very convenient (`type Coordinate = float * float`)
- light-weight mechanism

type alias; make `t1` known also under name `t2`

```
t2  = t1   // t2 is the ``same type''.
```

- also here: different choices wrt. *type equality*

Type aliases: different choices

Alias, for simple types

```
t1 = int;
t2 = int;
```

- often: `t1` and `t2` are the "same" type

Alias of structured types

```
t1 = array [10] of int;
t2 = array [10] of int;
t3 = t2
```

- mostly `t3` $\neq$ `t1` $\neq$ `t2`

7.4 Type checking

Type checking of expressions (and statements)

- types of subexpressions must “fit” to the expected types the contructs can operate on
- type checking: top-down and *bottom-up* task

⇒ *synthesized* attributes, when using AGs

- Here: using an attribute grammar specification of the type checker
  - type checking conceptually done *while parsing* (as actions of the parser)
  - more common: type checker operates on the AST *after* the parser has done its job
- type **system** vs. type **checker**
  - type system: specification of the rules governing the use of types in a language, type discipline
  - type checker: algorithmic formulation of the type system (resp. implementation thereof)

Synthesized attributes

When drawing the parallel that type checking is a buttom-up (“synthesized”) task, that is only *half* of the picture. The slide focuses on type checking if expresions (and statements). When it comes to *declarations* (i.e., declaring a type for a variable, for instance), that part corresponds more to “inherited” attributes. Remember that one standard way of implementing the association of variables (“symbols”) with (here) types (which can be seen as an “attribute”) are symbol tables.

Overloading

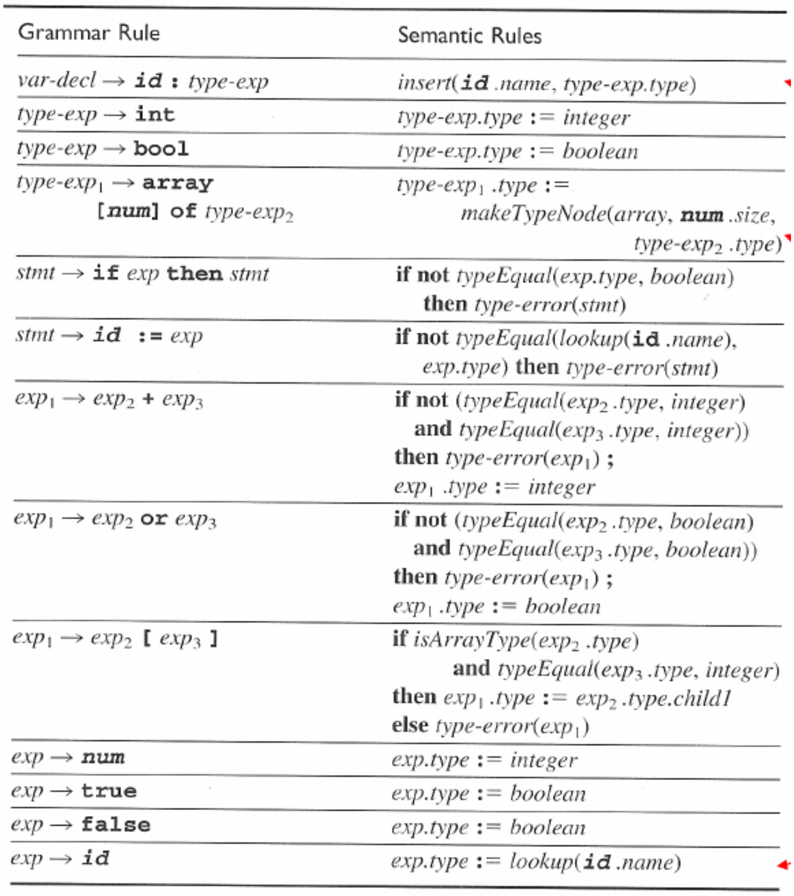
In case of (operator) overloading: that may complicate the picture slightly. Operators are selected depending on the type of the subexpressions. There will be some remarks concerning overloading later.

As said on the slides, the type checker mostly nowadays would work after the parser is finished, that means on the *abstract syntax tree*. One can, however, use grammars as specification of that *abstract* syntax tree as well, i.e., as a “second” grammar besides the grammar for concrete parsing, and that’s then the grammar the type checker works on.

Grammar for statements and expressions

$$\begin{array}{rcl}
\mathit{program} & \rightarrow & \mathit{var\text{-}decls}\,\mathbf{;}\,\mathit{stmts} \\
\mathit{var\text{-}decls} & \rightarrow & \mathit{var\text{-}decls}\,\mathbf{;}\,\mathit{var\text{-}decl} \mid \mathit{var\text{-}decl} \\
\mathit{var\text{-}decl} & \rightarrow & \mathbf{id}\,\mathbf{:}\,\mathit{type\text{-}exp} \\
\mathit{type\text{-}exp} & \rightarrow & \mathbf{int} \mid \mathbf{bool} \mid \mathbf{array}\,[\,\mathit{num}\,]\,:\,\mathit{type\text{-}exp} \\
\mathit{stmts} & \rightarrow & \mathit{stmts}\,\mathbf{;}\,\mathit{stmt} \mid \mathit{stmt} \\
\mathit{stmt} & \rightarrow & \mathbf{if}\ \mathit{exp}\ \mathbf{then}\ \mathit{stmt} \mid \mathbf{id}\,\mathbf{:=}\,\mathit{exp} \\
\mathit{exp} & \rightarrow & \mathit{exp} + \mathit{exp} \mid \mathit{exp}\,\mathbf{or}\,\mathit{exp} \mid \mathit{exp}\,[\,\mathit{exp}\,]
\end{array}$$

Type checking as semantic rules



Grammar Rule	Semantic Rules
var-decl → **id** **:** *type-exp*	*insert*(**id** *.name*, *type-exp.type*)
type-exp → **int**	*type-exp.type* := *integer*
type-exp → **bool**	*type-exp.type* := *boolean*
type-exp$_1$ → **array** **[num] of** *type-exp*$_2$	*type-exp*$_1$ *.type* := *makeTypeNode*(*array*, **num** *.size*, *type-exp*$_2$ *.type*)
stmt → **if** *exp* **then** *stmt*	**if not** *typeEqual*(*exp.type*, *boolean*) **then** *type-error*(*stmt*)
stmt → **id** **:=** *exp*	**if not** *typeEqual*(*lookup*(**id** *.name*), *exp.type*) **then** *type-error*(*stmt*)
exp$_1$ → *exp*$_2$ **+** *exp*$_3$	**if not** (*typeEqual*(*exp*$_2$ *.type*, *integer*) **and** *typeEqual*(*exp*$_3$ *.type*, *integer*)) **then** *type-error*(*exp*$_1$) ; *exp*$_1$ *.type* := *integer*
exp$_1$ → *exp*$_2$ **or** *exp*$_3$	**if not** (*typeEqual*(*exp*$_2$ *.type*, *boolean*) **and** *typeEqual*(*exp*$_3$ *.type*, *boolean*)) **then** *type-error*(*exp*$_1$) ; *exp*$_1$ *.type* := *boolean*

exp$_1$ → *exp*$_2$ **[** *exp*$_3$ **]**	**if** *isArrayType*(*exp*$_2$ *.type*)
	and *typeEqual*(*exp*$_3$ *.type*, *integer*)
	then *exp*$_1$ *.type* := *exp*$_2$ *.type.child1*
	else *type-error*(*exp*$_1$)

exp → **num**	*exp.type* := *integer*
exp → **true**	*exp.type* := *boolean*
exp → **false**	*exp.type* := *boolean*
exp → **id**	*exp.type* := *lookup*(**id** *.name*)

More “modern” presentation

- representation as derivation rules
- Γ: notation for symbol table
  - $\Gamma(x)$: look-up
  - $\Gamma, x : T$: insert
- more compact representation
- one reason: “errors” left implicit.

Type checking (expressions)

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \textsc{TE-Id} \qquad \frac{}{\Gamma \vdash \textbf{true} : \texttt{bool}}\ \textsc{TE-True} \qquad \frac{}{\Gamma \vdash \textbf{false} : \texttt{bool}}\ \textsc{T-False}$$

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \textsc{TE-Num}$$

$$\frac{\Gamma \vdash \mathit{exp}_2 : \ \texttt{array\_of}\ T \qquad \Gamma \vdash \mathit{exp}_3 : \texttt{int}}{\Gamma \vdash \mathit{exp}_2\,[\,\mathit{exp}_3\,] : T}\ \textsc{TE-Array}$$

$$\frac{\Gamma \vdash \mathit{exp}_1 : \texttt{bool} \qquad \Gamma \vdash \mathit{exp}_3 : \texttt{bool}}{\Gamma \vdash \mathit{exp}_2\ \textbf{or}\ \mathit{exp}_3 : \texttt{bool}}\ \textsc{Te-Or}$$

$$\frac{\Gamma \vdash \mathit{exp}_1 : \texttt{int} \qquad \Gamma \vdash \mathit{exp}_3 : \texttt{int}}{\Gamma \vdash \mathit{exp}_3 + \mathit{exp}_3 : \texttt{int}}\ \textsc{TE-Plus}$$

Declarations and statements

$$\frac{\Gamma, x : \texttt{int} \vdash \mathit{rest} : \texttt{ok}}{\Gamma \vdash x\,\textbf{:int;}\,\mathit{rest} : \texttt{ok}}\ \textsc{TD-Int} \qquad \frac{\Gamma, x : \texttt{bool} \vdash \mathit{rest} : \texttt{ok}}{\Gamma \vdash x\,\textbf{:bool;}\,\mathit{rest} : \texttt{ok}}\ \textsc{TD-Bool}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{num} : \texttt{int} \qquad \Gamma(\mathit{type\text{-}exp}) = T \\ \Gamma, x : \texttt{array}\ \mathit{num}\ \texttt{of}\ T \vdash \mathit{rest} : \texttt{ok}\end{array}}{\Gamma \vdash x\,\textbf{:array}\,[\,\mathit{num}\,] : \mathit{type\text{-}exp}\,;\,\mathit{rest} : \texttt{ok}}\ \textsc{TD-Array}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash \mathit{exp} : T}{\Gamma \vdash x := \mathit{exp} : \texttt{ok}}\ \textsc{TS-Assign} \qquad \frac{\Gamma \vdash \mathit{exp} : \texttt{bool}}{\Gamma \vdash \textbf{if}\ \ \mathit{exp}\ \ \textbf{then}\ \ \mathit{stmt} : \texttt{ok}}\ \textsc{TS-If}$$

$$\frac{\Gamma \vdash \mathit{stmt}_1 : \texttt{ok} \qquad \Gamma \vdash \mathit{stmt}_2 : \texttt{ok}}{\Gamma \vdash \mathit{stmt}_1\,;\,\mathit{stmt}_2 : \texttt{ok}}\ \textsc{TS-Seq}$$

Chapter
Run-time environments

What is it about?

Learning Targets of this Chapter

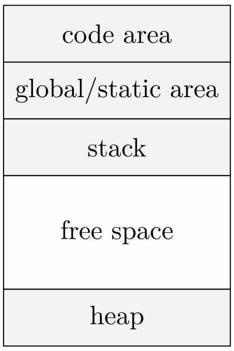
1. memory management
2. run-time environment
3. run-time stack
4. stack frames and their layout
5. heap

Contents

8.1 Intro

The chapter covers different aspects of the run-time environment of a language. The RTE refers to the design, organization and implementation of them

Static & dynamic memory layout at runtime



Memory

typical memory layout: for languages (as nowadays basically all) with

- static memory
- dynamic memory:

- stack
- heap

The picture represents schematically a typical layout of the memory associated with one (single-threaded) program under execution. At he highest level, there is a separation between "control" and "data" of the program. The "control" of a program is program code itself, in compiled form, of course, the machine code. The rest is the "data" the code operates on. Often, a strict separation between the two parts is enforced, even with the help of the hardware and/or the operating system. In principle, of course, the machine code is ultimately also "just bits", so conceptually the running program could modify the code section as well, leading to "self-modifying" code. That's seen as a no-no, and, as said, measures are taken that this does not happen. The generated code is not only kept immutable, it's also treated mostly as static (for instance as indicated in the picture): the compiler generates the code, decides on how to arrange the different parts of the code, i.e. to decide which code for which function comes where. Typically, as indicated at the picture, all code is grouped together into one big adjacent block of memory, which is called the code area.

The above discussing about the code area mentions that the control part of a program is structured into *procedures* (or functions, methods, subroutines . . . , generally one may use the term *callable unit*). That's a reminder that perhaps the single most important abstraction (as far as the control flow goes) of all but the lowest level languages is function abstraction: the ability to build "callable units" that can be reused at various points in a program, in different contexts, and with different arguments. Of course they may be reused not just by various points in one complied program, but by different programs (maybe even at the same time, in a multi-process environment). An collection of such callable units, arranged coherently and in a proper manner is, of course, a *library*.

The static placement of callable units into the code segment may remind us of functions as abstraction a programming mechanism, but it's not all that's needed to actually provide, i.e., implement that mechanism. At *run-time*, making use of a procedure means *calling it* and, when the procedure's code has executed till completion, *returning from it*. Returing means that that control continues at the point where the call originated (maybe not exactly at that point, but "immediately afterwards"). This call-and-return behavior is at the core of realizing the procedure abstraction. Calling a procedure can be seen as a jump (`JMP`) and likewise the return is nothing else than executing an according jump instruction. Execiting a jump does nothing else than setting program pointer to address given as argument of the instruction (which in the typical arrangement from the picture is supposed to be an address in the code segment). Jumps are therefore rather simple things, in particular, they are unaware of the intended call-return discipline. As a side remark: the platform may offer variations of the plain jump instruction (like `jump-to-subroutine` and `return-from-subroutine`, `JTS` and `RTS` or similar). That offer more "functionality" that helps realizing the procedure call-return discipline, but ulitmately, they are nothine else than a slighter more fancy form of jump, and the basic story remain: on top of hardware supported jumps, one has to arrange steps that, at run-time, realize the call and return behavior. That needs to involve the data area of the memory (since the code area is immutable). To the very least: a return from a procedure needs to know *where to return to* (since it's just a jump). So, when calling a function, the run-time system must arrange to remember where to return to (and then, when the time comes to actually return, look up that return address and us it for the jump back). In general, in all but the simplest languages, calls can be *nested*, i.e., a function being called can in turn call another function. In that nested situation procedures are executed *LIFO* fashion: the procedure called last is returned from first. That means, we need to arrange the remembered return addresses, one for each procedure call, in the form of a **stack**. The run-time stack is one key ingredient of the run-time system for many language. It's part of the *dynamic* portion of the data memory and separate in the picture from the other dynamic memory part, the heap, from a gulf of unused memory. In such an arrangement, the stack could grow "from above" and the heap "from below" (other arrangements are of course possible, for instance not having heap and stack compete for the same dynamic space, but each one living with an upper bound of their own).

So far we have discussed only the bare bones of the run-time environment to realize the procedure abstraction (the heap may be discussed later): in all by the very simplest settings, we need to arrange to maintain a stack for return addresses and manpulate the stack properly at run-time. If we had a trivial language, where function calls cannot be nested, we could do without a stack (or have a stack of maximal length 1, which is not much of a stack). In a setting without recursion (which we discuss also later), also similar simplifications are possible, and one could do without a official stack (though the call/return would still be executed under LIFO discipline, of course).

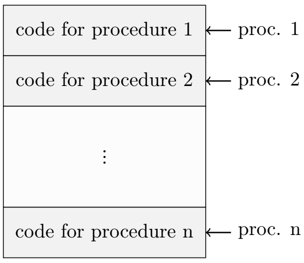
But beside those bare-bones return-address stack, the procedure abstraction has more to offer to the programmer than arranging a call/return execution of the control. What has been left out of the picture, which concentrated on the control so far, is the treatment of *data*, in particular *procedure local data*, so the question is related to how to realize at run-time the scoping rules that govern local data in the face of procedure calls. Related to that is that of procedure parameters and *parameter passing*. A procedure may have it's own local data, but als receives data upon being called as arguments. Indeed, the real power if the procedure abstraction not just relies on code (control) being available for repeated exection, it owes its power on equal parts that it can be executed variously on different *arguments*. Just relying on global variables and the fact that calling a function in different contexts or situations will give the procedure different states for some global values provides flexibility, but it's an undignified attempt to achieve something like *parameter passing*. All modern languages support syntax that allows the user to be explicit about what is considered the input of a procedures, it's formal parameters. And again, arrangements have to be made such that, at run-time the parameter passing is done properly. We will discuss different parameter-passing mechanisms later (the main being call-by-value, call-by-reference, and call-by-name, as well as some bastard scheme of lesser importance). Furthermore, when calling a procedure, the body may contain variables which are *not* local, but refer to variables defined and given values *outside* of the procedure (and without officially being passed as parameter). Also that needs to be arranged, and the arrangement varies deping on the scoping rules of the language (static vs. dynamic binding).

Anyway, the upshot of all of this is: we need a stack that contains *more* than just the return addresses, proper information pertaining to various aspects of *data* are needed as well. As a consequence, the single slots in the run-time stack become more complex; they are known as *activation record* (since the call of a procedure is also known as its activation).

The chapter will discuss different indgredients and variations of the activation record, depending on features of the language.

Modifying the control flow

Translated program code



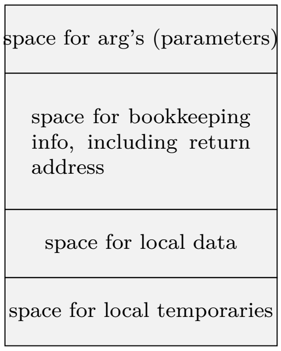
Code memory

- *code* segment: almost always considered as **statically** allocated

⇒ neither moved nor changed at runtime

- compiler aware of all addresses of "chunks" of code: *entry points* of the procedures
- but:
  - generated code often *relocatable*
  - final, absolute adresses given by *linker* / *loader*

Activation record


space for arg's (parameters)
space for bookkeeping info, including return address
space for local data
space for local temporaries

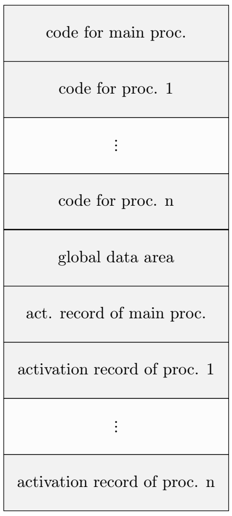
Schematic activation record

- *schematic* organization of activation records/activation block/stack frame ...
- goal: realize
  - parameter passing
  - scoping rules /local variables treatment
  - prepare for call/return behavior
- *calling conventions* on a platform

We will come back later to discuss possible designs for activation records in more detail, in the section about stack-based run-time environments. Activiation records (also known as stack frames) are the elementary slots of call stacks, a central way to organize the dynamic memory for languages with (recursive) procedures. There are also limitations of stack-based organizations, which we also touch upon.

8.2 Static layout

Full static layout



code for main proc.

code for proc. 1

⋮

code for proc. n

global data area

act. record of main proc.

activation record of proc. 1

⋮

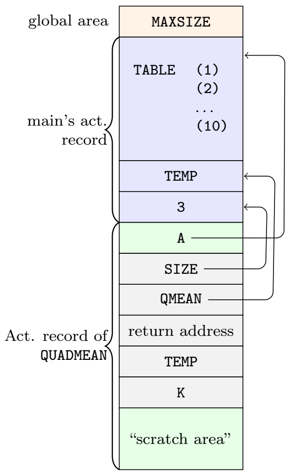
activation record of proc. n

- static addresses of all of memory known to the compiler
  - executable code
  - variables
  - all forms of auxiliary data (for instance big constants in the program, e.g., string literals)
- for instance: (old) Fortran
- nowadays rather seldom (or special applications like safety critical embedded systems)

Fortran example

```
      PROGRAM TEST
      COMMON MAXSIZE
      INTEGER MAXSIZE
      REAL TABLE(10),TEMP
      MAXSIZE = 10
      READ *, TABLE(1),TABLE(2),TABLE(3)
      CALL QUADMEAN(TABLE,3,TEMP)
      PRINT *,TEMP
      END

      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      COMMON MAXSIZE
      INTEGERMAXSIZE,SIZE
      REAL A(SIZE),QMEAN, TEMP
      INTEGER K
      TEMP = 0.0
      IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
      DO 10 K = 1, SIZE
         TEMP = TEMP + A(K)*A(K)
10    CONTINUE
99    QMEAN = SQRT(TEMP/SIZE)
      RETURN
      END
```

Static memory layout example/runtime environment



The details of the syntax and the exact way the program runs are not so important. Also for the layout on the next slides, the exact details don't matter too much. Important is the discinction between *global variables* and local ones, here for those for the "subroutine" (procedure). The local part of the memory for the procedure is a first taste of an *activation record*. Later they will be organized in a stack, and then they are also called *stack frames* but it's the same thing. It's space that will be used (at run-time) to fill the memory needs when calling the function (which is also known as "activation" of the function). That needed space involves slots used to pass arguments (parameter passing) and space for local variables. Needed also is a slot where to save the return address. Apart from the fact that exact details don't matter: what is often typical (and will also be typical) is that the parameters are stored in slots *before* the return address and the local variables afterwards. In a way, it's a design choice, not a logical necessity, but it's common (also later). It's often arranged like that, for reasons of efficiency. Later, the layout of the activation records will need some refinement, i.e., there will be more than the mentioned information (parameters, local variables, return address) to be stored, when we have to deal with recursion.

Static memory layout example/runtime environment

in Fortan (here Fortran77)

- **parameter passing** as *pointers* to the actual parameters
- activation record for QUADMEAN contains place for intermediate results, compiler calculates, how much is needed.
- note: one possible memory layout for FORTRAN 77, details vary, other implementations exists as do more modern versions of Fortran

8.3 Stack-based runtime environments

Stack-based runtime environments

- so far: no(!) *recursion*
- everything's static, incl. placement of activation records
- *ancient* and *restrictive* arrangement of the run-time envs
- calls and returns (also without recursion) follow at runtime a LIFO (= **stack-like**) discipline

Stack of activation records

- procedures as *abstractions* with own *local data*

⇒ run-time memory arrangement where procedure-local data together with other info (arrange proper returns, parameter passing) is organized as stack.

- AKA: *call stack*, *runtime stack*
- AR: exact format depends on language and platform

Situation in languages without local procedures

- recursion, but all procedures are *global*
- C-like languages

Activation record info (besides local data, see later)

- *frame pointer*
- *control link* (or *dynamic link*)[1]
- (optional): *stack pointer*
- *return address*

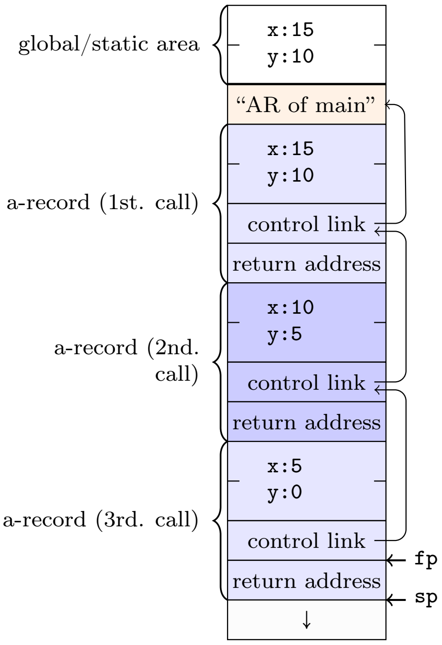
The notion of static links menioned in the footnote is basically the same we encountered before, when discussing the design of symbol tables, in particular how to arrange them properly for nested blocks and lexcical binding. Here (resp. shortly later down the road), the static links serve the same purpose, only not linking up (parts of a) symbol table, but activation records.

Euclid's recursive gcd algo

```
#include <stdio.h>

int x,y;

int gcd (int u, int v)
{ if (v==0) return u;
    else return gcd(v,u % v);
}

int main ()
{ scanf("%d%d",&x,&y);
  printf("%d\n",gcd(x,y));
  return 0;
}
```

[1] Later, we'll encounter also *static links* (aka *access* links).

Stack gcd



- **control link**
  - aka: dynamic link
  - refers to caller's FP
- **frame pointer** FP
  - points to a fixed location in the current a-record
- **stack pointer** (SP)
  - border of current stack and unused memory
- **return address**: program-address of call-site

Local and global variables and scoping

Code

```
int x  = 2; /* glob. var */
void g(int);/* prototype */

void f(int n)
  { static int x = 1;
    g(n);
    x--;
  }

void g(int m)
  { int y = m-1;
    if (y > 0)
      { f(y);
        x--;
        g(y);
      }
  }

int  main ()
  { g(x);
    return 0;
}
```

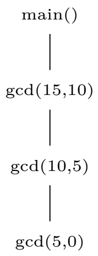
- global variable `x`
- but: (different) `x` *local* to `f`
- remember C:
  - call by value
  - static lexical scoping

Activation records and activation trees

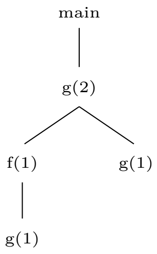
- *activation* of a function: corresponds to: *call* of a function
- **activation record**
  - data structure for run-time system
  - holds all relevant data for a function call and control-info in "standardized" form
  - control-behavior of functions: LIFO
  - if data *cannot* outlive activation of a function
  - ⇒ activation records can be arranged in as **stack** (like here)
  - in this case: activation record AKA *stack frame*

Activation record and activation trees

GCD

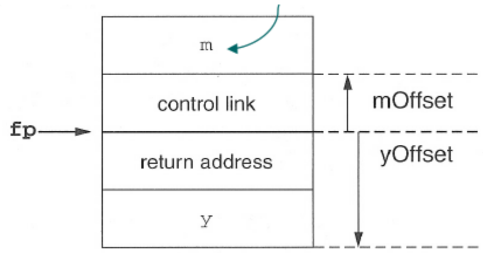


f and g example



Variable access and design of ARs

Layout g



- fp: frame pointer
- m (in this example): parameter of g

Possible arrangement of g's AR

- AR's: structurally *uniform* per language (or at least compiler) / platform
- different function defs, different size of AR

⇒ *frames* on the stack differently sized

- note: FP points
  - not: "top" of the frame/stack, but
  - to a well-chosen, well-defined position in the frame
  - other local data (local vars) accessible *relative* to that
- conventions
  - higher addresses "higher up"
  - stack "grows" towards lower addresses
  - in the picture: "pointers" to the "bottom" of the meant slot (e.g.: fp points to the control link: offset 0)

Layout for arrays of statically known size

Code

```
void f(int x, char c)
{ int a[10];
  double y;
  ..
}
```

name	offset
x	+5
c	+4
a	-24
y	-32

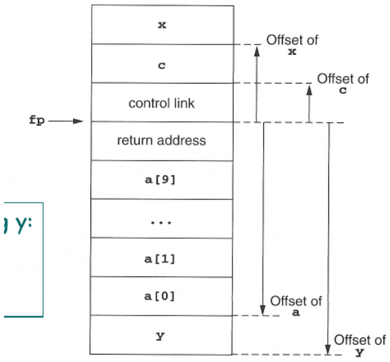
1. access of c and y

```
c:  4(fp)
y:  -32(fp)
```

2. access for `A[i]`

```
(-24+2*i)(fp)
```

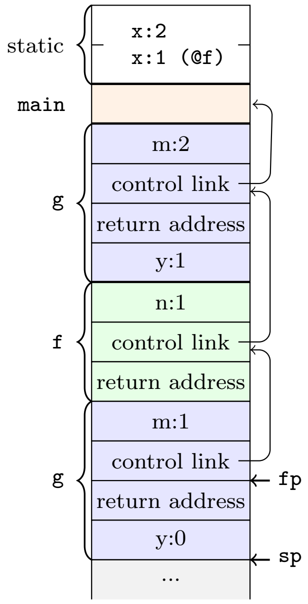
Layout

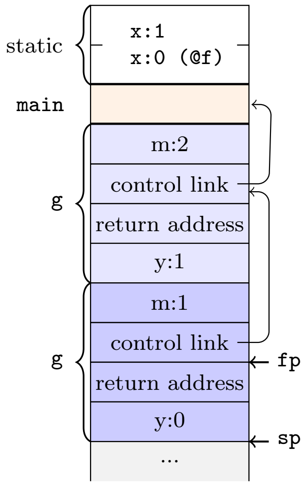


Back to the C code again (global and local variables)

```
int x  = 2; /* glob. var */
void g(int);/* prototype */

void f(int n)
  { static int x = 1;
    g(n);
    x--;
  }

void g(int m)
  { int y = m-1;
    if (y > 0)
      { f(y);
        x--;
       g(y);
      }
  }

int  main ()
  { g(x);
    return 0;
  }
```

2 snapshots of the call stack





- note: call by value, x in f *static*

How to do the "push and pop"

- **calling sequences**: AKA as *linking conventions* or *calling conventions*
- for RT environments: uniform design not just of
  - data structures (=ARs), but also of
  - uniform *actions* being taken when calling/returning from a procedure
- how to *do* details of "push and pop" on the call-stack

E.g: Parameter passing

- not just *where* (in the ARs) to find value for the actual parameter needs to be defined, but well-defined **steps** (ultimately **code**) that copies it there (and potentially reads it from there)

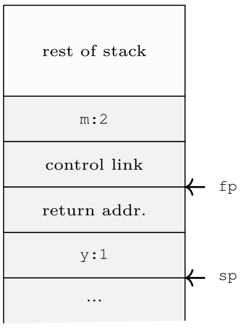
- “jointly” done by compiler + OS + HW
- distribution of *responsibilities* between caller and callee:
  - who copies the parameter to the right place
  - who saves registers and restores them
  - ...

Steps when calling

- For procedure call (entry)
  1. compute arguments, store them in the correct positions in the *new* activation record of the procedure (pushing them in order onto the runtime stack will achieve this)
  2. store (push) the `fp` as the *control link* in the new activation record
  3. change the `fp`, so that it points to the beginning of the new activation record. If there is an `sp`, copying the `sp` into the `fp` at this point will achieve this.
  4. store the return address in the new activation record, if necessary
  5. perform a *jump* to the code of the called procedure.
  6. Allocate space on the stack for local var’s by appropriate adjustement of the `sp`
- procedure exit
  1. copy the `fp` to the `sp` (inverting 3. of the entry)
  2. load the control link to the `fp`
  3. perform a jump to the return address
  4. change the `sp` to pop the arg’s

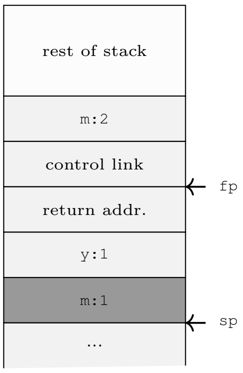
Steps when calling g

Before call



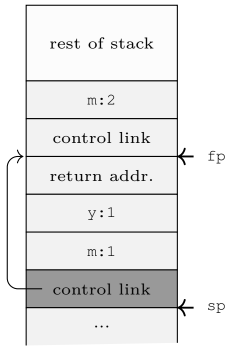
before call to g

Pushed m



pushed param.

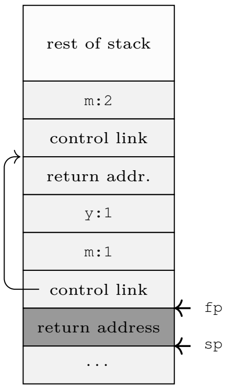
Pushed fp



pushed fp

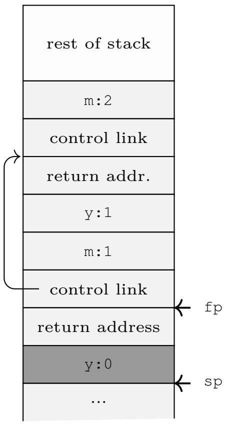
Steps when calling g (cont'd)

Return pushed



fp := sp,push return addr.

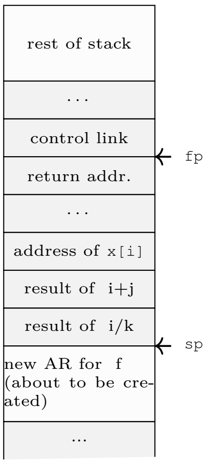
local var's pushed



alloc. local var y

Treatment of auxiliary results: “temporaries”

Layout picture



- calculations need *memory* for intermediate results.
- called **temporaries** in ARs.

```
x[i] = (i + j) * (i/k + f(j));
```

- note: x[i] represents an *address* or reference, i, j, k represent *values*[2]
- assume a strict left-to-right evaluation (call f(j) may change values.)
- *stack* of temporaries.
- [NB: compilers typically use **registers** as much as possible, what does not fit there goes into the AR.]

Variable-length data

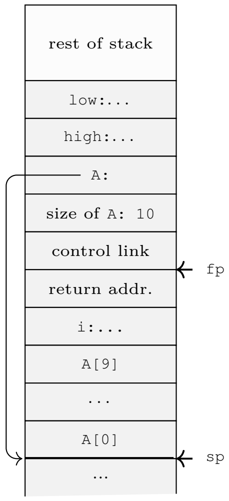
Ada code

```
type Int_Vector is
array(INTEGER range <>) of INTEGER;

procedure Sum(low, high: INTEGER;
 A: Int_Vector) return INTEGER
is
  i: integer
begin
     ...
  end Sum;
```

- Ada example
- assume: array passed *by value* (“copying”)
- A[i]: calculated as @6(fp) + 2*i
- in Java and other languages: arrays passed *by reference*
- note: space for A (as ref) and size of A is fixed-size (as well as low and high)

[2] integers are good for array-offsets, so they act as “references” as well.

Layout picture



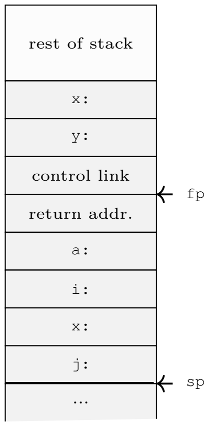
AR of call to SUM

Nested declarations ("compound statements")

C Code

```
void p(int x, double y)
{ char a;
  int i;
  ...;
 A:{ double x;
    int j;
    ...;
  }
  ...;
 B: { char * a;
    int k;
    ...;
  };
  ...;
}
```

Nested blocks layout (1)



rest of stack

x:

y:

control link

return addr.

a:

i:

x:

j:

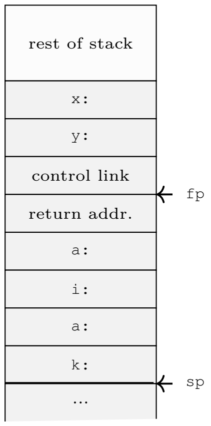
...

fp

sp

area for block A allocated

Nested blocks layout (2)



rest of stack

x:

y:

control link

return addr.

a:

i:

a:

k:

...

fp

sp

area for block B allocated

8.4 Stack-based RTE with nested procedures

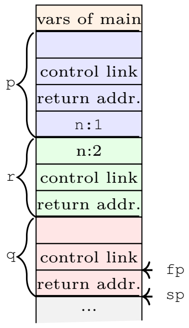
Nested procedures in Pascal

```
program nonLocalRef;
procedure p;
var n : integer;
   procedure q;
   begin
      (* a ref to n is now
       non-local, non-global *)
   end; (* q *)

   procedure r(n : integer);
   begin
      q;
   end; (* r *)

begin (* p *)
   n := 1;
   r(2);
end; (* p *)

begin (* main *)
   p;
end.
```

- proc. p contains q and r nested
- also "nested" (i.e., local) in p: integer n
  - in scope for q and r but
  - neither *global* nor *local* to q and r

Accessing non-local var's

Stack layout



calls m → p → r → q

- n in q: under *lexical* scoping: n declared in procedure p is meant
- this is not reflected in the stack (of course) as this stack represents the *run-time* call stack.
- remember: static links (or access links) in connection with *symbol tables*

Symbol tables

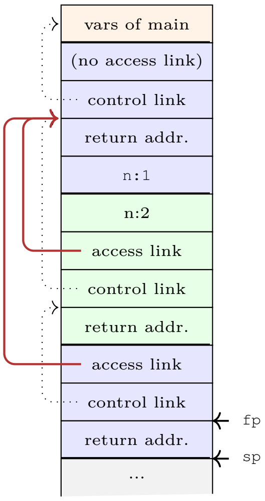
- "name-addressable" mapping
- access at compile time
- cf. scope tree

Dynamic memory

- “adresss-adressable” mapping
- access at run time
- stack-organized, reflecting paths in call graph
- cf. activation tree

Access link as part of the AR

Stack layout



calls m → p → r → q

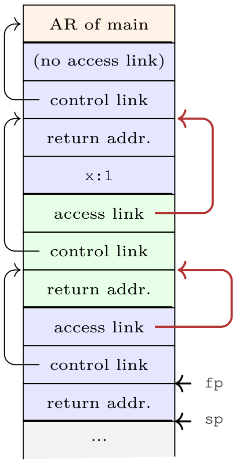
- **access link** (or **static link**): part of AR (at fixed position)
- points to stack-frame representing the current AR of the statically enclosed “procedural” scope

Example with multiple levels

```
program chain;

procedure p;
var x : integer;

   procedure q;
      procedure r;
      begin
         x:=2;
         ...;
         if ... then p;
      end; (*r *)
   begin
      r;
   end; (* q *)

begin
   q;
end; (* p *)

begin (* main *)
   p;
end.
```

Access chaining

Layout



calls m → p → q → r

- program chain
- access (conceptual): `fp.al.al.x`
- access link slot: fixed "offset" inside AR (but: AR's differently sized)
- "distance" from current AR to place of `x`
  - not fixed, i.e.
  - *statically* unknown!
- However: **number of access link dereferences statically known**
- lexical **nesting level**

Implementing access chaining

As example:

```
fp.al.al.al.  ...  al.x
```

- access need to be fast => use registers
- assume, at `fp` in dedicated register

```
4(fp) -> reg   // 1
4(fp) -> reg   // 2
...
4(fp) -> reg   // n = difference in nesting levels
6(reg)         // access content of x
```

- often: not so many block-levels/access chains nessessary

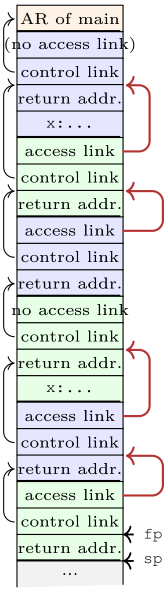
Calling sequence

- For procedure call (entry)
  1. compute arguments, store them in the correct positions in the *new* activation record of the procedure (pushing them in order onto the runtume stack will achieve this)
  2. – **push access link**, value calculated via link chaining (" `fp.al.al....` ")
     – store (push) the `fp` as the *control link* in the new AR
  3. change `fp`, to point to the "beginning"

  of the new AR. If there is an `sp`, copying `sp` into `fp` at this point will achieve this.
  1. store the return address in the new AR, if necessary
  2. perform a jump to the code of the called procedure.
  3. Allocate space on the stack for local var's by appropriate adjustement of the `sp`
- procedure exit
  1. copy the `fp` to the `sp`
  2. load the control link to the `fp`
  3. perform a jump to the return address
  4. change the `sp` to pop the arg's **and the access link**

Calling sequence: with access links

Layout



after 2nd call to `r`

- `main → p → q → r → p → q → r`
- calling sequence: actions to do the "push & pop"
- distribution of responsibilities between caller and callee
- generate an appropriate access chain, chain-length statically determined
- actual computation (of course) done at run-time

8.5 Functions as parameters

Nested procedures in Pascal

Access link (again)

Procedures as parameter

```
program closureex(output);

procedure p(procedure a);
begin
   a;
end;

procedure q;
var x : integer;
   procedure r;
   begin
      writeln(x);   // ``non-local''
   end;

begin
   x := 2;
   p (r);
end; (* q *)

begin (* main *)
   q;
end.
```

Procedures as parameters, same example in Go

```
package main
import ("fmt")

var p = func (a (func () ())) {  // (unit -> unit) -> unit
        a()
}

var q = func () {
        var x = 0
        var r = func () {
        fmt.Printf(" x = %v", x)
        }
        x = 2
        p(r)    // r as argument
}


func main() {
        q();
}
```

Procedures as parameters, same example in ocaml

```
let p (a :unit -> unit) : unit =  a();;

let q() =
  let x: int ref  =  ref 1
  in let r = function () ->  (print_int !x) (* deref *)
  in
  x := 2;    (* assignment to ref-typed var *)
  p(r);;

q();;  (* ``body of main'' *)
```

Closures and the design of ARs

- [9] rather "implementation centric"
- closure there:
  - **restricted** setting
  - specific way to achieve closures
  - specific semantics of non-local vars ("by reference")
- higher-order functions:
  - functions as arguments *and* return values
  - nested function declaration
- similar problems with: "function variables"
- Example shown: **only** procedures as *parameters*, not *returned*

Closures, schematically

- independent from concrete design of the RTE/ARs:
- what do we need to execute the body of a procedure?

Closure (abstractly)

A closure is a function body[3] *together* with the values for all its variables, including the non-local ones.[3]

- individual AR not enough for all variables used (non-local vars)
- in *stack*-organized RTE's:
  - fortunately ARs are *stack*-allocated
  - → with clever use of "links" (access/static links): possible to access variables that are "nested further out"/ deeper in the *stack* (following links)

Organize access with procedure parameters

- when calling `p`: allocate a stack frame
- executing `p` calls `a` => another stack frame
- number of parameters etc: knowable from the type of `a`
- *but* 2 problems

"control-flow" problem

currently only RTE, but: how can (the compiler arrange that) `p` calls `a` (and allocate a frame for `a`) if `a` is not know yet?

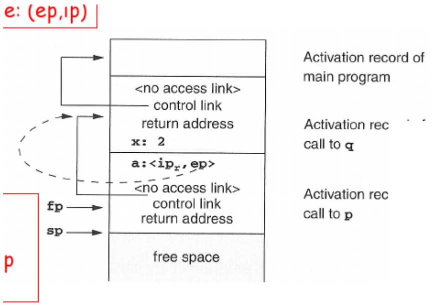
data problem

How can one statically arrange that `a` will be able to access non-local variables if statically it's not known what `a` will be?

- solution: for a procedure variable (like `a`): *store* in AR
  - **reference** to the code of argument (as representation of the function body)
  - **reference** to the frame, i.e., the relevant *frame pointer* (here: to the frame of `q` where `r` is defined)
- this pair = **closure**!

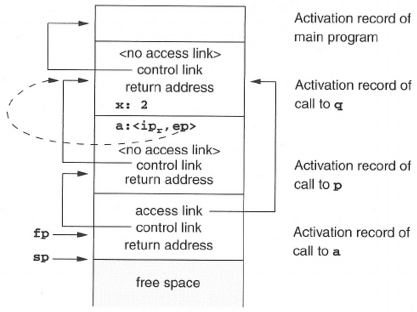
[3] Resp.: at least the possibility to locate them.

Closure for formal parameter a of the example



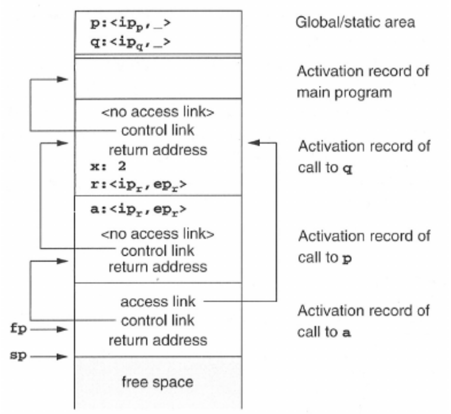
- stack after the call to p
- closure $\langle ip, ep \rangle$
- *ep*: refers to q's frame pointer
- note: distinction in calling sequence for
  - calling "ordinary" proc's and
  - calling procs in proc parameters (i.e., via closures)
- that may be unified ("closures" only)

After calling a (= r)



- note: *static* link of the new frame: used from the closure!

Making it uniform



- note: calling conventions *differ*
  - calling procedures as formal parameters
  - “standard” procedures (statically known)
- treatment can be made uniform

Limitations of stack-based RTEs

- procedures: **central** (!) control-flow abstraction in languages
- stack-based allocation: intuitive, common, and efficient (supported by HW)
- used in many languages
- procedure calls and returns: LIFO (= stack) behavior
- AR: local data for procedure body

Underlying assumption for stack-based RTEs

The data (=AR) for a procedure cannot **outlive** the activation where they are declared.

- assumption can break for many reasons
  - returning *references* of local variables
  - higher-order functions (or function variables)
  - “undisciplined” control flow (rather deprecated, goto’s can break any scoping rules, or procedure abstraction)
  - explicit memory allocation (and deallocation), pointer arithmetic etc.

Dangling ref’s due to returning references

```
int * dangle (void) {
  int x;        // local var
  return &x;    // address of x
}
```

- similar: returning references to objects created via `new`
- variable’s lifetime may be over, but the reference lives on . . .

Function variables

```
program Funcvar;
var pv : Procedure (x: integer);  (* procedur var     *)

   Procedure Q();
   var
      a : integer;
      Procedure P(i : integer);
      begin
         a:= a+i;    (* a def'ed outside               *)
      end;
   begin
      pv := @P;      (* ``return'' P (as side effect) *)
   end;              (* "@" dependent on dialect       *)
begin                (* here: free Pascal              *)
   Q();
   pv(1);
end.
```

```
funcvar
Runtime error 216 at $0000000000400233
  $0000000000400233
  $0000000000400268
  $00000000004001E0
```

Functions as return values

```
package main
import ("fmt")

var f = func () (func (int) int)  { // unit -> (int -> int)
        var x = 40                  // local variable
        var g = func (y int) int { // nested function
                return x + 1
        }
        x = x+1                     // update x
        return g                    // function as return value
}

func main() {
        var x = 0
        var h = f()
        fmt.Println(x)
        var r = h (1)
        fmt.Printf(" r = %v", r)
}
```

- function `g`
  - defined local to `f`
  - uses `x`, non-local to `g`, local to `f`
  - is being returned from `f`

Fully-dynamic RTEs

- full higher-order functions = functions are “data” same as everything else
  - function being locally defined
  - function as arguments to other functions
  - functions returned by functions

→ ARs cannot be stack-allocated

- closures needed, but *heap*-allocated (≠ Louden)
- objects (and references): *heap*-allocated
- less “disciplined” memory handling than stack-allocation
- **garbage** collection
- often: stack based allocation + fully-dynamic (= heap-based) allocation

The stack discipline can be seen as a particularly simple (and efficient) form of garbage collection: returning from a function makes it clear that the local data can be thrashed.

8.6 Parameter passing

Communicating values between procedures

- procedure *abstraction*, **modularity**
- parameter passing = communication of values between procedures
- from caller to callee (and back)
- binding actual parameters
- with the help of the RTE
- **formal** parameters vs. **actual** parameters
- two modern versions
  1. call by value
  2. call by reference

CBV and CBR, roughly

Core distinction/question

on the level of caller/callee *activation records* (on the stack frame): how does the AR of the callee get hold of the value the caller wants to hand over?

1. callee's AR with a *copy* of the value for the formal parameter
2. the callee AR with a *pointer* to the memory slot of the actual parameter

- if one has to choose only one: it's call-by-value
- remember: non-local variables (in lexical scope), nested procedures, and even closures:
  - those variables are "smuggled in" *by reference*
  - [NB: there are also *by value* closures]

CBV is in a way the prototypical, most dignified way of parameter passsing, supporting the procedure abstraction. If one has references (explicit or implicit, of data on the *heap*, typically), then one has call-by-value-of-references, which, in some way "feels" for the programmer as call-by-reference. Some people even call that call-by-reference, even if it's technically not.

Parameter passing by-value

- in C: CBV only parameter passing method
- in some lang's: formal parameters "immutable"
- straightforward: *copy* actual parameters → formal parameters (in the ARs).

C examples

```
void inc2 (int x)
{ ++x, ++x; }
```

```
void inc2 (int* x)
{ ++(*x), ++(*x); }
/* call: inc(&y) */
```

```
void init(int x[], int size) {
  int i;
  for (i=0;i<size,++i) x[i]= 0
}
```

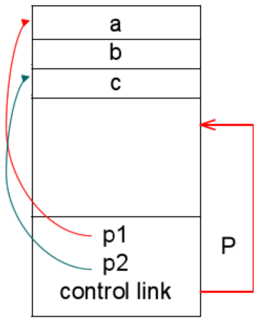
arrays: “by-reference” data

Call-by-reference

- hand over pointer/reference/address of the actual parameter
- useful especially for large data structures
- typically (for cbr): actual parameters must be *variables*
- Fortran actually allows things like `P(5,b)` and `P(a+b,c)`.

```
void inc2 (int* x)
{ ++(*x), ++(*x); }
/* call: inc(&y) */
```

```
void P(p1,p2) {
  ..
  p1 = 3
}
var a,b,c;
P(a,c)
```



Call-by-value-result

- *call-by-value-result* can give *different* results from cbr
- allocated as a *local* variable (as cbv)
- however: copied “two-way”
  - when calling: actual → formal parameters
  - when returning: actual ← formal parameters
- aka: “copy-in-copy-out” (or “copy-restore”)
- Ada’s `in` and `out` paremeters
- *when* are the value of actual variables determined when doing “actual ← formal parameters”
  - when calling
  - when returning
- not the cleanest parameter passing mechanism around…

Call-by-value-result example

```
void p(int x, int y)
{
  ++x;
  ++y;
}

main ()
{  int a = 1;
  p(a,a);    // :-O
  return 0;
}
```

- C-syntax (C has cbv, not cbvr)
- note: *aliasing* (via the arguments, here obvious)
- cbvr: same as cbr, unless *aliasing* “messes it up”[4]

Call-by-name (C-syntax)

- most complex (or is it . . . ?)
- hand over: textual representation (“name”) of the argument (substitution)
- in that respect: a bit like *macro expansion* (but lexically scoped)
- actual paramater *not* calculated *before* actually used!
- on the other hand: if needed more than once: *recalculated* over and over again
- aka: *delayed evaluation*
- Implementation
  - actual paramter: represented as a small procedure (*thunk*, *suspension*), if actual parameter = expression
  - optimization, if actually parameter = variable (works like call-by-reference then)

Call-by-name examples

- in (imperative) languages without procedure parameters:
  - delayed evaluation most visible when dealing with things like `a[i]`
  - `a[i]` is actually like “apply `a` to index `i`”
  - combine that with side-effects (`i++`) ⇒ pretty confusing

Example 1

```
void p(int x) {...; ++x; }
```

- call as `p(a[i])`
- corresponds to `++(a[i])`
- note:
  - `++ _` has a side effect
  - `i` may change in . . .

Example 2

```
int i;
int a[10];
void p(int x) {
  ++i;
  ++x;
}

main () {
  i = 1;
  a[1] = 1;
  a[2] = 2;
  p(a[i]);
  return 0;
}
```

[4] One can ask though, if not call-by-reference would be messed-up in the example already.

Another example: “swapping”

```
int i; int a[i];

swap (int a, b) {
  int i;
  i = a;
  a = b;
  b = i;
}

i = 3;
a[3] = 6;

swap (i,a[i]);
```

- note: local and global variable *i*

Call-by-name illustrations

Code

```
procedure P(par): name par, int par
begin
  int x,y;
  ...
  par := x + y; (* alternative: x:= par + y *)

end;

P(v);
P(r.v);
P(5);
P(u+v)
```

	v	r.v	5	u+v
par := x+y	ok	ok	error	error
x := par +y	ok	ok	ok	ok

Call by name (Algol)

```
begin comment Simple array example;
  procedure zero (Arr,i,j,u1,u2);
  integer Arr;
  integer i,j,u1,u2;
begin
    for i := 1 step 1 until u1 do
      for j := 1 step 1 until u2 do
        Arr := 0

end;

integer array Work  [1:100,1:200];
integer p,q,x,y,z;
x := 100;
y := 200
zero(Work[p,q],p,q,x,y);
end
```

Lazy evaluation

- call-by-name
  - complex & potentially confusing (in the presence of *side effects*)
  - not really used (there)
- declarative/functional languages: **lazy** evaluation
- optimization:

  - avoid recalculation of the argument
  - ⇒ remember (and share) results after first calculation (“memoization”)
  - works only in absence of side-effects
- most prominently: Haskell
- useful for operating on *infinite* data structures (for instance: streams)

Lazy evaluation / streams

```
magic :: Int -> Int -> [Int]
magic 0 _ = []
magic m n = m : (magic n (m+n))

getIt :: [Int] -> Int -> Int
getIt []      _ = undefined
getIt (x:xs) 1 = x
getIt (x:xs) n = getIt xs (n-1)
```

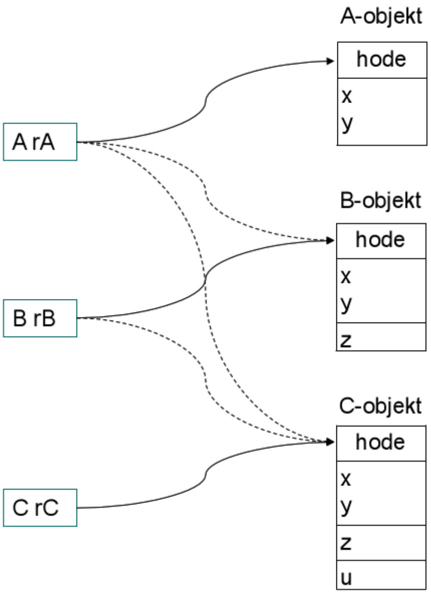
8.7 Virtual methods in OO

Object-orientation

- class-based/inheritance-based OO
- classes and sub-classes
- typed references to objects
- *virtual* and *non-virtual* methods

Virtual and non-virtual methods + fields

```
class A {
  int x,y
    void f(s,t) { ... $F_A$ ... };
  virtual void g(p,q) { ... $G_A$ ... };
};


class B extends A {
  int z
    void f(s,t) { ... $F_B$ ... };
    redef void g(p,q) {... $G_B$ ...};
    virtual void h(r) { ... $H_B$ ...}
};


class C extends B {
  int u;
  redef void h(r) { ... $H_C$ ... };
}
```



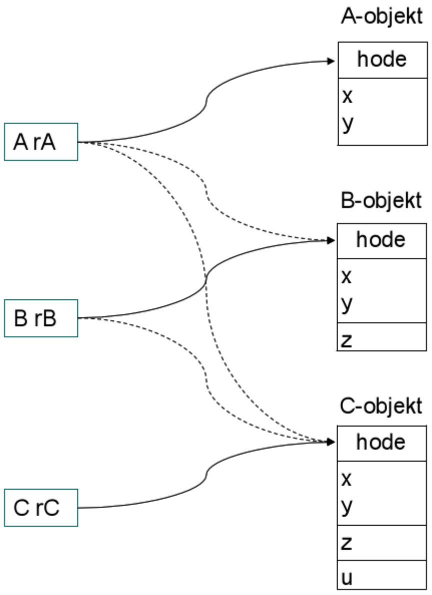
Call to virtual and non-virtual methods

non-virtual method f

call	target
$r_A.f$	F_A
$r_B.f$	F_B
$r_C.f$	F_B

virtual methods g and h

call	target
$r_A.g$	G_A or G_B
$r_B.g$	G_B
$r_C.g$	G_B
$r_A.h$	illegal
$r_B.h$	H_B or H_C
$r_C.h$	H_C



Late binding/dynamic binding

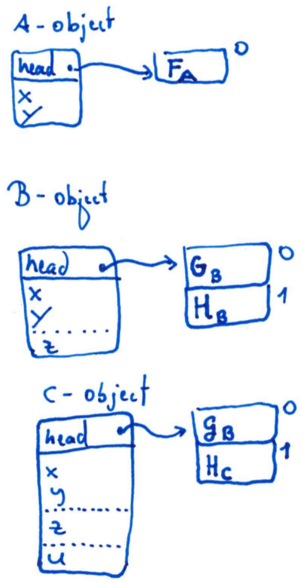
- details very much depend on the language/flavor of OO
  - single vs. multiple inheritance?
  - method update, method extension possible?
  - how much information available (e.g., static type information)?
- simple approach: “embedding” methods (as references)
  - seldomly done (but needed for updateable methods)
- using *inheritance graph*
  - each object keeps a pointer to its class (to locate virtual methods)
- virtual function table
  - in static memory
  - no traversal necessary
  - class structure need be known at compile-time
  - C^{++}

Virtual function table

- static check (“type check”) of $r_X.f()$
  - for virtual methods: `f` must be defined in X or one of its superclasses
- non-virtual binding: finalized by the compiler (static binding)
- virtual methods: enumerated (with offset) from the first class with a virtual method, redefinitions get the same “number”
- object “headers”: point to the class’s **virtual function table**
- $r_A.g()$:

```
call r_A.virttab[g_offset]
```

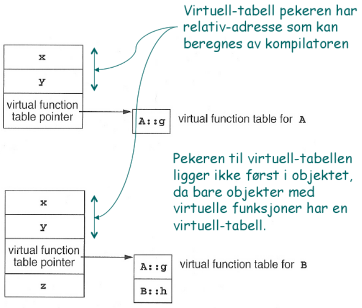
- compiler knows
  - `g_offset = 0`
  - `h_offset = 1`



Virtual method implementation in C++

- according to [9]

```
class A {
  public:
  double x,y;
  void f();
  virtual void g();
};

class B: public A {
  public:
  double z;
  void f();
  virtual void h();
};
```


x
y
virtual function table pointer
A::g
virtual function table for A
x
y
virtual function table pointer
z
A::g
B::h
virtual function table for B

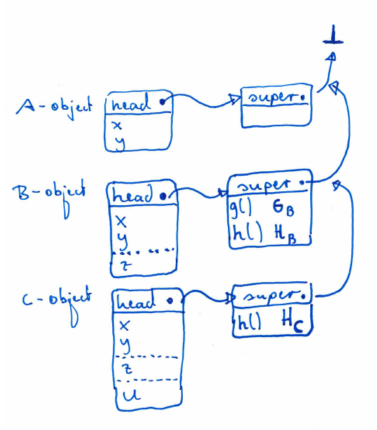
Virtuell-tabell pekeren har relativ-adresse som kan beregnes av kompilatoren

Pekeren til virtuell-tabellen ligger ikke først i objektet, da bare objekter med virtuelle funksjoner har en virtuell-tabell.

Untyped references to objects (e.g. Smalltalk)

- all methods *virtual*
- *problem* of virtual-tables now: virtual tables need to contain all methods of all classes
- additional complication: *method extension*, extension methods
- Thus: implementation of `r.g()` (assume: `f` omitted)
  - go to the object's class

  - *search* for g following the superclass hierarchy.



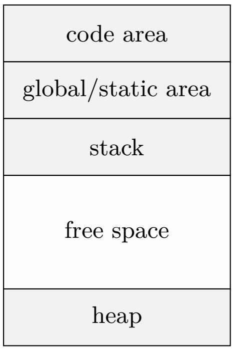
8.8 Garbage collection

Management of dynamic memory: GC & alternatives

- *dynamic* memory: allocation & deallocation at *run-time*
- different alternatives
  1. manual
     - "alloc", "free"
     - error prone
  2. "stack" allocated dynamic memory
     - typically not called GC
  3. automatic *reclaim* of unused dynamic memory
     - requires extra provisions by the compiler/RTE

Heap

- "heap" unrelated to the well-known heap-data structure from A&D
- part of the *dynamic* memory
- contains typically
  - objects, records (which are dynamocally allocated)
  - often: arrays as well
  - for "expressive" languages: heap-allocated activation records
    - ∗ coroutines (e.g. Simula)
    - ∗ higher-order functions



code area

global/static area

stack

free space

heap

Memory

Problems with free use of pointers

```
int * dangle (void) {
  int x;       // local var
  return &x;   // address of x
}
```

```
typedef int (* proc) (void);

proc g(int x) {
  int f(void) { /* illegal  */
    return x;
  }
  return f;
}

main () {
  proc c;
  c = g(2);
  printf("%d\n", c()); /* 2? */
  return 0;
}
```

- as seen before: references, higher-order functions, coroutines etc ⇒ heap-allocated ARs
- higher-order functions: typical for functional languages,
- heap memory: no LIFO discipline
- *unreasonable* to expect user to "clean up" AR's (already `alloc` and `free` is error-prone)
- ⇒ garbage collection (already dating back to 1958/Lisp)

Some basic design decisions

- gc *approximative*, but non-negotiable condition: **never** reclaim cells which *may* be used in the future
- one basic decision:
  1. never *move* "objects"
     – may lead to fragmentation
  2. *move* objects which are still needed
     – extra administration/information needed
     – all reference of moved objects need adaptation
     – all free spaces collected adjacently (defragmentation)

- *when* to do gc?
- *how* to get info about definitely unused/potentially used obects?
  – "monitor" the interaction program ↔ heap while it *runs*, to keep "up-to-date" all the time
  – inspect (at approriate points in time) the *state* of the heap

Objects here are meant as heap-allocated entities, which in OO languages includes objects, but here referring also to other data (records, arrays, closures . . .).

Mark (and sweep): marking phase

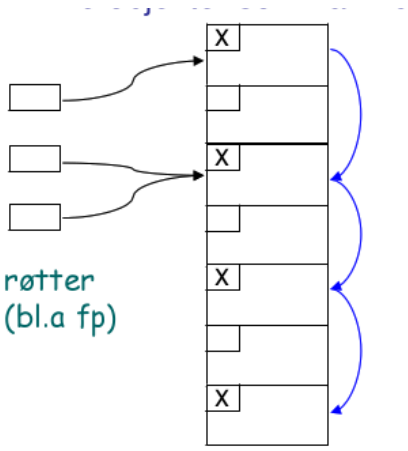
- observation: heap addresses only **reachable**

  directly through variables (with references), kept in the run-time stack (or registers)

  indirectly following fields in reachable objects, which point to further objects . . .

- heap: *graph* of objects, entry points aka "roots" or *root set*
- *mark*: starting from the root set:
  - find reachable objects, *mark* them as (potentially) used
  - one boolean (= 1 *bit* info) as mark
  - depth-first search of the graph

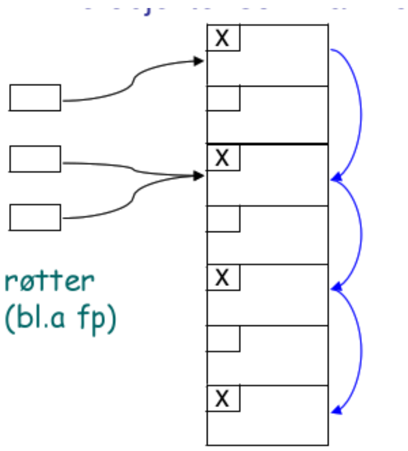
Marking phase: follow the pointers via DFS



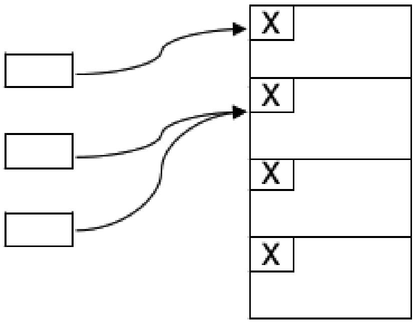
- layout (or "type") of objects need to be known to determine where pointers are
- food for thought: doing DFS requires a *stack*, in the worst case of comparable size as the heap itself . . . .

Compactation

Marked



Compacted

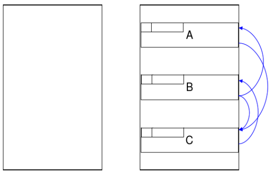


After marking?

- known *classification* in “garbage” and “non-garbage”
- pool of “unmarked” objects
- however: the “free space” not really ready at hand:
- two options:
  1. *sweep*
     - go again through the heap, this time sequentially (no graph-search)
     - collect all unmarked objects in **free list**
     - objects remain at their place
     - RTE need to allocate new object: grab free slot from free list
  2. *compaction* as well:
     - avoid fragmentation
     - move non-garbage to one place, the rest is big free space
     - when *moving* objects: adjust pointers

Stop-and-copy

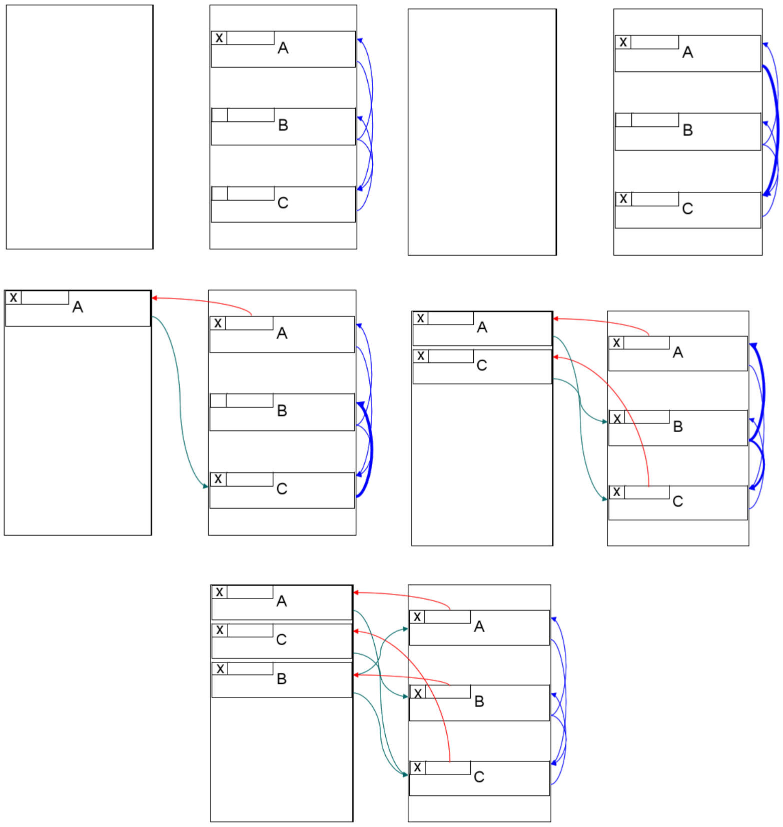
- variation of the previous compactation
- mark & compactation can be done in recursive pass
- space for heap-managment
  - split into *two halves*
  - only one half used at any given point in time
  - compactation by copying all non-garbage (marked) to the currently unused half



Hvert objekt må
ha et ledig bit
(”er flyttet”)

Da angir ”neste
ordet” adressen
det er flyttet til

Step by step



Chapter
Intermediate code generation

Learning Targets of this Chapter

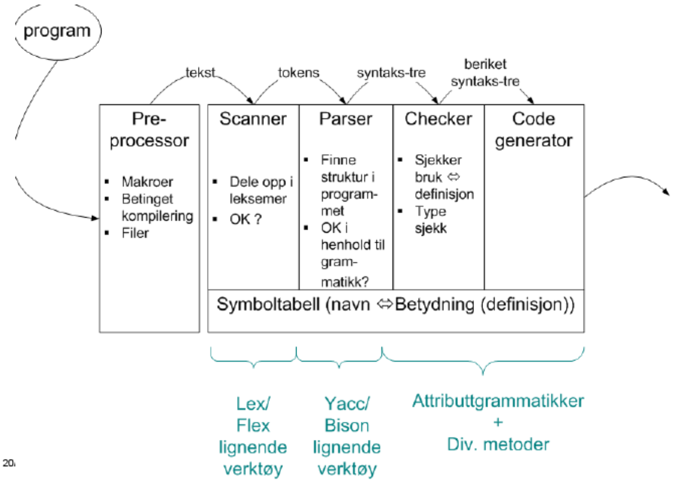
1. intermediate code
2. three-address code and P-code
3. translation to those forms
4. translation between those forms

Contents

What is it about?

9.1 Intro

Schematic anatomy of a compiler[1]

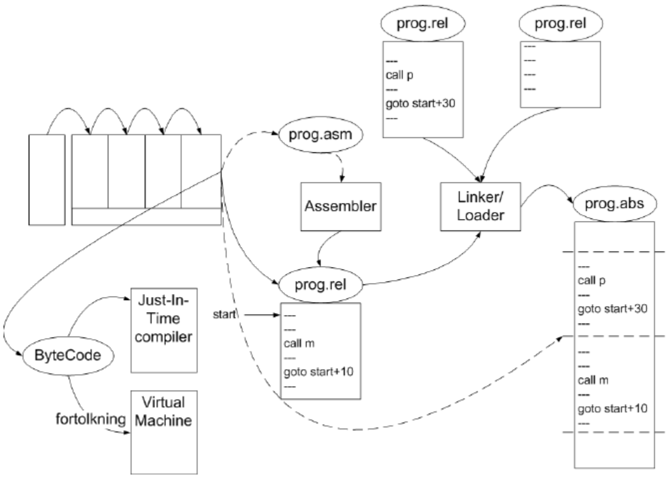

program
tekst
tokens
syntaks-tre
beriket syntaks-tre
Pre-processor
Makroer
Betinget kompilering
Filer
Scanner
Dele opp i leksemer
OK ?
Parser
Finne struktur i program-met
OK i henhold til gram-matikk?
Checker
Sjekker bruk ⇔ definisjon
Type sjekk
Code generator
Symboltabell (navn ⇔Betydning (definisjon))
Lex/ Flex lignende verktøy
Yacc/ Bison lignende verktøy
Attributtgrammatikker + Div. metoder
20.

- code generator:

[1] This section is based on slides from Stein Krogdahl, 2015.

  - may in itself be "phased"
  - using additional intermediate representation(s) (IR) and *intermediate code*

A closer look



Various forms of "executable" code

- different forms of code: relocatable vs. "absolute" code, relocatable code from libraries, assembler, etc
- often: specific file extensions
  - Unix/Linux etc.
    - asm: `*.s`
    - rel: `*.a`
    - rel from library: `*.a`
    - abs: files without file extension (but set as executable)
  - Windows:
    - abs: `*.exe`[2]
- *byte code* (specifically in Java)
  - a form of intermediate code, as well
  - executable on the JVM
  - in .NET/C$\sharp$: *CIL*
    - also called byte-code, but compiled further

Generating code: compilation to machine code

- 3 main forms or variations:
  1. machine code in textual **assembly format** (assembler can "compile" it to 2. and 3.)
  2. **relocatable** format (further processed by *loader*)
  3. **binary** machine code (directly executable)
- seen as different representations, but otherwise equivalent
- in practice: for *portability*
  - as another intermediate code: "platform independent" *abstract machine code* possible.
  - capture features shared roughly by many platforms
    - e.g. there are *stack frames*, static links, and push and pop, but *exact* layout of the frames is platform dependent
  - platform dependent details:

[2] `.exe`-files include more, and "assembly" in .NET even more

    - ∗ platform dependent code
    - ∗ filling in call-sequence / linking conventions

  done in a last step

Byte code generation

- semi-compiled well-defined format
- platform-independent
- further away from any HW, quite more high-level
- for example: Java byte code (or CIL for .NET and C$^\sharp$)
  - can be interpreted, but often compiled further to machine code ("just-in-time compiler" JIT)
- executed (interpreted) on a "virtual machine" (JVM)
- often: *stack-oriented* execution code (in post-fix format)
- also *internal* intermediate code (in compiled languages) may have stack-oriented format ("P-code")

9.2 Intermediate code

Use of intermediate code

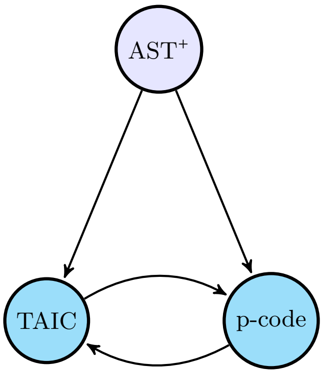
- two kinds of IC covered
  1. **three-address code** (3AC, 3AIC)
     - generic (platform-independent) abstract machine code
     - new names for all intermediate results
     - can be seen as unbounded pool of maschine registers
     - advantages (portability, optimization . . .)
  2. **P-code** ("Pascal-code", cf. Java "byte code")
     - originally proposed for interpretation
     - now often translated before execution (cf. JIT-compilation)
     - intermediate results in a *stack* (with postfix operations)
- *many* variations and elaborations for both kinds
  - addresses represented *symbolically* or as *numbers* (or both)
  - granularity/"instruction set"/level of abstraction: high-level op's available e.g., for array-access or: translation in more elementary op's needed.
  - operands (still) typed or not
  - . . .

Various translations in the lecture

Text

- AST here: tree structure *after* semantic analysis, let's call it AST$^+$ or just simply AST.
- translation AST ⇒ P-code: appox. as in Oblig 2
- we touch upon many general problems/techniques in "translations"
- one (important) aspect ignored for now: *register allocation*

Picture



9.3 Three address code

Three-address code

- common (form of) IR

TA: Basic format

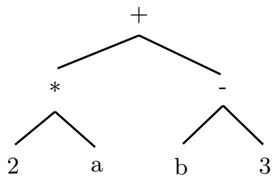
$$x = y \textbf{ op } z$$

- x, y, z: names, constants, temporaries …
- some operations need fewer arguments

- example of a (common) **linear IR**
- *linear* IR: ops include *control-flow* instructions (like jumps)
- alternative linear IRs (on a similar level of abstraction): 1-address code (stack-machine code), 2 address code
- well-suited for optimizations
- modern archictures often have 3-address code like instruction sets (RISC-architectures)

3AC example (expression)

`2*a+(b-3)`



Three-address code

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

alternative sequence

```
t1 = b - 3
t2 = 2 * a
t3 = t2 + t1
```

3AIC instruction set

- basic format: $x = y\,\mathbf{op}\,z$
- but also:
  - $x = \mathbf{op}\,z$
  - $x = y$
- *operators*: +,-,*,/, <, >, `and`, `or`
- `read` x, `write` x
- `label` L (sometimes called a “pseudo-instruction”)
- conditional jumps: `if_false` x `goto` L
- t_1, t_2, t_3 .... (or `t1, t2, t3, ...`): **temporaries** (or temporary variables)
  - assumed: *unbounded* reservoir of those
  - note: “non-destructive” assignments (single-assignment)

Illustration: translation to 3AIC

Source

```
read x; { input an integer }
if  0<x then
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0;
  write fact { output:
                 factorial of x }
end
```

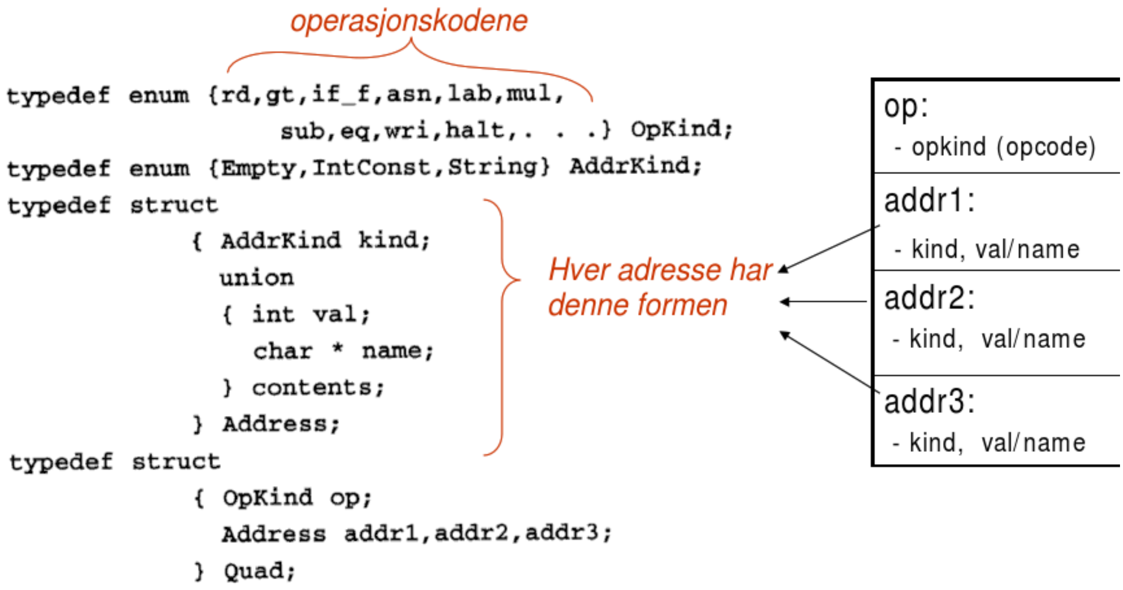
Target: 3AIC

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Variations in the design of TA-code

- provide operators for `int`, `long`, `float` ....?
- how to represent program *variables*
  - names/symbols
  - pointers to the declaration in the symbol table?
  - (abstract) machine address?
- how to store/represent TA *instructions*?
  - **quadruples**: 3 "addresses" + the op
  - *triple* possible (if target-address (left-hand side) is always a *new temporary*)

Quadruple-representation for 3AIC (in C)



```
typedef enum {rd,gt,if_f,asn,lab,mul,
                   sub,eq,wri,halt,. . .} OpKind;
typedef enum {Empty,IntConst,String} AddrKind;
typedef struct
             { AddrKind kind;
               union
               { int val;
                 char * name;
               } contents;
             } Address;
typedef struct
             { OpKind op;
               Address addr1,addr2,addr3;
             } Quad;
```

9.4 P-code

P-code

- different common intermediate code / IR
- aka "one-address code"[3] or stack-machine code
- originally developed for Pascal
- remember: post-fix printing of syntax trees (for expressions) and "reverse polish notation"

Example: expression evaluation `2*a+(b-3)`

[3] There's also two-address codes, but those have fallen more or less in disuse.

```
ldc 2   ; load constant 2
lod a   ; load value of variable a
mpi     ; integer multiplication
lod b   ; load value of variable b
ldc 3   ; load constant 3
sbi     ; integer substraction
adi     ; integer addition
```

P-code for assignments: `x := y + 1`

- assignments:
  - variables left and right: *L-values* and *R-values*
  - cf. also the values ↔ references/addresses/pointers

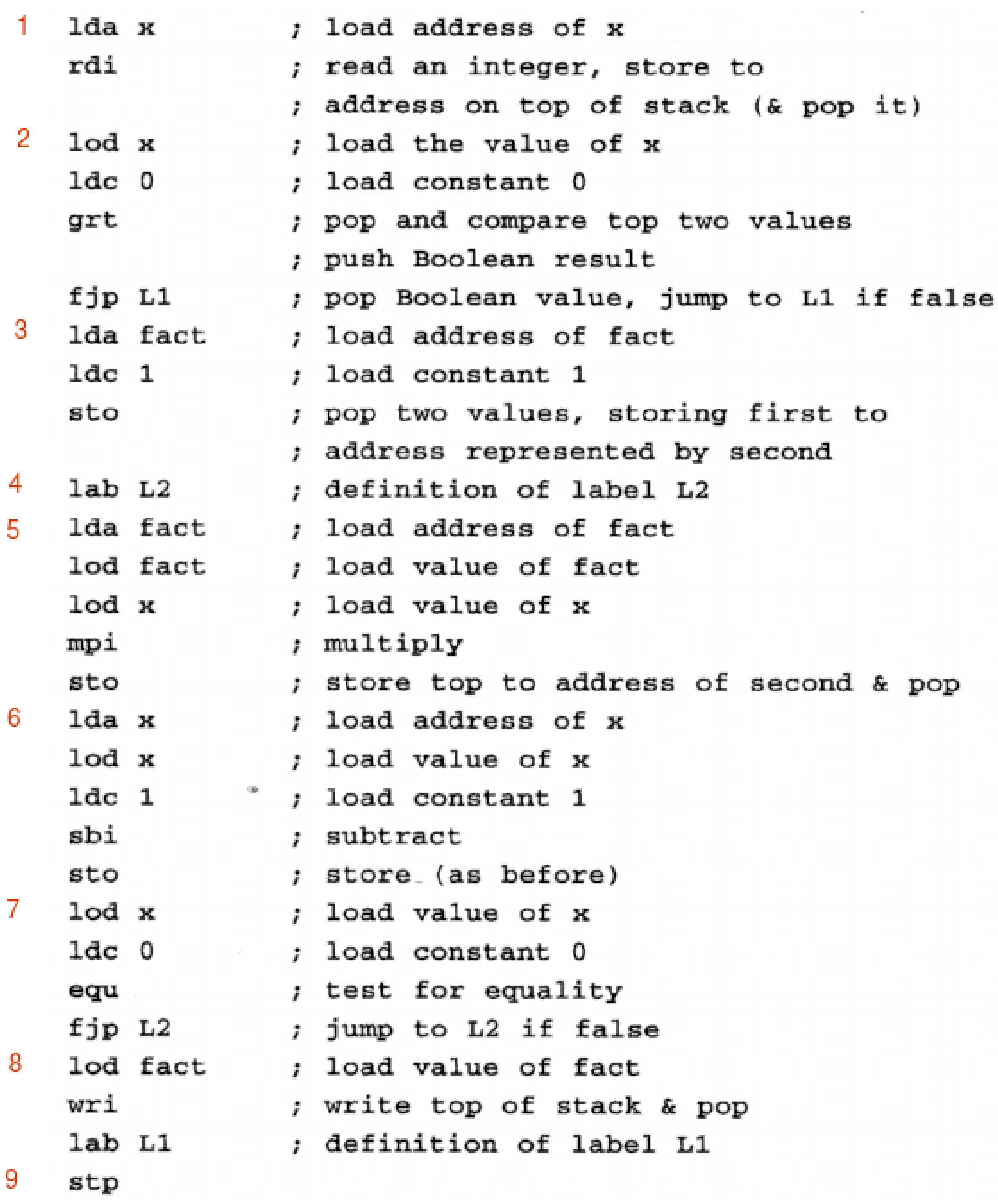
```
lda x      ; load address of x
lod y      ; load value of y
ldc 1      ; load constant 1
adi        ; add
sto        ; store top to address
           ; below top & pop both
```

P-code of the faculty function

Source

```
read x; {  input an integer }
if  0<x then
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0;
  write fact { output:
               factorial of x }
end
```

P-code



```
1   lda x        ; load address of x
    rdi          ; read an integer, store to
                 ; address on top of stack (& pop it)
2   lod x        ; load the value of x
    ldc 0        ; load constant 0
    grt          ; pop and compare top two values
                 ; push Boolean result
    fjp L1       ; pop Boolean value, jump to L1 if false
3   lda fact     ; load address of fact
    ldc 1        ; load constant 1
    sto          ; pop two values, storing first to
                 ; address represented by second
4   lab L2       ; definition of label L2
5   lda fact     ; load address of fact
    lod fact     ; load value of fact
    lod x        ; load value of x
    mpi          ; multiply
    sto          ; store top to address of second & pop
6   lda x        ; load address of x
    lod x        ; load value of x
    ldc 1        ; load constant 1
    sbi          ; subtract
    sto          ; store (as before)
7   lod x        ; load value of x
    ldc 0        ; load constant 0
    equ          ; test for equality
    fjp L2       ; jump to L2 if false
8   lod fact     ; load value of fact
    wri          ; write top of stack & pop
    lab L1       ; definition of label L1
9   stp
```

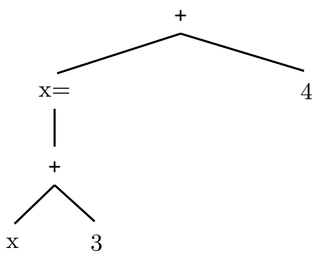
9.5 Generating P-code

Expression grammar

Grammar

$$
\begin{aligned}
exp_1 &\rightarrow \mathbf{id} = exp_2 \\
exp &\rightarrow aexp \\
aexp &\rightarrow aexp_2 + factor \\
aexp &\rightarrow factor \\
factor &\rightarrow (\, exp \,) \\
factor &\rightarrow \mathbf{num} \\
factor &\rightarrow \mathbf{id}
\end{aligned}
$$

`(x=x+3)+4`



Generating p-code with A-grammars

- goal: p-code as *attribute* of the grammar symbols/nodes of the syntax trees
- *syntax-directed translation*
- technical task: turn the syntax tree into a *linear* IR (here P-code)

⇒
  - “linearization” of the syntactic tree structure
  - while translating the nodes of the tree (the syntactical sub-expressions) one-by-one

- not recommended at any rate (for modern/reasonably complex language): code generation *while* parsing[4]

The use of A-grammars is perhps more a conceptual picture, In practice, one may not use a-grammars and corresponding tools in the *implementation*.

A-grammar for statements/expressions

- focus here on expressions/assignments: leaving out certain complications
- in particular: control-flow complications
  - two-armed conditionals
  - loops, etc.
- also: code-generation “intra-procedural” only, rest is filled in as *call-sequences*
- A-grammar for intermediate code-gen:
  - rather simple and straightforwad
  - only 1 *synthesized* attribute: `pcode`

[4] one can use the a-grammar formalism also to describe the treatment of ASTs, not concrete syntax trees/parse trees.

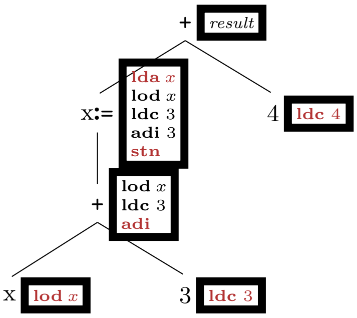
A-grammar

- “string” concatenation: ++ (construct separate instructions) and ˆ (construct one instruction)[5]

productions/grammar rules			semantic rules
exp_1	→	**id** = exp_2	exp_1 .pcode = ”lda”ˆ**id**.strval ++ exp_2 .pcode ++ ”stn”
exp	→	$aexp$	exp .pcode = $aexp$ .pcode
$aexp_1$	→	$aexp_2$ + $factor$	$aexp_1$ .pcode = $aexp_2$ .pcode ++ $factor$ .pcode ++ ”adi”
$aexp$	→	$factor$	$aexp$ .pcode = $factor$ .pcode
$factor$	→	(exp)	$factor$ .pcode = exp .pcode
$factor$	→	**num**	$factor$ .pcode = ”ldc”ˆ**num**.strval
$factor$	→	**id**	$factor$ .pcode = ”lod”ˆ**num**.strval

`(x = x + 3) + 4`

Attributed tree



“result” attr.

```
lda  x
lod  x
ldc  3
adi
stn
ldc  4
adi      ;  +
```

Rest

- note: here x=x+3 has side effect *and “return” value* (as in C . . .):
- **stn** (“store non-destructively”)
  - similar to **sto** , but *non-destructive*
    1. take top element, store it at address represented by 2nd top
    2. discard address, but not the top-value

[5] So, the result is not 100% linear. In general, one should not produce a flat string already.

Overview: p-code data structures

Source

```
type symbol = string

type expr =
  | Var of symbol
  | Num  of int
  | Plus of expr * expr
  | Assign of symbol * expr
```

Target

```
type instr   =  (* p-code instructions *)
    LDC of int
  | LOD of symbol
  | LDA of symbol
  | ADI
  | STN
  | STO


type tree = Oneline of instr
  | Seq of tree * tree

type program = instr list
```

Rest

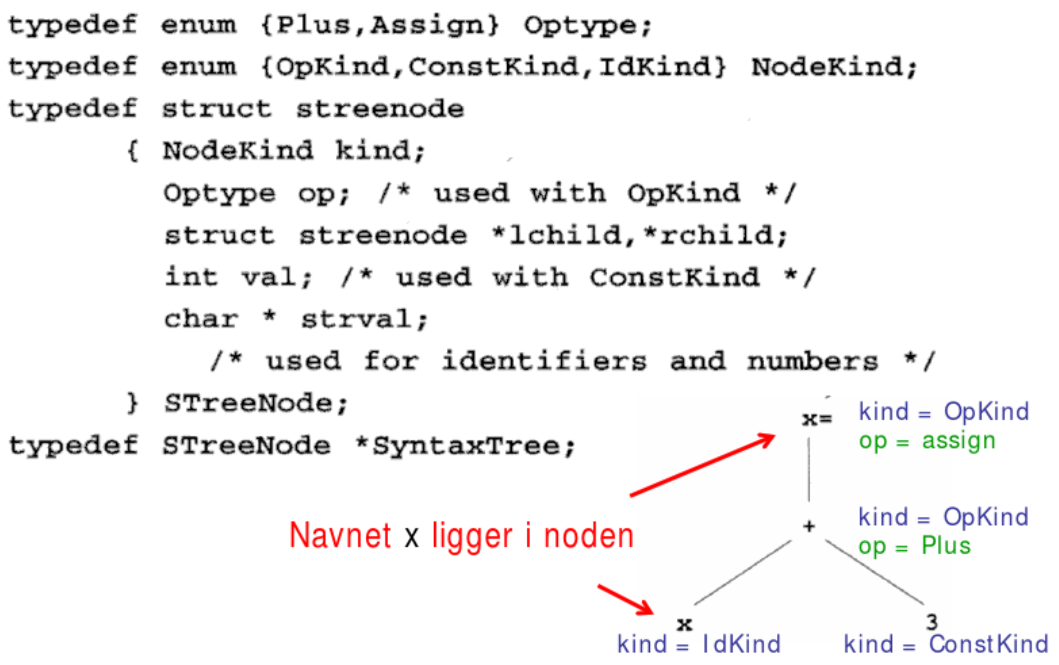
- symbols:
  - here: strings for *simplicity*
  - concretely, symbol table may be involved, or variable names already resolved in addresses etc.

Two-stage translation

```
val to_tree: Astexprassign.expr -> Pcode.tree

val linearize: Pcode.tree ->  Pcode.program

val to_program: Astexprassign.expr -> Pcode.program
```

```
let rec to_tree (e: expr) =
  match e with
  | Var s -> (Oneline (LOD s))
  | Num n  -> (Oneline (LDC n))
  | Plus (e1,e2) ->
      Seq (to_tree e1 ,
           Seq(to_tree e2, Oneline ADI))
  | Assign (x, e) ->
      Seq (Oneline (LDA x),
           Seq( to_tree e, Oneline STN))

let rec linearize (t: tree) : program =
  match t with
    Oneline i -> [i]
  | Seq (t1, t2) -> (linearize t1) @ (linearize t2);; // list concat

let to_program e = linearize (to_tree e);;
```

Source language AST data in C



```
typedef enum {Plus,Assign} Optype;
typedef enum {OpKind,ConstKind,IdKind} NodeKind;
typedef struct streenode
      { NodeKind kind;
        Optype op; /* used with OpKind */
        struct streenode *lchild,*rchild;
        int val; /* used with ConstKind */
        char * strval;
          /* used for identifiers and numbers */
      } STreeNode;
typedef STreeNode *SyntaxTree;
```

- remember though: there are more dignified ways to design ASTs …

Code-generation via tree traversal (schematic)

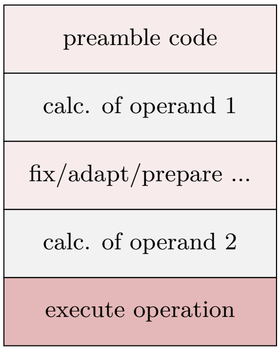
```
procedure genCode(T: treenode)
begin
 if   T $\not = $ nil
 then
   ``generate code to prepare for code for  left child''  // prefix
  genCode (left child of T);  // prefix ops
   ``generate code to prepare for code for right child''  //infix
   genCode (right child of T); // infix ops
   ``generate code to implement action(s) for T''  //postfix
end;
```

Code generation from AST$^+$

Text

- main “challenge”: linearization
- here: relatively simple
- no control-flow constructs
- linearization here (see a-grammar):
  - string of p-code
  - not necessarily the best choice (p-code might still need translation to “real” executable code)

Figure



preamble code

calc. of operand 1

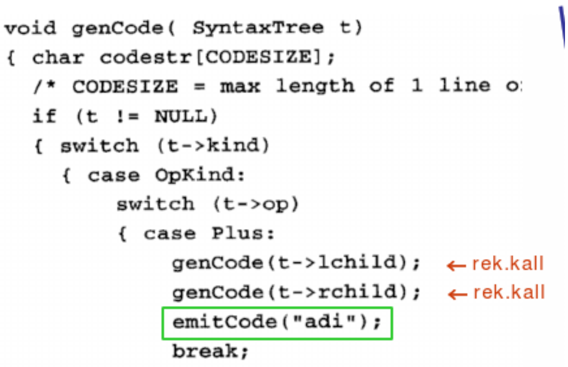
fix/adapt/prepare ...

calc. of operand 2

execute operation

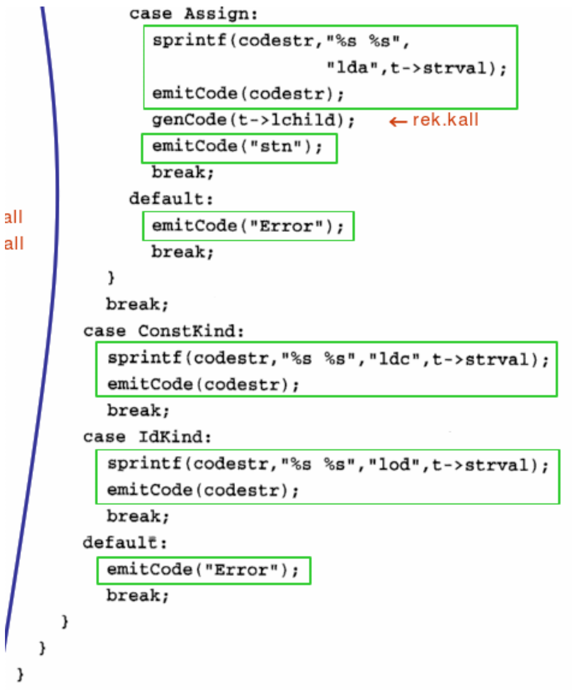
Code generation

First



```
void genCode( SyntaxTree t)
{ char codestr[CODESIZE];
  /* CODESIZE = max length of 1 line o
  if (t != NULL)
  { switch (t->kind)
    { case OpKind:
        switch (t->op)
        { case Plus:
            genCode(t->lchild);   ← rek.kall
            genCode(t->rchild);   ← rek.kall
            emitCode("adi");
            break;
```

Second



```
          case Assign:
            sprintf(codestr,"%s %s",
                           "lda",t->strval);
            emitCode(codestr);
            genCode(t->lchild);
            emitCode("stn");
            break;
          default:
            emitCode("Error");
            break;
        }
        break;
      case ConstKind:
        sprintf(codestr,"%s %s","ldc",t->strval);
        emitCode(codestr);
        break;
      case IdKind:
        sprintf(codestr,"%s %s","lod",t->strval);
        emitCode(codestr);
        break;
      default:
        emitCode("Error");
        break;
    }
  }
}
```

9.6 Generation of three address code

3AC manual translation again

Source

```
read x; {  input an integer }
if  0<x then
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0;
  write fact { output:
                factorial of x }
end
```

Target: 3AC

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Expression grammar again

Three-address code data structures (some)

Data structures (1)

```
type symbol = string

type expr =
  | Var of symbol
  | Num  of int
  | Plus of expr * expr
  | Assign of symbol * expr
```

Data structures (2)

```
type mem =
    Var of symbol
  | Temp of symbol
  | Addr of symbol   (* &x *)

type operand = Const of int
  | Mem  of mem

type cond = Bool of operand
  | Not of operand
  | Eq of operand * operand
  | Leq of operand * operand
  | Le of operand * operand

type rhs = Plus of operand * operand
  | Times of operand * operand
  | Id of operand

type instr =
    Read of symbol
  | Write of symbol
  | Lab of symbol      (* pseudo instruction *)
  | Assign of symbol * rhs
  | AssignRI of operand * operand * operand       (* a    := b[i] *)
  | AssignLI of operand * operand * operand       (* a[i] := b *)
  | BranchComp  of cond * label
  | Halt
  | Nop

type tree = Oneline of instr
  | Seq of tree * tree

type program = instr list


(* Branches are not so clear. I take inspiration first from ASU. It seems
that Louden has the TAC if_false t goto L. The Dragonbook allows actually
more complex structure, namely comparisons. However, two-armed branches are
not welcome (that would be a tree-IR) *)

(* Array access: For array accesses like a[i+1] = b[j] etc. one could add
   special commands. Louden indicates that, but also indicates that if one
   has indirect addressing and arithmetic operations, one does not need
   those. In the TAC of the dragon books, they have such operations, so I
   add them here as well.  Of course one sure not allow completely free
   forms like a[i+1] = b[j] in TAC, as this involves more than 3
   addresses. Louden suggests two operators, ``[]='' and ``[]=''.

   We could introduce more complex operands, like a[i] but then we would
   allow non-three address code things. We don't do that (of course, the
   syntax is already slightly too liberal...)


*)
```

Rest

- symbols: again strings for simplicity
- again "trees" not really needed (for simple language without more challenging control flow)

Translation to three-address code

```
let rec to_tree (e: expr) : tree * temp =
  match e with
    Var s ->  (Oneline Nop, s)
  | Num i ->  (Oneline Nop, string_of_int i)
  | Ast.Plus (e1,e2) ->
      (match (to_tree e1, to_tree e2) with
        ((c1,t1), (c2,t2)) ->
          let t = newtemp() in
          (Seq(Seq(c1,c2),
               Oneline (
               Assign (t,
                       Plus(Mem(Temp(t1)),Mem(Temp(t2)))))),
           t))
  | Ast.Assign (s',e') ->
      let (c,t2) = to_tree(e')
      in  (Seq(c,
               Oneline (Assign(s',
                               Id(Mem(Temp(t2)))))),
           t2)
```

Three-address code by synthesized attributes

- similar to the representation for p-code
- again: purely synthesized
- semantics of executing expressions/assignments[6]
    - side-effect plus also
    - value
- *two* attributes (before: only 1)
    - `tacode`: instructions (as before, as string), potentially empty
    - `name`: “name” of variable or tempary, where result resides[7]
- evaluation of expressions: *left-to-right* (as before)

A-grammar

productions/grammar rules			semantic rules		
exp_1	→	**id** = exp_2	exp_1 `.name`	=	exp_2 `.name`
			exp_1 `.tacode`	=	exp_2 `.tacode` ++ **id**`.strval` ^ "=" ^ exp_2 `.name`
exp	→	$aexp$	exp `.name`	=	$aexp$ `.name`
			exp `.tacode`	=	$aexp$ `.tacode`
$aexp_1$	→	$aexp_2$ + $factor$	$aexp_1$ `.name`	=	$newtemp()$
			$aexp_1$ `.tacode`	=	$aexp_2$ `.tacode` ++ $factor$ `.tacode` ++ $aexp_1$ `.name` ^ "=" ^ $aexp_2$ `.name` ^ "+" ^ $factor$ `.name`
$aexp$	→	$factor$	$aexp$ `.name`	=	$factor$ `.name`
			$aexp$ `.tacode`	=	$factor$ `.tacode`
$factor$	→	(exp)	$factor$ `.name`	=	exp `.name`
			$factor$ `.tacode`	=	exp `.tacode`
$factor$	→	**num**	$factor$ `.name`	=	**num**`.strval`
			$factor$ `.tacode`	=	""
$factor$	→	**id**	$factor$ `.name`	=	**num**`.strval`
			$factor$ `.tacode`	=	""

[6] That’s one possibility of a semantics of assignments (C, Java).

[7] In the p-code, the result of evaluating expression (also assignments) ends up in the stack (at the top). Thus, one does not need to capture it in an attribute.

Another sketch of TA-code generation

```
switch kind {
  case OpKind:
    switch op {
      case Plus: {
        tempname = new temorary name;
        varname_1 = recursive call on left subtree;
        varname_2 = recursive call on right subtree;
        emit ("tempname = varname_1 + varname_2");
        return (tempname);}
      case Assign: {
        varname = id. for variable on lhs (in the node);
        varname 1 = recursive call in left subtree;
        emit ("varname = opname");
        return (varname);}
      }
  case ConstKind; { return (constant-string);} // emit nothing
  case IdKind:  { return (identifier);}        // emit nothing
}
```

- "return" of the two attributes
  - name of the variable (a *temporary*): officially returned
  - the code: via *emit*
- note: *postfix* emission only (in the shown cases)

Generating code as AST methods

- possible: add `genCode` as *method* to the nodes of the AST
- e.g.: define an abstract method `String genCodeTA()` in the `Exp` class (or `Node`, in general all AST nodes where needed)

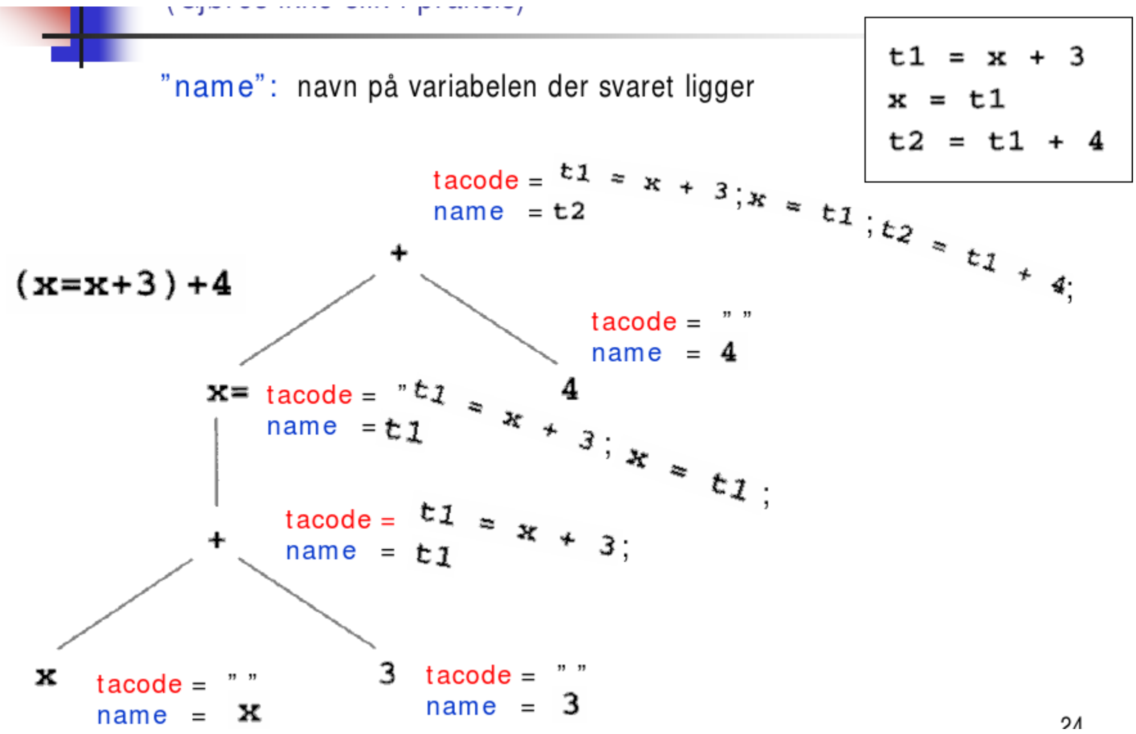
```
String genCodeTA() { String s1,s2; String t = NewTemp();
  s1 = left.GenCodeTA();
  s2 = right.GenCodeTA();
  emit (t + "="  + s1 + op + s2);
  return t
}
```

Whether it is a good design from the perspective of modular compiler architecture and code maintenance, to clutter the AST with methods for code generation and god knows what else, e.g. type checking, optimization ..., is a different question.

Translation to three-address code (from before)

```
let rec to_tree (e: expr) : tree * temp =
  match e with
    Var s ->  (Oneline Nop, s)
  | Num i ->  (Oneline Nop, string_of_int i)
  | Ast.Plus (e1,e2) ->
      (match (to_tree e1, to_tree e2) with
        ((c1,t1), (c2,t2)) ->
          let t = newtemp() in
          (Seq(Seq(c1,c2),
               Oneline (
               Assign (t,
                           Plus(Mem(Temp(t1)),Mem(Temp(t2)))))),
             t))
   | Ast.Assign (s',e') ->
       let (c,t2) = to_tree(e')
       in  (Seq(c,
                 Oneline (Assign(s',
                                 Id(Mem(Temp(t2)))))),
             t2)
```

Attributed tree (x=x+3) + 4



"name": navn på variabelen der svaret ligger

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

- note: room for optimization

9.7 Basic: From P-code to 3A-Code and back: static simulation & macro expansion

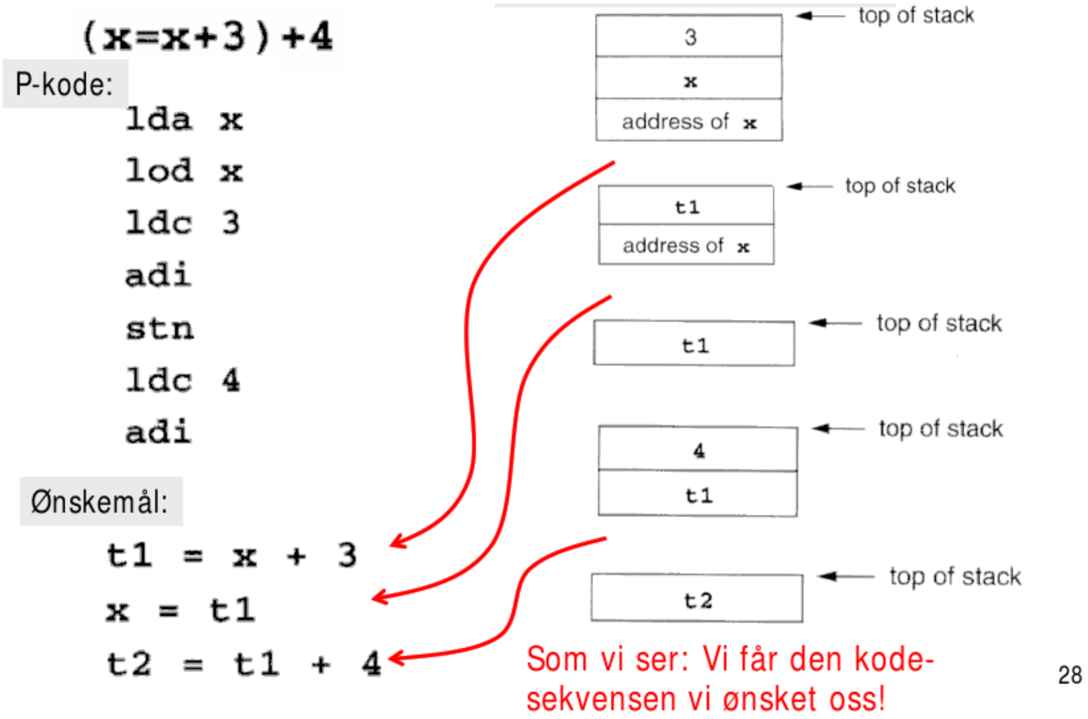
"Static simulation"

- *illustrated* by transforming P-code ⇒ 3AC
- restricted setting: straight-line code
- cf. also *basic blocks* (or elementary blocks)
  - code without branching or other control-flow complications (jumps/conditional jumps...)
  - often considered as basic building block for static/semantic analyses,
  - e.g. basic blocks as nodes in *control-flow graphs*, the "non-semicolon" control flow constructs result in the edges
- terminology: static simulation seems not widely established
- cf. *abstract interpretation*, *symbolic execution*, etc.

P-code ⇒ 3AIC via "static simulation"

- difference:
  - p-code operates on the *stack*
  - leaves the needed "temporary memory" implicit
- given the (straight-line) p-code:
  - traverse the code = list of instructions from beginning to end
  - seen as "simulation"
    - ∗ conceptually at least, but also
    - ∗ concretely: the translation can make *use* of an actual stack

From P-code ⇒ 3AIC: illustration



(x=x+3)+4

P-kode:

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

Ønskemål:

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

Som vi ser: Vi får den kode-sekvensen vi ønsket oss!

28

P-code ⇐ 3AIC: macro expansion

- also here: simplification, illustrating the general technique, only
- main simplification:
  - register allocation
  - but: better done in just another optmization "phase"

Macro for general 3AIC instruction: a = b + c

```
lda a
lod b;   or ``ldc b'' if b is a const
lod c:   or ``ldc c'' if c is a const
adi
sto
```

Example: P-code ⇐ 3AIC `((x=x+3)+4)`

Left

1. source 3A-code

```
t1 = x + 3
x  = t2
t2 = t1 + 4
```

2. Direct P-code

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi      ;  +
```

P-code via 3A-code by macro exp.

```
;--- t1 = x + 3
lda t1
lod x
ldc 3
adi
sto
;--- x = t1
lda x
lod t1
sto
;--- t2 = t1 + 4
lda t2
lod t1
ldc 4
adi
sto
```

Rest

cf. indirect 13 instructions vs. direct: 7 instructions

Indirect code gen: source code ⇒ 3AIC ⇒ p-code

- as seen: *detour* via 3AIC leads to sub-optimal results (code size, also efficiency)
- basic deficiency: too many *temporaries*, memory traffic etc.
- several possibilities
  - avoid it altogether, of course (but remember JIT in Java)
  - chance for *code optimization* phase
  - here: more clever “macro expansion” (but sketch only)

  the more clever macro expansion: some form of *static simulation* again

- don’t macro-expand the linear 3AIC
  - brainlessly into another *linear* structure (P-code), but
  - “statically simulate” it into a more *fancy* structure (a *tree*)

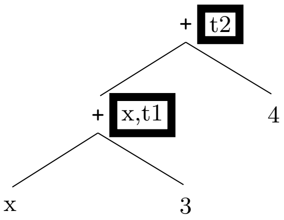
"Static simulation" into tree form (sketch)

- more fancy form of "static simulation" of 3AIC
- *result*: **tree** labelled with
  - operator, together with
  - variables/temporaries containing the results

Source

```
t1 = x + 3
x  = t2
t2 = t1 + 4
```

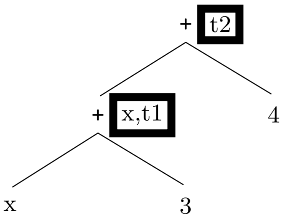
Tree



note: instruction `x = t1` from 3AIC: does *not* lead to more nodes in the tree

P-code generation from the generated tree

Tree from 3AIC



Direct code = indirect code

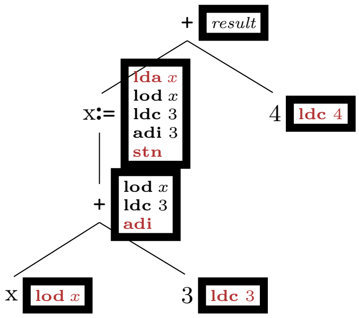
```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi     ; +
```

Rest

- with the thusly (re-)constructed tree

⇒ p-code generation
  - as before done for the AST
  - remember: code as synthesized attributes
- the "trick": reconstruct essential syntactic tree structure (via "static simulation") from the 3AI-code
- Cf. the macro expanded code: additional "memory traffic" (e.g. temp. t_1)

Compare: AST (with direct p-code attributes)



9.8 More complex data types

Status update: code generation

- so far: a number of simplifications
- data types:
  - integer constants only
  - no complex types (arrays, records, references, etc.)
- control flow
  - only expressions and
  - sequential composition

  ⇒ **straight-line code**

Address modes and address calculations

- so far
  - just standard "variables" (l-variables and r-variables) and temporaries, as in `x = x + 1`
  - variables referred to by their *names* (symbols)
- but in the end: variables are represented by *addresses*
- more complex *address calculations* needed

addressing modes in 3AIC:

- `&x`: *address* of `x` (not for temporaries!)
- `*t`: *indirectly* via `t`

addressing modes in P-code

- `ind i`: *indirect load*
- `ixa a`: *indexed address*

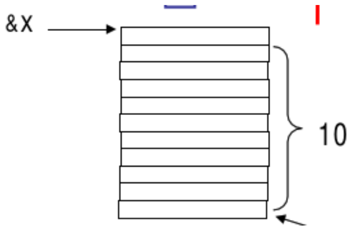
Address calculations in 3AIC: `x[10] = 2`

- notationally represented as in C
- "pointer arithmetic" and address calculation with the available numerical ops

Code

```
t1  = &x + 10
*t1 = 2
```

Picture

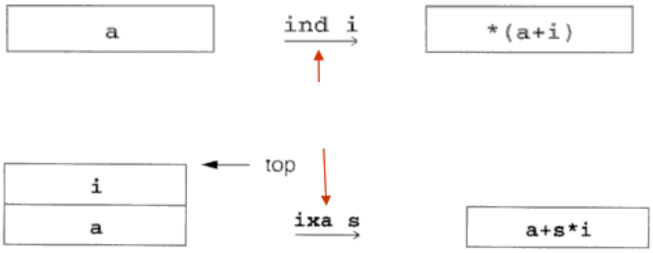


Rest

- 3-address-code data structure (e.g., quadrupel): *extended* (adding address mode)

Address calculations in P-code: `x[10] = 2`

- tailor-made commands for address calculation

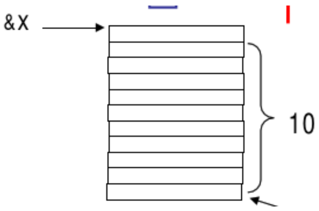


- `ixa i`: integer *scale* factor (here factor 1)

Code

```
lda x
ldc 10
ixa 1
ldc 2
sto
```

Picture



In the two pictures, the `a` is mnonic for a value representing an address. In the code example: The `ixa` command expects two argument on the stack (and has as third argument the scale factor as part of the command. To make use of the command, we first load the *address* of `x` loaded and afterwards constant `10`. Executing then the `ixa 1` command yields does the calculation in the box, which is intended as address calculation. So the result of that calculation is (intended as) an address again. To that address, the constant `2` is stored (and the values discared from the stack: `sto` is the "destructive" write).

Array references and address calculations

```
int a[SIZE]; int i,j;
a[i+1] = a[j*2] + 3;
```

- difference between left-hand use and right-hand use
- arrays: stored sequentially, starting at *base address*
- offset, calculated with a *scale factor* (dep. on size/type of elements)
- for example: for `a[i+1]` (with C-style array implementation)[8]

```
a + (i+1) * sizeof(int)
```

- `a` here *directly* stands for the base address

Array accesses in 3AI code

- *one* possible way: assume 2 additional 3AIC instructions
- remember: 3AIC can be seen as *intermediate code*, not as instruction set of a particular HW!
- 2 **new instructions**[9]

```
t2 = a[t1] ; fetch value of array element

a[t2] = t1 ; assign to the address of an array element
```

[8] In C, arrays start at a 0-offset as the first array index is 0. Details may differ in other languages.
[9] Still in 3AIC format. Apart from the "readable" notation, it's just two op-codes, say `=[]` and `[]=`.

Source code

```
a[i+1] = a[j*2] + 3;
```

TAC

```
t1     = j * 2
t2     = a[t1]
t3     = t2 + 3
t4     = i + 1
a[t4]  = t3
```

We have mentioned that IC is an intermediate representation that may be more or less closes to actual machine code. It's a design decision, and there are trade-offs either way. Like in this case: obviously it's (slightly) easier to translate array accesses to a 3AIC which offers such array accesses itself (like on this slide). It's, however, not too big a step to do the translation without this extra luxury. In the next following we see how to do without those array-accesses at the IC level (both for 3AIC as well as for P-code). That's done by macro-expansion, something that we touched upon earlier. The fact that one can "expand away" the extra commands show there are no real complications either way (with or without that extra expressivity).

One interesting aspect, though, is the use of the helper-function `elem_size`. Note that this depends on the type of the data structure (the elements of the array). It may also depend on the platform, which means, the function `elem_size` is (at the point of intermediate code generation) conceptually not yet available, but must provided and used when generating platform-dependent code. As similar "trick" we will see soon when compiling record-accesses (in the form of a function `field_offset`.

As a side remark: syntactic constract that can be expressed in that easy way, by forms of macro-expansion, are sometimes also called "syntactic sugar".

Or "expanded": array accesses in 3AI code (2)

Expanding `t2=a[t1]`

```
t3 = t1 * elem_size(a)
t4 = &a + t3
t2 = *t4
```

Expanding `a[t2]=t1`

```
t3  = t2 * elem_size(a)
t4  = &a + t3
*t4 = t1
```

Rest

- "expanded" result for `a[i+1] = a[j*2] + 3`

```
t1  = j * 2
t2  = t1 * elem_size(a)
t3  = &a + t2
t4  = *t3
t5  = t4 +3
t6  = i + 1
t7  = t6 * elem_size(a)
t8  = &a + t7
*t8 = t5
```

Array accessses in P-code

Expanding `t2=a[t1]`

```
lda t2
lda a
lod t1
ixa element_size(a)
ind 0
sto
```

Expanding `a[t2]=t1`

```
lda a
lod t2
ixa elem_size(a)
lod t1
sto
```

Rest

- "expanded" result for `a[i+1] = a[j*2] + 3`

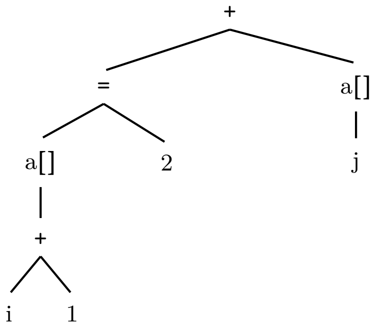
```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
lda a
lod j
ldc 2
mpi
ixa elem_size(a)
ind 0
ldc 3
adi
sto
```

Extending grammar & data structures

- extending the previous grammar

$$\begin{array}{rcl} \mathit{exp} & \rightarrow & \mathit{subs} = \mathit{exp}_2 \mid \mathit{aexp} \\ \mathit{aexp} & \rightarrow & \mathit{aexp} + \mathit{factor} \mid \mathit{factor} \\ \mathit{factor} & \rightarrow & (\, \mathit{exp}\,) \mid \mathbf{num} \mid \mathit{subs} \\ \mathit{subs} & \rightarrow & \mathbf{id} \mid \mathbf{id}\, [\, \mathit{exp}\,] \end{array}$$

Syntax tree for `(a[i+1]=2)+a[j]`



Code generation for P-code

The next slides show (as C code) how one could generate code for the “array access” grammar from before. Compared to the procedures for code generation before, the procedure has one additional argument, a boolean flag. That has to do with the disciction we want to make (here) whether the argument is to be interpeted as address or not. And that in turn is related between so called L-values and R-values and the fact that the grammar allows “assignments” (written `x = exp2`) to be expressions themsevlves. In the code generation, that is reflected also by the fact we use `stn` (non-destructive writing).

Otherwise: compare the code snippet from the earlier slides about “Array accesses in P-code”.

Code generation for P-code (op)

```
void genCode (SyntaxTree t, int isAddr)  {
  char codestr[CODESIZE];
  /* CODESIZE = max length of 1 line of P-code */
  if (t != NULL) {
    switch (t->kind) {
    case OpKind:
      { switch (t->op) {
        case Plus:
          if (isAddress) emitCode("Error");  // new check
          else {                             // unchanged
            genCode(t->lchild,FALSE);
            genCode(t->rchild,FALSE);
            emitCode("adi");                 // addition
          }
          break;
        case Assign:
          genCode(t->lchild,TRUE);          //``l-value''
          genCode(t->rchild,FALSE);         //``r-value''
          emitCode("stn");
```

Code generation for P-code ("subs")

- new code, of course

```
case Subs:
  sprintf(codestring,"%s %s", "lda",t->strval);
  emitCode(codestring);
  genCode(t->lchild. FALSE);
  sprintf(codestring,"%s %s %s",
          "ixa elem_size(", t->strval,")");
  emitCode(codestring);
  if (!isAddr) emitCode("ind 0");  // indirect load
  break;
default:
  emitCode("Error");
  break;
```

Code generation for P-code (constants and identifiers)

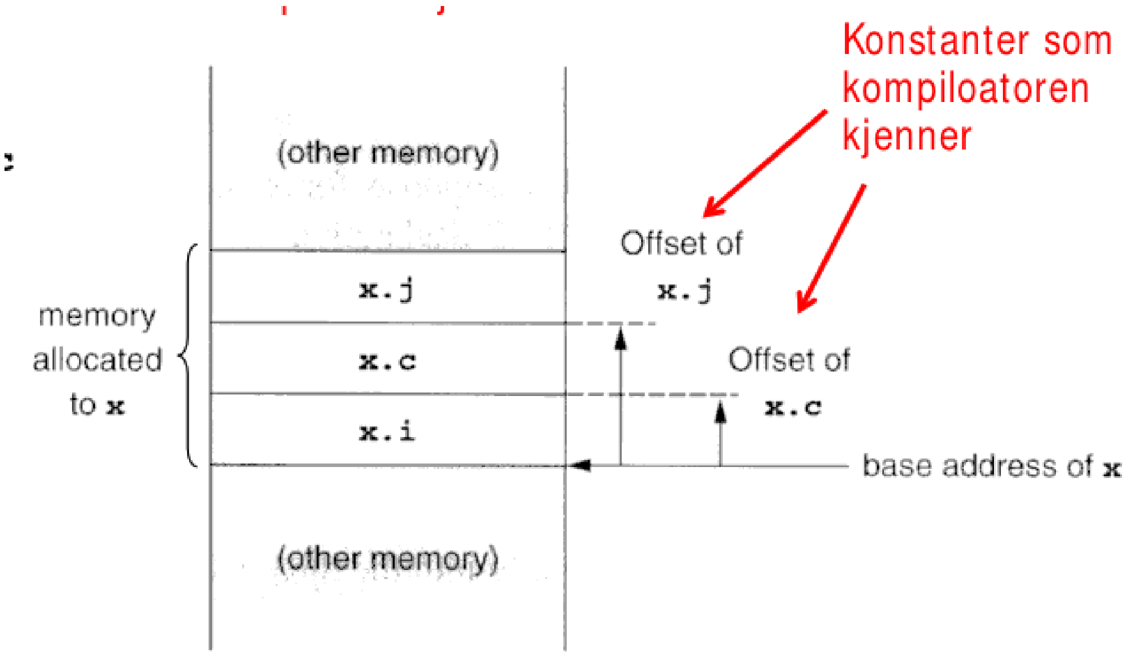
```
    case ConstKind:
      if (isAddr) emitCode("Error");
      else {
        sprintf(codestr,"%s %s","lds",t->strval);
        emitCode(codestr);
      }
      break;
    case IdKind:
      if (isAddr)
        sprintf(codestr,"%s %s", "lda",t->strval);
      else
        sprintf(codestr,"%s %s", "lod",t->strval);
      emitCode(codestr);
      break;
    default:
      emitCode("Error");
      break;
    }
  }
}
```

Access to records

C-Code

```
typedef struct rec {
  int i;
  char c;
  int j;
} Rec;
...

Rec x;
```

Layout



Rest

- fields with (statically known) offsets from base address
- note:
  - goal: intermediate code generation *platform independent*
  - another way of seeing it: it's still IR, not *final* machine code yet.
- thus: introduce function `field_offset(x,j)`
- calculates the offset.
- can be looked up (by the code-generator) in the *symbol table*

⇒ call replaced by actual off-set

Records/structs in 3AIC

- note: typically, records are implicitly references (as for objects)
- in (our version of a) 3AIC: we can just use `&x` and `*x`

simple record access `x.j`

```
t1 = &x +  field_offset(x,j)
```

left and right: `x.j = x.i`

```
t1  = &x + field_offset(x,j)
t2  = &x + field_offset(x,i)
*t1 = *t2
```

Field selection and pointer indirection in 3AIC

C code

```
typedef struct treeNode {
  int val;
  struct treeNode * lchild,
                  * rchild;
} treeNode
...

Treenode *p;
```

Assignment involving fields

```
p -> lchild = p;
p           = p->rchild;
```

1. 3AIC

```
t1   = p + field_access(*p,lchild)
*t1  = p
t2   = p + field_access(*p,rchild)
p    = *t2
```

Structs and pointers in P-code

- basically same basic "trick"
- make use of `field_offset(x,j)`

3AIC

```
p -> lchild = p;
p           = p->rchild;
```

```
lod p
ldc field_offset(*p, lchild)
ixa 1
lod p
sto
lda p
lod p
ind field_offset(*p, rchild)
sto
```

9.9 Control statements and logical expressions

So far, we have dealt with straight-line code only. The main "complication" were compound expression, which do not exist in the intermediate code, neither in 3AIC nor in the p-code. That reqired the introduction of temporaries resp. the use of the stack to store those intermediate results. The core addition to deal with control statements is the use of *labels*. Labels can be seen as "symbolic" respresentations of "programming lines" or "control points". Ultimately, in the final binary, the platform will support jumps and conditional jumps which will "transfer" control (= program pointer) from one address to another, "jumping to an address". Since we are still at an intermediate code level, we do jumps not to real addressed but to labels (referring to the starting point of seqquences of intermediate code). As a side remark: also assembly language editors will in general support *labels* the assembly programmer can use to make the program at least a bit more human-readable (and relocatable). Labels and *goto* statements are also known in (not-so-)high-level languages such as classic Basic (and even Java has `goto` as reserved word, even if it makes no use of it).

Besides the treatment of control constructs, we discuss a related issue namely a particular use of boolean expression. It's discussed here as well, as (in some languages) boolean expression can behave as control-constructs, as well. Consequently, the translation of that form of booleans, require similar mechanisms (labels) as the translation of standard-control statements. In C-like languages, that's know as short-circuiting.

As a not-so-important side remark: Concretely in C, "booleans" and conditions operate also on more than just a boolean two valued domain (containting `true` and `false` or `0` and `1`). In C, "everything" that's not `0` is treated as `1`. That may sounds not too "logical" but reflects how some hardware instructions and conditional jumps work. Doing some operations sets " hardware flags" which then are used for conditional jumps: jump-on-zero checks whether the corresponds flag is set accordingly. Furthermore, in functional languges, the phenomenon also occurs (but typically not called short-circuiting), and in general there, the dividing line between control and data is blurred anyway.

Control statements

- so far: basically *straight-line code*
- general (intra-procedural) control more complex thanks to *control-statements*
  - conditionals, switch/case
  - loops (while, repeat, for ...)
  - breaks, gotos, exceptions ...

important "technical" device: labels

- symbolic representation of addresses in static memory
- specifically named (= labelled) control flow points
- nodes in the *control flow graph*
- generation of labels (cf. also temporaries)

Intra-procedural means "inside" a procedure. *Inter*-procedural control-flow refers to calls and returns, which is handled by calling sequences (which also maintain (in standard C-like languages) the call-stack of the RTE).

Concerning gotos: gotos (if the language supports them) are almost trivial in code generation, as they are basically available at machine code level. Nonetheless, they are "considered harmful", as they mess up/break abstractions and other things in a compiler/language.

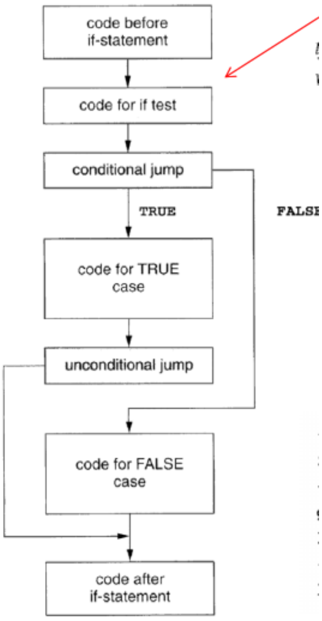
Loops and conditionals: linear code arrangement

$$\begin{array}{rcl} \textit{if-stmt} & \rightarrow & \textbf{if}\,\textbf{(}\,\textit{exp}\,\textbf{)}\,\textit{stmt}\ \textbf{else}\ \textit{stmt} \\ \textit{while-stmt} & \rightarrow & \textbf{while}\,\textbf{(}\,\textit{exp}\,\textbf{)}\,\textit{stmt} \end{array}$$

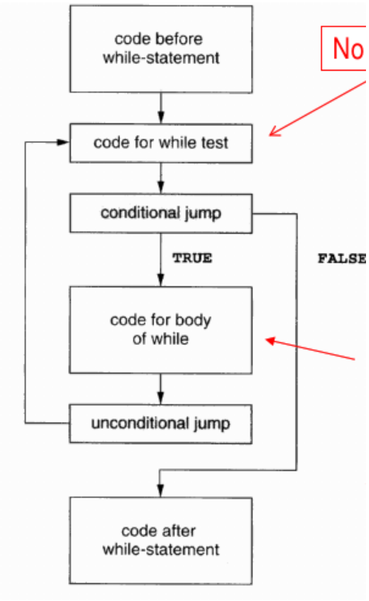
- challenge:
  - high-level syntax (AST) well-structured (= tree) which implicitly (via its structure) determines complex control-flow beyond SLC
  - low-level syntax (3AIC/P-code): rather flat, linear structure, ultimately just a *sequence* of commands

Arrangement of code blocks and cond. jumps

Conditional



While



The “graphical” representation can also be understood as *control flow graph*. The nodes contain sequences of “basic statements” of the form we covered before (like one-line 3AIC assignments) but not conditionals and similar and no procedure calls (we don’t cover them in the chapter anyhow). So the nodes (also known as *basic blocks*) contain staight-line code.

In the following we show how to translate conditionals and while statements into intermediate code, both for 3AIC and p-code. The translation is rather straightforward (and actually very similar for both cases, both making use of labels).

To do the translation, we need to enhance the set of available "op-codes" (= available commands). We need a mechanism for *labelling* and a mechanism for *conditional jumps*. Both kind of statement need to be added to 3AIC and p-code, and it basically works the same, except that the actual syntax of the commands is different. But that's details.

Jumps and labels: conditionals

if ($E$) **then** $S_1$ **else** $S_2$

3AIC for conditional

```
<code to eval $E$ to t1>
if_false t1 goto L1
<code for $S_1$>
goto L2
label L1
<code for $S_2$>
label L2
```

P-code for conditional

```
<code to evaluate $E$>
fjp L1
<code for $S_1$>
ujp L2
lab L1
<code for S2>
lab L2
```

3 new op-codes:

- **ujp**: unconditional jump ("goto")
- **fjp**: jump on false
- **lab**: label (for pseudo instructions)

Jumps and labels: while

while ($E$) S

3AIC for while

```
label L1
<code to evaluate $E$ to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2
```

P-code for while

```
lab L1
<code to evaluate $E$>
fjp L2
<code for $S$>
ujp L1
lab L2
```

Boolean expressions

- two alternatives for treatment
  1. as *ordinary* expressions
  2. via *short-circuiting*
- ultimate representation in HW:
  - no built-in booleans (HW is generally untyped)
  - but "arithmetic" 0, 1 work equivalently & fast
  - bitwise ops which corresponds to logical $\wedge$ and $\vee$ etc
- comparison on "booleans": $0 < 1$?
- boolean values vs. jump conditions

Short circuiting boolean expressions

Short circuit illustration

```
if ((p!=NULL) && p -> val==0)) ...
```

- done in C, for example
- semantics must *fix* evaluation order
- note: logically equivalent $a \wedge b = b \wedge a$
- cf. to conditional expressions/statements (also left-to-right)

$$\begin{aligned} a \textbf{ and } b &\triangleq \textbf{if } a \textbf{ then } b \textbf{ else false} \\ a \textbf{ or } b &\triangleq \textbf{if } a \textbf{ then true else } b \end{aligned}$$

Pcode

```
lod x
ldc 0
neq      ; x!=0 ?
fjp L1   ; jump, if x=0
lod y
lod x
equ      ;  x =? y
ujp L2   ; hop over
lab L1
ldc FALSE
lab L2
```

- new op-codes
  - **equ**
  - **neq**

The code is a bit cryptic (one should ponder what it computes ...). It might not be also the best represetation, for instance, one may come up with a different solution that does *not* load x two times.

A side remark: we are still at intermediate code. Optimizations and the use of registers have not yet entered the picture. That is to say, that the above remark that x is loaded two times might be of not so much concern ultimately, as an optimizer and register allocator should be able to do something about it. On the other hand: why generate inefficient code in the hope the optimizer will clean it up.

Grammar for loops and conditionals

$$\begin{array}{rcl} \textit{stmt} & \rightarrow & \textit{if-stmt} \mid \textit{while-stmt} \mid \textbf{break} \mid \textbf{other} \\ \textit{if-stmt} & \rightarrow & \textbf{if}\,(\,\textit{exp}\,)\,\textit{stmt}\ \textbf{else}\ \textit{stmt} \\ \textit{while-stmt} & \rightarrow & \textbf{while}\,(\,\textit{exp}\,)\,\textit{stmt} \\ \textit{exp} & \rightarrow & \textbf{true} \mid \textbf{false} \end{array}$$

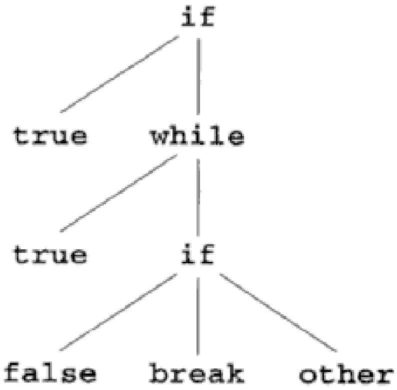
- note: simplistic expressions, only *true* and *false*

```
typedef enum {ExpKind, Ifkind, Whilekind,
             BreakKind, OtherKind} NodeKind;

typedef struct streenode {
  NodeKind kind;
  struct streenode * child[3];
  int val; /* used with ExpKind */
           /* used for true vs. false */
} STreeNode;

type StreeNode * SyntaxTree;
```

Translation to P-code

```
if (true) while (true) if (false) break else other
```

Syntax tree



P-code

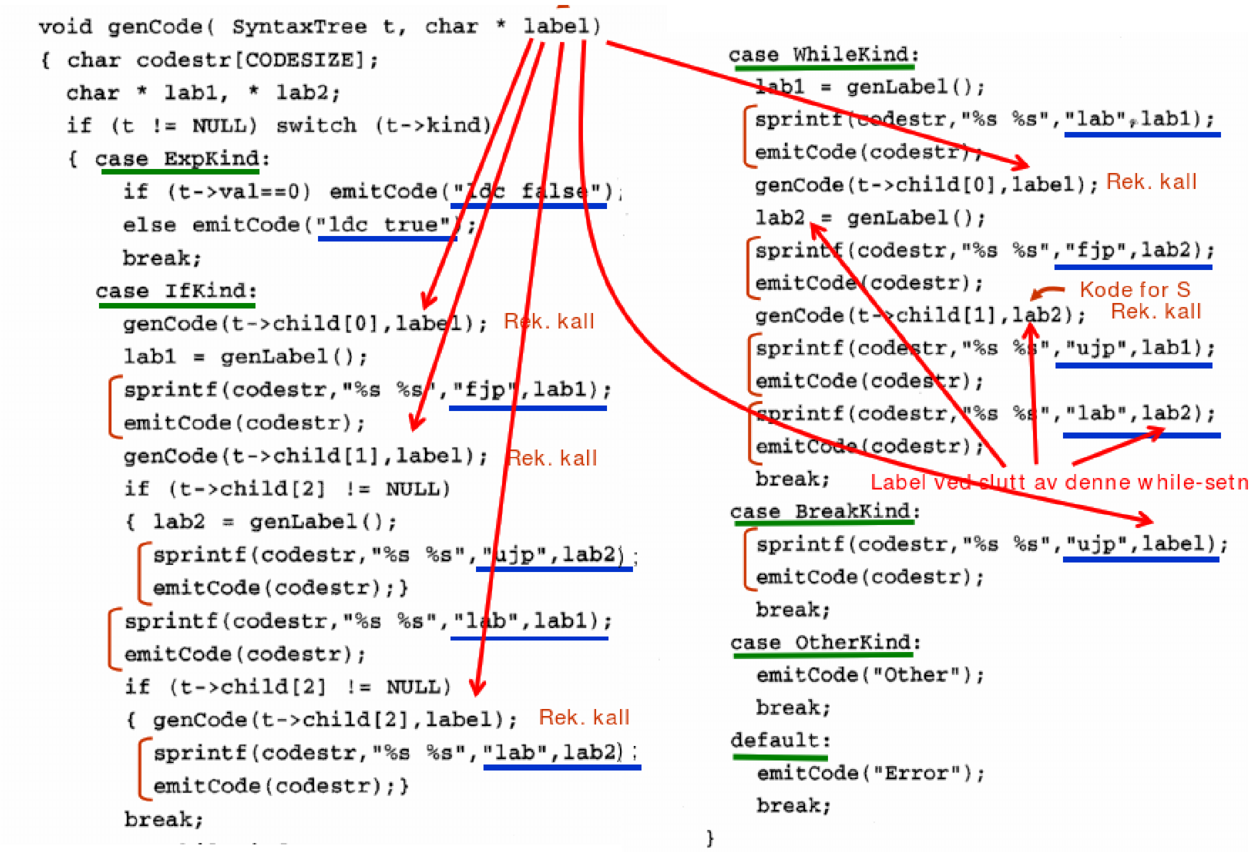
```
ldc true
fjp L1
lab L2
ldc true
fjp L3
ldc false
fjp L4
ujp L3
ujp L5
lab L4
Other
lab L5
ujp L2
lab L3
lab L1
```

Code generation

- extend/adapt `genCode`
- **break** statement:
  - absolute *jump* to *place afterwards*
  - *new argument*: label to jump-to when hitting a break
- assume: *label generator* `genLabel()`
- case for if-then-else
  - has to deal with one-armed if-then as well: test for NULL-ness

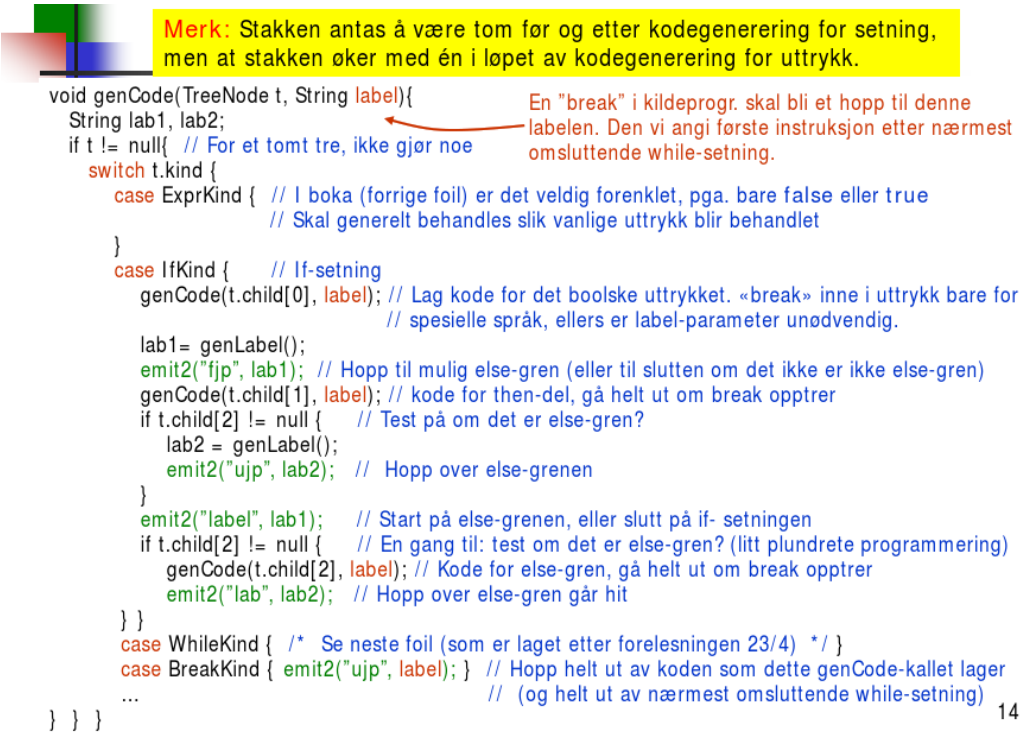
- side remark: **control-flow graph** (see also later)
  - labels can (also) be seen as *nodes* in the *control-flow graph*
  - `genCode` generates labels while traversing the AST
  - ⇒ implict generation of the CFG
  - also possible:
    - ∗ separately generate a CFG first
    - ∗ as (just another) IR
    - ∗ generate code from there

Code generation procedure for P-code



```
void genCode( SyntaxTree t, char * label)
{ char codestr[CODESIZE];
  char * lab1, * lab2;
  if (t != NULL) switch (t->kind)
  { case ExpKind:
        if (t->val==0) emitCode("ldc false");
        else emitCode("ldc true");
        break;
    case IfKind:
        genCode(t->child[0],label);
        lab1 = genLabel();
        sprintf(codestr,"%s %s","fjp",lab1);
        emitCode(codestr);
        genCode(t->child[1],label);
        if (t->child[2] != NULL)
        { lab2 = genLabel();
          sprintf(codestr,"%s %s","ujp",lab2);
          emitCode(codestr);}
        sprintf(codestr,"%s %s","lab",lab1);
        emitCode(codestr);
        if (t->child[2] != NULL)
        { genCode(t->child[2],label);
          sprintf(codestr,"%s %s","lab",lab2);
          emitCode(codestr);}
        break;
    case WhileKind:
      lab1 = genLabel();
      sprintf(codestr,"%s %s","lab",lab1);
      emitCode(codestr);
      genCode(t->child[0],label);
      lab2 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab2);
      emitCode(codestr);
      genCode(t->child[1],lab2);
      sprintf(codestr,"%s %s","ujp",lab1);
      emitCode(codestr);
      sprintf(codestr,"%s %s","lab",lab2);
      emitCode(codestr);
      break;
    case BreakKind:
      sprintf(codestr,"%s %s","ujp",label);
      emitCode(codestr);
      break;
    case OtherKind:
      emitCode("Other");
      break;
    default:
      emitCode("Error");
      break;
}
```

Code generation (1)



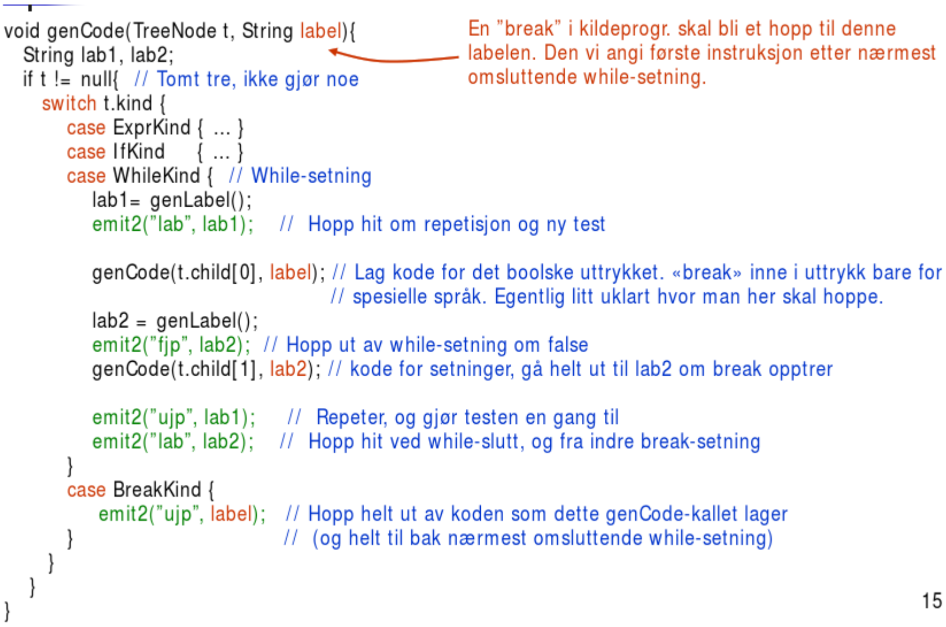
Merk: Stakken antas å være tom før og etter kodegenerering for setning, men at stakken øker med én i løpet av kodegenerering for uttrykk.

En "break" i kildeprogr. skal bli et hopp til denne labelen. Den vi angi første instruksjon etter nærmest omsluttende while-setning.

```
void genCode(TreeNode t, String label){
   String lab1, lab2;
   if t != null{  // For et tomt tre, ikke gjør noe
     switch t.kind {
        case ExprKind {  // I boka (forrige foil) er det veldig forenklet, pga. bare false eller true
                         // Skal generelt behandles slik vanlige uttrykk blir behandlet
        }
        case IfKind {      // If-setning
           genCode(t.child[0], label); // Lag kode for det boolske uttrykket. «break» inne i uttrykk bare for
                                       // spesielle språk, ellers er label-parameter unødvendig.
           lab1= genLabel();
           emit2("fjp", lab1);  // Hopp til mulig else-gren (eller til slutten om det ikke er ikke else-gren)
           genCode(t.child[1], label); // kode for then-del, gå helt ut om break opptrer
           if t.child[2] != null {     // Test på om det er else-gren?
              lab2 = genLabel();
              emit2("ujp", lab2);   //  Hopp over else-grenen
           }
           emit2("label", lab1);     // Start på else-grenen, eller slutt på if- setningen
           if t.child[2] != null {     // En gang til: test om det er else-gren? (litt plundrete programmering)
              genCode(t.child[2], label); // Kode for else-gren, gå helt ut om break opptrer
              emit2("lab", lab2);   // Hopp over else-gren går hit
        } }
        case WhileKind {  /*  Se neste foil (som er laget etter forelesningen 23/4)  */ }
        case BreakKind {  emit2("ujp", label); }  // Hopp helt ut av koden som dette genCode-kallet lager
        ...                                        //  (og helt ut av nærmest omsluttende while-setning)
} } }
```

14

Code generation (2)



En "break" i kildeprogr. skal bli et hopp til denne labelen. Den vi angi første instruksjon etter nærmest omsluttende while-setning.

```
void genCode(TreeNode t, String label){
   String lab1, lab2;
   if t != null{  // Tomt tre, ikke gjør noe
     switch t.kind {
        case ExprKind { ... }
        case IfKind     { ... }
        case WhileKind {  // While-setning
           lab1= genLabel();
           emit2("lab", lab1);    //  Hopp hit om repetisjon og ny test

           genCode(t.child[0], label); // Lag kode for det boolske uttrykket. «break» inne i uttrykk bare for
                                       // spesielle språk. Egentlig litt uklart hvor man her skal hoppe.
           lab2 = genLabel();
           emit2("fjp", lab2);  // Hopp ut av while-setning om false
           genCode(t.child[1], lab2); // kode for setninger, gå helt ut til lab2 om break opptrer

           emit2("ujp", lab1);     //  Repeter, og gjør testen en gang til
           emit2("lab", lab2);    //  Hopp hit ved while-slutt, og fra indre break-setning
        }
        case BreakKind {
            emit2("ujp", label);    // Hopp helt ut av koden som dette genCode-kallet lager
        }                           //  (og helt til bak nærmest omsluttende while-setning)
     }
   }
}
```

15

More on short-circuiting (now in 3AIC)

- boolean expressions contain only two (official) values: true and false

- as stated: boolean expressions are often treated special: via short-circuiting
- short-circuiting especially for boolean expressions in *conditionals* and *while*-loops and similar
  - treat boolean expressions *different* from ordinary expressions
  - avoid (if possible) to calculate boolean value “till the end”
- short-circuiting: specified in the language definition (or not)

Example for short-circuiting

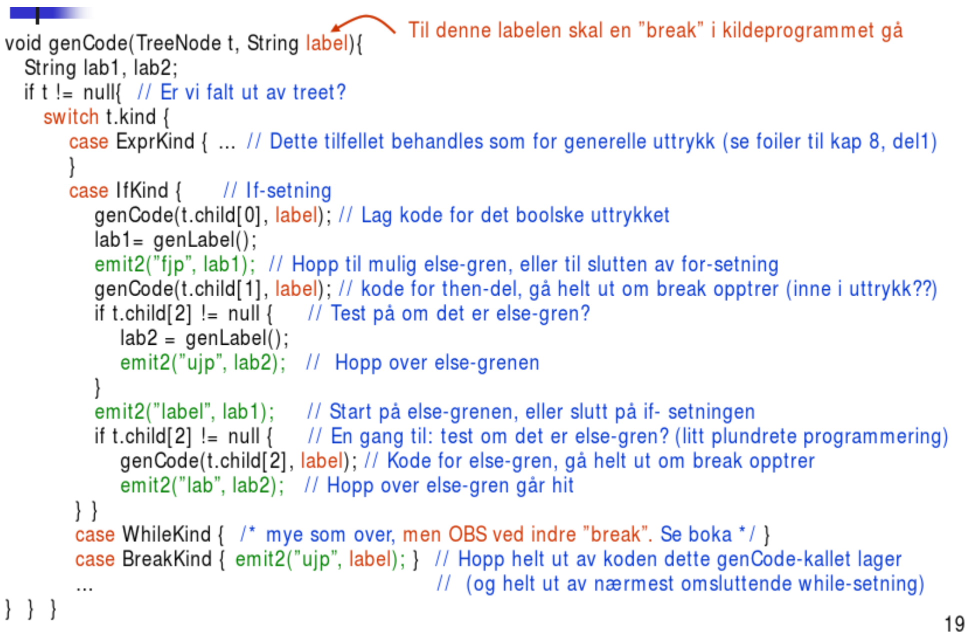
Source

```
if a < b ||
   (c > d && e >= f)
then
  x = 8
else
  y = 5
endif
```

3AIC

```
t1 = a < b
if_true t1 goto 1 // short circuit
t2 = c > d
if_false goto 2   // short circuit
t3 = e >= f
if_false t3 goto 2
label 1
x = 8
goto 3
label 2
y = 5
label 3
```

Code generation: conditionals (as seen)



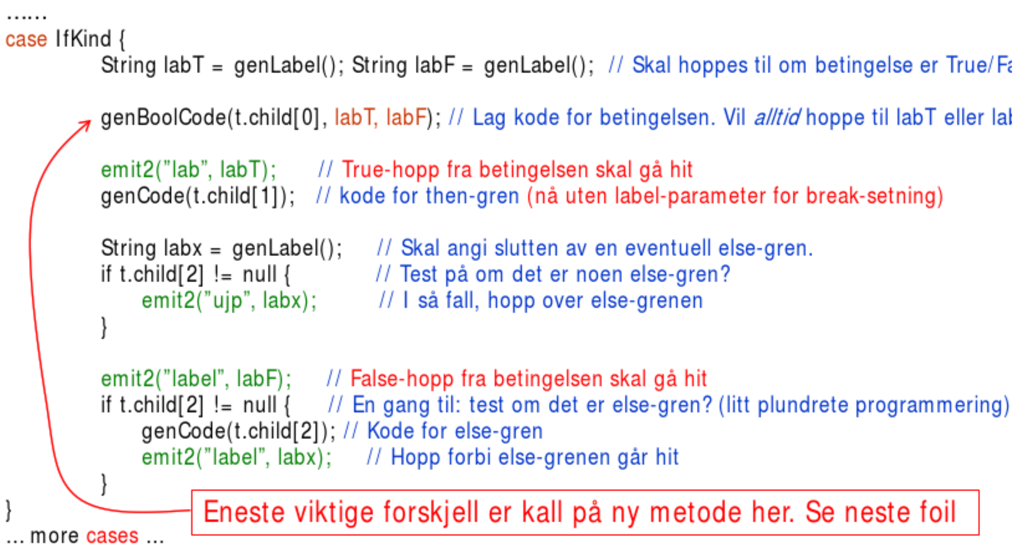
Til denne labelen skal en "break" i kildeprogrammet gå

```
void genCode(TreeNode t, String label){
  String lab1, lab2;
  if t != null{  // Er vi falt ut av treet?
    switch t.kind {
      case ExprKind { ... // Dette tilfellet behandles som for generelle uttrykk (se foiler til kap 8, del1)
      }
      case IfKind {      // If-setning
         genCode(t.child[0], label); // Lag kode for det boolske uttrykket
         lab1= genLabel();
         emit2("fjp", lab1);  // Hopp til mulig else-gren, eller til slutten av for-setning
         genCode(t.child[1], label); // kode for then-del, gå helt ut om break opptrer (inne i uttrykk??)
         if t.child[2] != null {     // Test på om det er else-gren?
            lab2 = genLabel();
            emit2("ujp", lab2);   //  Hopp over else-grenen
         }
         emit2("label", lab1);     // Start på else-grenen, eller slutt på if- setningen
         if t.child[2] != null {     // En gang til: test om det er else-gren? (litt plundrete programmering)
            genCode(t.child[2], label); // Kode for else-gren, gå helt ut om break opptrer
            emit2("lab", lab2);   // Hopp over else-gren går hit
      } }
      case WhileKind {  /* mye som over, men OBS ved indre "break". Se boka */ }
      case BreakKind { emit2("ujp", label); }  // Hopp helt ut av koden dette genCode-kallet lager
      ...                                       //  (og helt ut av nærmest omsluttende while-setning)
} } }
```

19

Alternative P/3A-Code generation for conditionals

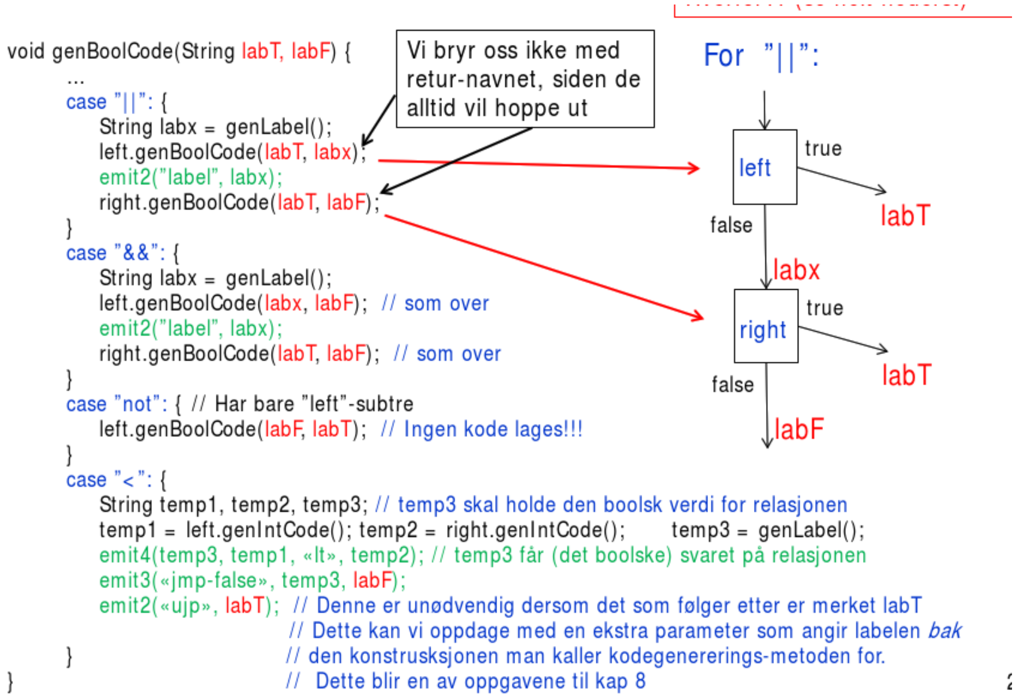
- Assume: **no break** in the language for simplicity
- focus here: conditionals
- not covered of [9]



```
......
case IfKind {
        String labT = genLabel(); String labF = genLabel();  // Skal hoppes til om betingelse er True/Fa

        genBoolCode(t.child[0], labT, labF); // Lag kode for betingelsen. Vil alltid hoppe til labT eller lal

        emit2("lab", labT);      // True-hopp fra betingelsen skal gå hit
        genCode(t.child[1]);   // kode for then-gren (nå uten label-parameter for break-setning)

        String labx = genLabel();     // Skal angi slutten av en eventuell else-gren.
        if t.child[2] != null {           // Test på om det er noen else-gren?
            emit2("ujp", labx);         // I så fall, hopp over else-grenen
        }

        emit2("label", labF);     // False-hopp fra betingelsen skal gå hit
        if t.child[2] != null {      // En gang til: test om det er else-gren? (litt plundrete programmering)
            genCode(t.child[2]); // Kode for else-gren
            emit2("label", labx);     // Hopp forbi else-grenen går hit
        }
}
... more cases ...
```

Eneste viktige forskjell er kall på ny metode her. Se neste foil

Alternative 3A-Code generation for boolean expressions



```
void genBoolCode(String labT, labF) {
        ...
        case "||": {
            String labx = genLabel();
            left.genBoolCode(labT, labx);
            emit2("label", labx);
            right.genBoolCode(labT, labF);
        }
        case "&&": {
            String labx = genLabel();
            left.genBoolCode(labx, labF);  // som over
            emit2("label", labx);
            right.genBoolCode(labT, labF);  // som over
        }
        case "not": { // Har bare "left"-subtre
            left.genBoolCode(labF, labT);  // Ingen kode lages!!!
        }
        case "<": {
            String temp1, temp2, temp3; // temp3 skal holde den boolsk verdi for relasjonen
            temp1 = left.genIntCode(); temp2 = right.genIntCode();      temp3 = genLabel();
            emit4(temp3, temp1, «lt», temp2); // temp3 får (det boolske) svaret på relasjonen
            emit3(«jmp-false», temp3, labF);
            emit2(«ujp», labT);  // Denne er unødvendig dersom det som følger etter er merket labT
                                 // Dette kan vi oppdage med en ekstra parameter som angir labelen bak
        }                        // den konstruksjonen man kaller kodegenererings-metoden for.
}                                //  Dette blir en av oppgavene til kap 8
```

Vi bryr oss ikke med retur-navnet, siden de alltid vil hoppe ut

Chapter
Code generation

What is it about?

Learning Targets of this Chapter

1. 2AC
2. cost model
3. register allocation
4. control-flow graph
5. local liveness analysis (data flow analysis)
6. “global” liveness analysis

Contents

10.1 Intro

Code generation

- note: *code generation* so far: AST$^+$ to **intermediate code**
  - three address intermediate code (3AIC)
  - P-code
- ⇒ *intermediate code generation*
- i.e., we are still not there ...
- material here: based on the (old) *dragon book* [2] (but principles still ok)
- there is also a new edition [1]

This section is based on slides from Stein Krogdahl, 2015. In this section we work with 2AC as machine code (as from the older, classical "dragon book"). An alternative would be 3AC also on code level (not just intermediate code); details would change, but the principles would be comparable. Note: the message of the chapter is *not*: in the last translation and code generation step, one has to find a way to translate 3-address code two 2-address code. If one assumed machine code in a 3-address format, the principles would be similar. The core of the code generation is the (here rather simple) treatment of *registers.* The code generation and register allocation presented here is rather straightforward; it will look "detailed" and "complicated", but it's not very complex in that the optimization puts very much computational effort into the code generation. One optimization done is is based on liveness analysis. An occurrence of a variable is "dead", if the variable will not be read in the future (unless it's first overwritten). The opposite concept is that the occurrence of a variable is live. It should be obvious that this kind of information is essential for making good decisions for register allocation. The general problem there is: we have typically less registers than variables and temps. So the compiler must make a section: who should be in a register and who not? A static scheme like "the first variables in, say, alphabetical order, should be in registers, the others not" is not worth being called optimization... First-come-first-serve like "if I need a variable, I load it to a registers, if there is still some free, otherwise not" is not much better. Basically, what is missing is taking into account information when a variable is no longer used (when no longer life), thereby figuring out, at which point a register can be considered free again. Note that we are not talking about run-time, we are talking about code generation, i.e., compile time. The code generator must generate instructions that loads variables to registers it has figured out to be free (again). The code generator therefore needs to keep track over the free and occupied registers; more precisely, it needs to keep track of which variable is contained in which register, resp. which register contains which variable. Actually, in the code generation later, it can even happen that one register contains the values of more than one variable. Based on such a book-keeping the code generation must also make decisions like the following: if a value needs to be read from main memory and is intended to be in a register but all of them are full, which register should be "purged". As far as the last question is concerned, the lecture will not drill deep. We will concentrate on liveness analysis and we will do that in two stages: a block-local one and a global one. the local one concentrates on one basic block, i.e., one block of straight-line code. That makes the code generation kind of like what had been called "static simulation" before. In particular, the liveness information is *precise* (inside the block): the code generator knows at each point which variables are live (i.e., will be used in the rest of the block) and which not. When going to a *global* liveness analysis, that precision is no longer doable, and one goes for an approximative approach. The treatment there is *typical* for data flow analysis. There are *many* data flow analyses, for different purposes, but we only have a look at liveness analysis with the purpose of optimizing register allocation.

Intro: code generation

- goal: translate intermediate code (= 3AI-code) to machine language
- machine language/assembler:
  - even *more* restricted
  - here: 2 address code
- limited number of *registers*
- different *address modes* with different *costs* (registers vs. main memory)

Goals

- **efficient** code
- small code size also desirable
- but first of all: **correct** code

When not said otherwise: efficiency refers in the following to efficiency of the generated code. Fastness of compilation may be important, as well (and same for the size of the compiler itself, as opposed to the size of the generated code). Obviously, there are trade-offs to be made.

Code "optimization"

- often conflicting goals
- code generation: *prime* arena for achieving *efficiency*
- **optimal code**: undecidable anyhow (and: don't forget there's trade-offs).
- even for many more clearly defined subproblems: *untractable*

"optimization"

interpreted as: *heuristics* to achieve "good code" (without hope for *optimal* code)

- due to importance of optimization at code generation
  - time to bring out the "heavy artillery"
  - so far: all techniques (parsing, lexing, even sometimes type checking) are computationally "easy"
  - at code generation/optimization: perhaps *invest* in aggressive, computationally complex and rather advanced techniques
  - **many** different techniques used

The above statement that everything so far was computationally simple is perhaps an over-simplificcation. For example, type inference, aka type reconstruction, is computationally heavy, at least in the worst case. There are indeed technically advanced type systems around. Nonetheless, it's often a valuable goal not to spend too much time in type checking and furthermore, as far as later optimization is concerned one could give the user the option how much time he is willing to invest and consequently, how agressive the optimization is done.

The word "untractable" on the slides refers to computational complexity; untractable are those for which there is no *efficient* algorithm to solve them. Tractable refers conventionally to *polynomial time* efficiency. Note that it does not say how "bad" the polynomial is, so being tractable in that sense still might not mean practically useful. For non-tractable problems, it's often guaranteed that they don't scale.

10.2 2AC and costs of instructions

2-address machine code used here

- "typical" op-codes, but not a instruction set of a *concrete* machine
- **two address instructions**
- Note: cf. 3-address-code intermediate representation vs. 2-address machine code
  - machine code is **not** lower-level/closer to HW because it has one argument less than 3AC
  - it's just one illustrative choice
  - the new Dragon book: uses **3-address-machine code** (being more modern)
- 2 address machine code: closer to *CISC* architectures,
- *RISC* architectures rather use 3AC.
- translation task from IR to 3AC or 2AC: comparable challenge

2-address instructions format

Format

OP source dest

- note: *order* of arguments here
- restrictions on *source* and *target*
  - register or memory cell
  - source: can additionally be a constant

Also the book Louden [9] uses 2AC. In the 2A machine code there for instance on page 12 or the introductory slides, the order of the arguments is the opposite!

```
ADD a b   // b := a + b
SUB a b   // b := b - a
MUL a b   // b := b + a
GOTO i    // unconditional jump
```

- further opcodes for conditional jumps, procedure calls ....

Side remarks: 3A machine code

Possible format

```
OP source1 source2 dest
```

- but: what's the *difference* to 3A *intermediate* code?
- apart from a more restricted instruction set:
- **restriction** on the **operands**, for example:
  - only *one* of the arguments allowed to be a memory access
  - *no fancy addressing* modes (indirect, indexed ... see later) for memory cells, only for registers
- not "too much" memory-register traffic back and forth per machine instruction
- example:

```
&x = &y + *z
```

may be 3A-intermediate code, but not 3A-machine code

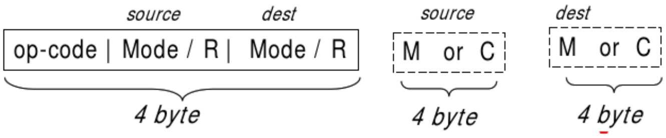
Cost model

- "optimization": need some well-defined "measure" of the "quality" of the produced code
- interested here in *execution* time
- not all instructions take the same time
- estimation of execution
- factors outside our control/not part of the cost model: effect of *caching*

cost factors:

- *size* of instruction
  - it's here not about code size, but
  - instructions need to be *loaded*
  - longer instructions ⇒ perhaps longer load
- address modes (as *additional costs*: see later)
  - registers vs. main memory vs. constants
  - direct vs. indirect, or indexed access

Instruction modes and additional costs



Mode	Form	Address	Added cost	
absolute	M	M	1	
register	R	R	0	
indexed	c(R)	$c + cont(R)$	1	
indirect register	*R	$cont(R)$	0	
indirect indexed	*c(R)	$cont(c + cont(R))$	1	
literal	$\#M$	the *value* M	1	only for source

- indirect: useful for elements in "records" with known off-set
- indexed: useful for slots in arrays

Examples `a := b + c`

Two variants

1. Using registers

```
MOV b, R0   // R0 = b
ADD c, R0   // R0 = c + R0
MOV R0, a   // a = R0

cost = 6
```

2. Memory-memory ops

```

MOV b, a    // a = b
ADD c, a    // a = c + a

cost = 6
```

Use of registers

1. Data already in registers

```
MOV *R1, *R0 // *R0 = *R1
ADD *R2, *R1    // *R1 = *R2 + *R1

cost = 2
```

Assume `R0`, `R1`, and `R2` contain *addresses* for `a`, `b`, and `c`

2. Storing back to memory

```
ADD R2, R1   // R1 = R2 + R1
MOV R1, a    // a = R1

cost = 3
```

Assume R1 and R2 contain *values* for b, and c

10.3 Basic blocks and control-flow graphs

Basic blocks

- machine code level equivalent of straight-line code
- (a largest possible) sequence of instructions without
  - jump out
  - jump in
- elementary unit of code analysis/optimization[1]
- amenable to analysis techniques like
  - static simulation/symbolic evaluation
  - abstract interpretation
- basic unit of code generation

Control-flow graphs

CFG

basically: *graph* with

- nodes = basic blocks
- edges = (potential) jumps (and "fall-throughs")

- here (as often): CFG on 3AIC (linear intermediate code)
- also possible CFG on low-level code,
- or also:
  - CFG extracted from AST[2]
  - here: the opposite: synthesizing a CFG from the linear code
- explicit data structure (as another intermediate representation) or implicit only.

When saying on the slides, a CFG is "basically" a graph, we mean that, apart from some fundamentals which makes them graphs, details may vary. In particular, it may well be the case in a compiler, that cfg's are some accessible intermediate representation, i.e., a specific concrete data structure, with concrete choices for representation. For example, we present here control-flow graphs as *directed* graphs: nodes are connected to other nodes via edges (depicted as arrows), which represent potential successors in terms of the control flow of the program. Concretely, the data structure may additionally (for reasons of efficiency) also represent arrows from successor nodes to predecessor nodes, similar to the way, that linked lists may be implemented in a doubly-linked fashion. Such a representation would be useful when dealing with data flow analyses that work "backwards". As a matter of fact: the one data flow analysis we cover in this lecture (live variable analysis) is of that "backward" kind. Other bells and whistles may be part of the concrete representation, like dedicated start and end nodes. For the purpose of the lecture, when don't go into much concrete details, for us, cfg's are: nodes (corresponding to basic blocks) and edges. This general setting is the most conventional view of cfg's.

[1] Those techniques can also be used across basic blocks, but then they become more costly and challenging.
[2] See also the exam 2016.

From 3AC to CFG: "partitioning algo"

- remember: 3AIC contains *labels* and (conditional) jumps

⇒ algo rather straightforward

- the only complication: some labels can be ignored
- we ignore procedure/method calls here
- concept: "leader" representing the nodes/basic blocks

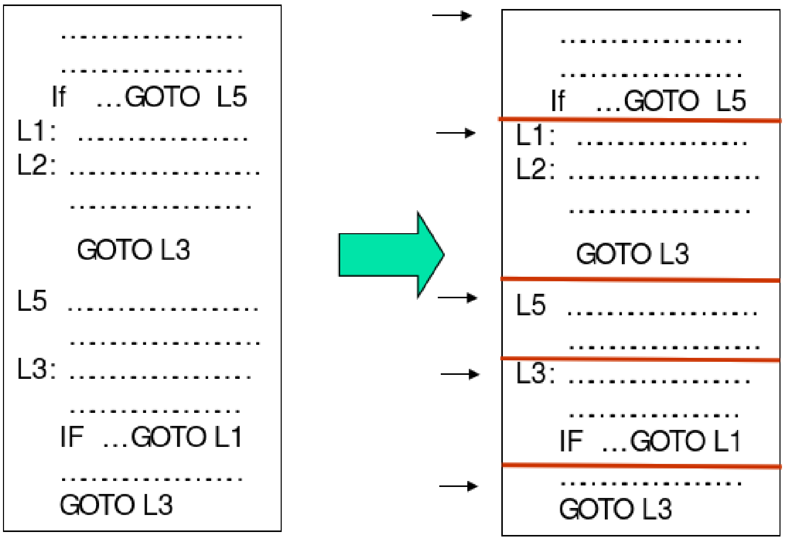
Leader

- first line is a leader
- **GOTO** i: line labelled i is a leader
- instruction *after* a **GOTO** is a leader

Basic block

instruction sequence from (and including) one leader to (but excluding) the next leader or to the end of code

Partitioning algo



- note: no line jumps to L_2

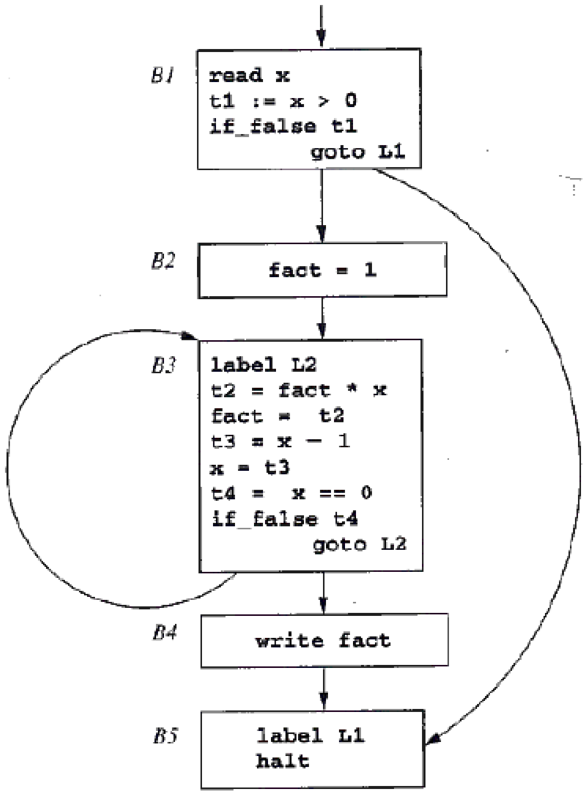
3AIC for faculty (from before)

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
```

```
label L1
halt
```

Faculty: CFG

CFG picture



```
read x
t1 := x > 0
if_false t1
         goto L1
```

```
fact = 1
```

```
label L2
t2 = fact * x
fact =  t2
t3 = x - 1
x = t3
t4 =  x == 0
if_false t4
         goto L2
```

```
write fact
```

```
label L1
halt
```

Remarks

- goto/conditional goto: never *inside* block
- not every block
  - ends in a goto
  - starts with a label
- ignored here: function/method calls, i.e., focus on
- *intra-procedural* cfg

Intra-procedural refers to "inside" one procedure. The opposite is *inter-procedural.* Inter-procedural analyses and the corresponding optimizations are quite harder than *intra*-procedural. In this lecture, we don't cover inter-procedural considerations. Except that call sequences and parameter passing has to do of course with relating different procedures and in that case deal with inter-procedural aspects. But that was in connection with the run-time environments, not what to do about in connection with analysis, register allocation, or optimization. So, in this lecture resp. this chapter, "local" refers to inside one basic block, "global" refers to across many blocks (but inside one procedure). Later, we have a short look at "global" liveness analysis. As mentioned, we dont' cover analyses across procedures, in the terminogy used here, they would be even "more global" than what we call "global".

Levels of analysis

- here: *three* levels where to apply code analysis / optimizations
  1. **local**: per basic block (block-level)
  2. **global**: per function body/intra-procedural CFG
  3. **inter-procedural**: really global, whole-program analysis

- the "more global", the more *costly* the analysis and, especially the optimization (if done at all)

Loops in CFGs

- *loop optimization*: "loops" are thankful places for optimizations
- important for analysis to *detect* loops (in the cfg)
- importance of *loop discovery*: not too important any longer in modern languages.

Loops in a CFG vs. graph cycles

- concept of loops in CFGs **not** identical with **cycles** in a graph
- all **loops** are graph **cycles** but not vice versa

- intuitively: loops are cycles originating from source-level looping constructs ("while")
- goto's may lead to non-loop cycles in the CFG
- importance of loops: loops are "well-behaved" when considering certain optimizations/code transformations (goto's can destroy that…)

Cycles in a graph are well-known. The definition of loops here, while closely related, is *not* identical with that. So, loop-detection is not the same as cycle-detection. Otherwise there'd be no much point discussing it, since cycle detection in graphs is well known, for instance covered in standard algorithms and data structures courses like INF2220/IN2010.

Loops are considered for specific graphs, namely CFGs. They are those kinds of cycles which come from high-level looping constructs (while, for, repeat-until).

Loops in CFGs: definition

- remember: strongly connected components

Loop

A **loop** L in a CFG is a collection of nodes s.t.:

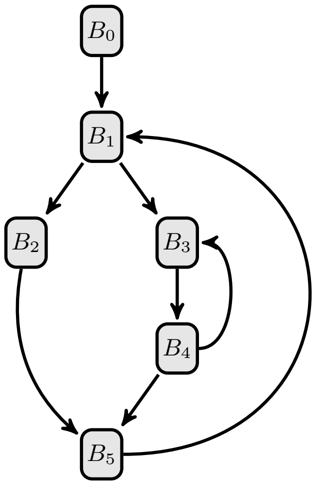
- strongly connected component (with edges completely in L)
- 1 (unique) *entry* node of L, i.e. no node in L has an incoming edge[3] from outside the loop *except* the *entry*

- often additional assumption/condition: "root" node of a CFG (there's only one) is *not* itself an entry of a loop

[3] alternatively: general reachability.

Loop

CFG



- Loops:
  - $\{B_3, B_4\}$
  - $\{B_4, B_3, B_1, B_5, B_2\}$
- Non-loop:
  - $\{B_1, B_2, B_5\}$
- unique entry marked red

The *additional assumption* mentioned on the slide about the special role of the root node of a control flow graph is reminiscent, for example of the condition we assumed for the start-symbol of context-free grammars in the LR(0)-DFA construction: the start symbol must not be mentioned on the right-hand side of any production (and if so, one simply added another start symbol S'). The reasons for the assumption here is similar: assuming that the root node is not itself part of a loop is not a fundamental thing, it just avoids (in some degenerate cases) a special case treatment. The assumption about the form of the control-flow graph is sometime called "isolated entry". A corresponding restriction for the "end" of a control-flow graph is "isolated exit".

Loops as fertile ground for optimizations

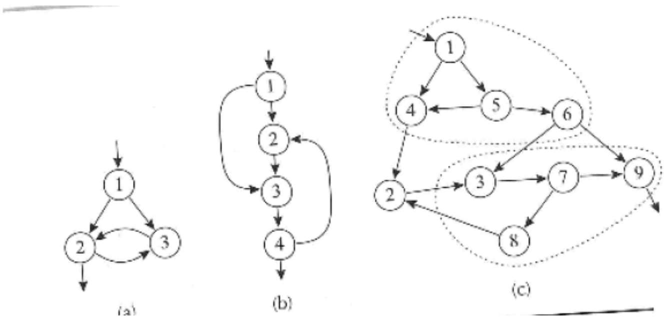
```
while (i < n) {i++; A[i] = 3*k }
```

- possible optimizations
  - move 3*k "out" of the loop
  - put frequently used variables into *registers* while in the loop (like i)
- when moving out computation from the loop:
- put it "right in front of the loop"

⇒ add extra node/basic block in front of the *entry* of the loop[4]

[4] That's one of the motivations for unique entry.

Loop non-examples



Data flow analysis in general

- general *analysis technique* working on CFGs
- **many** concrete forms of analyses
- such analyses: basis for (many) *optimizations*
- *data*: info stored in memory/temporaries/registers etc.
- *control*:
  - movement of the instruction pointer
  - abstractly represented by the CFG
    - inside elementary blocks: increment of the instruction pointer
    - edges of the CFG: (conditional) jumps
    - jumps together with RTE and calling convention

Data flowing from (a) to (b)

Given the control flow (normally as CFG): is it *possible* or is it *guaranteed* ("may" vs. "must" analysis) that some "data" originating at one control-flow point (a) reaches control flow point (b).

Data flow as abstraction

- data flow analysis **DFA**: fundamental and important *static* analysis technique
- it's impossible to decide statically if data from (a) *actually* "flows to" (b)

⇒ approximative (= abstraction)

- therefore: work on the CFG: if there is two options/outgoing edges: *consider both*
- Data-flow answers therefore **approximatively**
  - if it's *possible* that the data flows from (a) to (b)
  - it's *neccessary* or unavoidable that data flows from (a) to (b)
- for *basic blocks*: **exact** answers possible

Treatment of basic blocs

Basic blocks are "maximal" sequences of straight-line code. We encountered a treatment of straight-line code also in the chapter about *intermediate* code generatation. The technique there was called *static simulation* (or simple symbolic execution). *Static simulation* was done for basic blocks only and for the purpose of *translation*. The translation of course needs to be exact, non-approximative. Symbolic evaluation also exist (also for other purposes) in more general forms, especially also working on conditionals.

In summary, the general message is: for SLC and basic blocks, *exact* analyses are possible, it's for the global analysis, when one (necessarily) resorts to overapproximation and abstraction.

Data flow analysis: Liveness

- prototypical / important data flow analysis
- especially important for register allocation

Basic question

When (at which control-flow point) can I be *sure* that I don't need a specific variable (temporary, register) any more?

- optimization: if sure that not needed in the future: register can be used otherwise

Live

A "variable" is **live** at a given control-flow point if there *exists* an execution starting from there (given the level of abstraction), where the variable is *used* in the future.

Static liveness

The notion of liveness given in the slides correspond to static liveness (the notion that static liveness analysis deals with). That is hidden in the condition "given the level of abstraction" for example, using the given control-flow graph. A variable in a given concrete execution of a program is *dynamically live* if in the future, it is still needed (or, for non-deterministic programs: if there exists a future, where it's still used. Dynamic liveness is undecidable, obviously.

Definitions and uses of variables

- talking about "variables": also temporary variables are meant.
- basic notions underlying most data-flow analyses (including liveness analysis)
- here: def's and uses of *variables* (or temporaries etc.)
- all data, including intermediate results) has to be stored somewhere, in variables, temporaries, etc.

Def's and uses

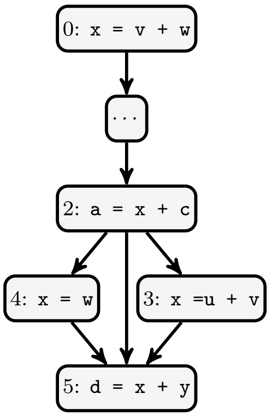
- a **"definition"** of x = assignment to x (store to x)
- a **"use"** of x: read content of x (load x)

- variables can occur more than once, so

- a definition/use refers to *instances* or *occurrences* of variables ("use of x in line l " or "use of x in block b ")
- same for liveness: "x is live here, but not there"

Defs, uses, and liveness

CFG



- *x* is "defined" (= assigned to) in 0, 3, and 4
- *u* is **live** "in" (= at the end of) block 2, as it *may* be *used* in 3
- a *non-live* variable at some point: "dead", which means: the corresponding memory can be reclaimed
- *note*: here, liveness across block-boundaries = "global" (but blocks contain only one instruction here)

Def-use or use-def analysis

- use-def: given a "use": determine all possible "definitions"
- def-use: given a "def": determine all possible "uses"
- for straight-line-code/inside one basic block
  - deterministic: each line has has exactly one place where a given variable has been assigned to last (or else not assigned to in the block). Equivalently for uses.
- for whole CFG:
  - approximative ("may be used in the future")
  - more advanced techiques (caused by presence of loops/cycles)
- def-use analysis:
  - closely connected to liveness analysis (basically the same)
  - *prototypical* data-flow question (same for use-def analysis), related to many data-flow analyses (but not all)

Side-remark: SSA

Side remark: *Static single-assignment* (SSA) format:

- at most one assignment per variable.

- "definition" (place of assignment) for each variable thus *clear* from its name

Calculation of def/uses (or liveness . . .)

- three levels of complication
  1. inside basic block
  2. branching (but no loops)
  3. Loops
  4. [even more complex: inter-procedural analysis]

For SLC/inside basic block

- deterministic result
- simple “one-pass” treatment enough
- similar to “static simulation”
- [Remember also AG’s]

For whole CFG

- iterative algo needed
- dealing with non-determinism: over-approximation
- “closure” algorithms, similar to the way e.g., dealing with *first* and *follow* sets
- = fix-point algorithms

Inside one block: optimizing use of temporaries

- simple setting: *intra*-block analysis & optimization, only
- temporaries:
  - symbolic representations to hold intermediate results
  - generated on request, assuming unbounded numbers
  - intention: use **registers**
- limited about of register available (platform dependent)

Assumption about temps (here)

- temp’s *don’t transfer* data across blocks (≠ program var’s)

⇒ temp’s *dead* at the beginning and at the end of a block

- but: variables have to be *assumed* live at the end of a block (block-local analysis, only)

At this point, one can check one’s undestanding: why is it that the variables are *assumed* live (as opposed to assumed *dead*, or perhaps assumed a status “I-don’t-know”)?

Intra-block liveness

Code

```
t1 := a - b
t2 := t1 * a
a  := t1 * t2
t1 := t1 - c
a  := t1 * a
```

- neither temp’s nor vars in the example are “single assignment”,
- but first occurrence of a temp in a block: a definition (but for temps it would often be the case, anyhow)
- let’s call *operand:* variables or temp’s
- *next use* of an operand:
- uses of operands: on the rhs’s, definitions on the lhs’s
- not good enough to say “t_1 is live in line 4” (why?)

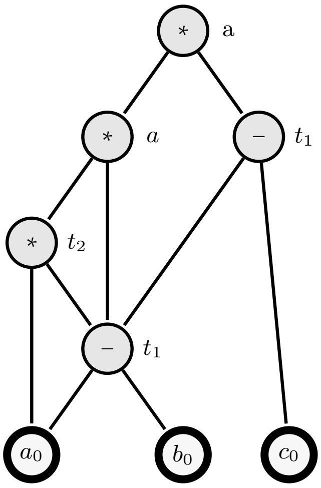
Note: the 3AIC may allow also literal constants as operator arguments; they don't play a role right now.

In the following, the "next-uses" of operands and variables are arranged in a graph-like manner. As we are treating straight-line code, there are no cycles in that graph. In other words it's an *acyclic* graph. That form of graph is also known as DAG: *directed acyclic graph*. NB: the graph on the next slides don't use "arrows" (as would be common in directed graphs. Being acyclic, the is only one direction here, that's from bottom to top. The incoming edges indicate the dependencies of an intermediate result on it's operands. Since we are dealing with 3AC, there are two operands (or less), which means, nodes have typically 2 incoming edges (from below). The nodes are labelled by the operator as well as the target memory location (variable or temporary).

The DAG, reading it from bottom to top, represents the "next-use" for each variable/temporary. As mentioned, each node has at most 2 incoming edges (an in-degree of 2). Since a variable may have more than 2 next uses, the out-degree may well arbitrarily large. In the example, t_1 is used for instance, 3 times at some point in the code.

DAG of the block

DAG



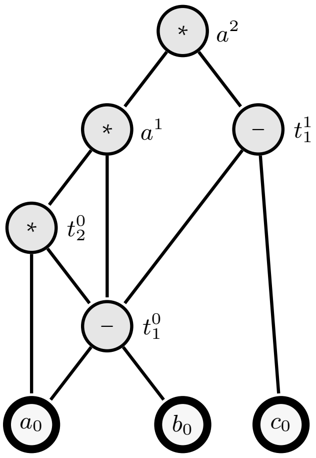
Text

- no linear order (as in code), only *partial order*
- *the* next use: meaningless
- but: *all "next" uses* visible (if any) as "edges upwards"
- node = occurrences of a variable
- e.g.: the "lower node" for "defining"*assigning* to t_1 has *three* uses
- different "versions" (instances) of t_1

DAG / SA

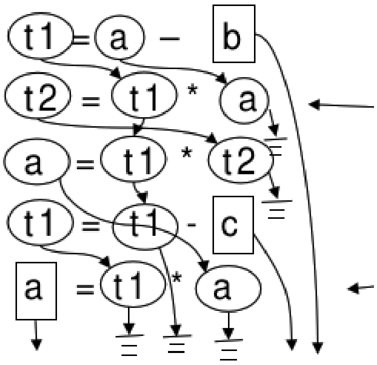
SA = "single assignment"

- indexing different "versions" of right-hand sides
- often: temporaries generated as single-assignment already
- cf. also *constraints* + remember AGs



Intra-block liveness: idea of algo

Picture



- liveness-status of an operand: *different* from lhs vs. rhs in a given instruction
- informal definition: an operand is live at some occurrence, if it's used some place in the future

consider statement $x_1 := x_2\ op\ x_3$

- A variable x is live at the *beginning* of $x_1 := x_2\ op\ x_3$, if
  1. if x is x_2 or x_3, or
  2. if x live at its *end*, if x and x_1 are different variables
- A variable x is live at the *end* of an instruction,
  - if it's live at *beginning of the next* instruction
  - if no next instruction
    - temp's are dead
    - user-level variables are (assumed) live

Note: the graph on the top left-hand side of the slide is *not* the same as the DAG shown earlier. Maybe the graphical represention here is not too usefule. It indicates the next uses of a variable, if any. It also indicates if a variable is not used in the future (but the special "ground symbol"). However, the start-point of the edges are not all really helpful in getting an overview. In the first line: the arrow from t_1 to t_1 in the second line rougly corresponds to the edge in the DAG (as it goes from a definition (of t_1) its next use. However, the edge from a in the first line to a in the second line is less motivated: it would correspond to an edge from a "use" to a "next use", but normally one is not interested in that too much. Therefore, one should not "overinterpret" the graph in the figure too much.

A better representation would be, for each line, pointers from all variables to next uses, not just from variables that happen to be mentioned in a line.

Liveness

Previous "inductive" definition

expresses liveness status of variables *before* a statement dependent on the liveness status of variables *after* a statement (and the variables used in the statement)

- *core* of a straightforward iterative algo
- simple **backward** scan
- the algo we sketch:
  - not just boolean info (live = yes/no), instead:
  - operand live?
    - yes, and with next use inside is block (and indicate instruction where)
    - yes, but with no use inside this block
    - not live
  - even more info: not just that but indicate, where's the **next use**

Backward scan and SLC

Remember in connection with the given algo for intra-block analysis, i.e. analysis for straight-line code. In the presence of loops/analysing a complete CFG, a simple 1-pass does not suffice. More advanced techniques ("multiple-scans") are needed then, which may amount to fixpoint calculations. Doing fixpoint calculations increases the complexity of the problem (And the needed theoretical background). As a further side remark: earlier in this chapter we elaborated on the fine line that separates cycles in a graph from the notion of loops, where loops are a particular well-structured from of cycles. Without going into details: if one is dealing with cfg's which are guaranteed to contain only loops (but not proper more general cycles), one can apply special techniques or strategies to deal with the cycles. In particular, one can attack the loops "inside out". That strategy is possible, as loops (as opposed to cycles) appear "nested". Attacking the loops in that manner is more efficient than iterating though the graph without taking the nesting structure as compass.

Algo: dead or alive (binary info only)

```
// ----- initialise T   ----------------------------
  for all entries: T[i,x] := D
  except: for all variables a // but not temps
             T[n,a] := L,
//------- backward pass ----------------------------
for instruction i =  n-1 down to 0
   let current instruction at i+1: $x := y \ op\  z$;
      T[i,x] := D // note order; x can ``equal'' y or z
      T[i,y] := L
      T[i,z] := L
end
```

- Data structure T: table, mapping for each line/instruction i and variable: boolean status of "live"/"dead"
- represents liveness status per variable *at the end (i.e. rhs)* of that line
- basic block: n instructions, from 1 until n, where "line 0" represents the "sentry" imaginary line "before" the first line (no instruction in line 0)
- *backward scan* through instructions/lines from n to 0

Algo′: dead or else: alive with next use

- More refined information
- not just binary “dead-or-alive” but **next-use** info

⇒ three kinds of information

1. Dead: D
2. Live:
   - with *local* line number of *next use*: $L(n)$
   - *potential* use of outside local basic block $L(\perp)$

- otherwise: basically the same algo

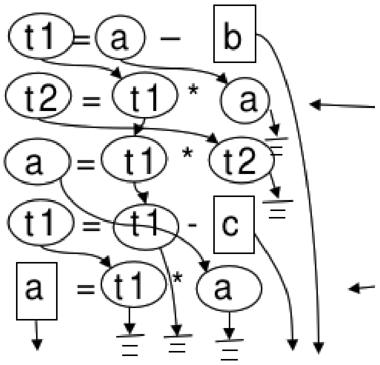
```
// -----  initialise T   ----------------------------
  for all entries: T[i,x] := $\livenextdeadnonlocal$
  except: for all variables a // but not temps
              T[n,a] := $\livenextnonlocal$ ,
//------- backward pass ----------------------------
for instruction i = n-1 down to 0
    let current instruction at i+1: $x := y \ op\  z$;
       T[i,x] := $\livenextdeadlocal$ // note order; x can ``equal'' y or z
       T[i,y] := $\livenextlocal{i+1}$
       T[i,z] := $\livenextlocal{i+1}$
end
```

Run of the algo′

Run/result of the algo

line	a	b	c	t_1	t_2
[0]	$L(1)$	$L(1)$	$L(4)$	$L(2)$	D
1	$L(2)$	$L(\perp)$	$L(4)$	$L(2)$	D
2	D	$L(\perp)$	$L(4)$	$L(3)$	$L(3)$
3	$L(5)$	$L(\perp)$	$L(4)$	$L(4)$	D
4	$L(5)$	$L(\perp)$	$L(\perp)$	$L(5)$	D
5	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	D

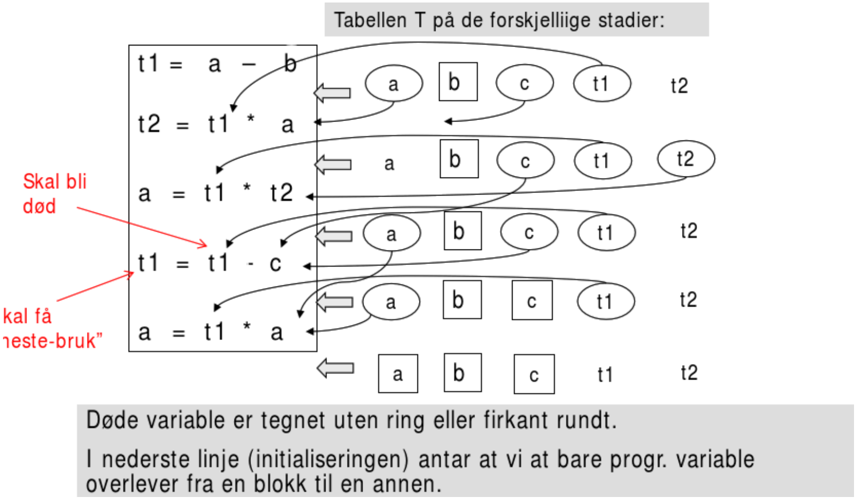
Picture



```
t1  := a - b
t2  := t1 * a
a   := t1 * t2
t1  := t1 - c
a   := t1 * a
```

In the table, the entries marked read indicate where "changes" occur; remember that the table is filled from bottom to top, we are doing a backward scan.

Liveness algo remarks



Tabellen T på de forskjelliige stadier:

Døde variable er tegnet uten ring eller firkant rundt.

I nederste linje (initialiseringen) antar at vi at bare progr. variable overlever fra en blokk til en annen.

- here: *T* data structure traces (L/D) status per variable × "line"
- in the *remarks* in the notat:
  - alternatively: store liveness-status *per variable* only
  - works as well for one-pass analyses (but only without loops)
- this version here: corresponds better to *global* analysis: 1 line can be seen as one small basic block

10.4 Code generation algo

Simple code generation algo

- simple algo: *intra-block* code generation
- core problem: **register use**
- register allocation & assignment[5]
- hold calculated values in registers longest possible
- intra-block only ⇒ at exit:
  - all *variables* stored back to main memory
  - all temps assumed "lost"
- remember: assumptions in the intra-block liveness analysis

Limitations of the code generation

- local **intra block**:
  - no analysis across blocks
  - no procedure calls, etc.
- no complex data structures
  - arrays
  - pointers
  - ...

[5] Some distinguish register *allocation*: "should the data be held in register (and how long)" vs. register *assignment*: "which of the available registers to use for that"

some limitations on how the algo itself works for one block

- for read-only variables: never put in registers, even if variable is *repeatedly* read
  - algo works only with the temps/variables given and does not come up with new ones
  - for instance: DAGs could help
- no *semantics* considered
  - like *commutativity*: $a + b$ equals $b + a$

The limitation that read-only variables are not put into registers is not a “design-goal”, it’s a not so smart side-effect on the way the algo works. The algo is a quite straightforward way of making use of registers which works block-local. Due to its simplicity, the treatment of read-only variables leaves room for improvement. The code generation makes use of liveness information, if available. In case one has invested in some global liveness analysis (as opposed to a local one), the code generation could profit from that by getting more efficient. But its *correctness* does not rely on that. Even without liveness information, it is correct, by assuming conservatively or defensively, that all variables are always live (which is the worst-case assumption).

Purpose and “signature” of the *getreg* function

- one *core* of the code generation algo
- simple code-generation here ⇒ simple *getreg*

getreg function

available: *liveness/next-use* info

Input: TAIC-instruction $x := y$ **op** z

Output: return *location* where x is to be stored

- **location**: register (if possible) or memory location

Coge generation invariant

it should go without saying . . . :

Basic safety invariant

At each point, “live” variables (with or without next use in the current block) must exist in at least one location

- another invariant: the location returned by getreg: the one where the rhs of a 3AIC assignment ends up

Register and address descriptors

- code generation/*getreg*: keep track of
  1. register contents
  2. addresses for names

Register descriptor

- tracking current “content” of reg’s (if any)
- consulted when new reg needed
- as said: at block entry, assume all regs unused

Address descriptor

- tracking location(s) where current value of name can be found
- possible locations: register, stack location, main memory
- > 1 location possible (but not due to overapproximation, exact tracking)

By saying that the register descriptor is needed to track the content of a register, it’s not meant the actual *value* (which will only be known at run-time). It’s rather keeping track of the following information: the content of the register correspond to the (current content of the following) variable(s). Note: there might be situations where a register corresponds to more than one variable.

Code generation algo for $x := y$ op z

1. determine location (preferably register) for result

```
l = getreg( ``x := y op z'')
```

2. make sure, that the value of y is in l :
   - consult address descriptor for y ⇒ current locations l_y for y
   - choose the best location l_y from those (preferably register)
   - if value of y *not* in l, generate

   ```
   MOV $l_y$, l
   ```

3. generate

```
OP $l_z$, l  // $l_z$: a current location of z (prefer reg's)
```

   - update address descriptor $[x \mapsto_{\cup} l]$
   - if l is a reg: update reg descriptor $l \mapsto x$
4. exploit liveness/next use info: update register descriptors

Skeleton code generation algo for $x := y$ op z

```
$l$ = getreg(``x:= y op z'') // target location for x
if $l \notin \locsof{y}{\tablead}$ then let $l_y \in \locsof{y}{\tablead}$) in  emit ("MOV $l_y,\ l$");
let $l_z \in \locsof{z}{\tablead}$      in emit ("OP $l_z, l$");
```

- “skeleton”
  - *non-deterministic*: we ignored how to choose l_z and l_y
  - we ignore *book-keeping* in the *name* and *address* descriptor tables (⇒ step 4 also missing)
  - details of *getreg* hidden.

Non-deterministic code generation algo for $x := y$ op z

```
l = getreg(``x:= y op z'') // generate target location for x
if $l \notin \locsof{y}{\tablead}$
then let $l_y \in \locsof{y}{\tablead}$) // pick a location for  y
      in  emit (MOV $l_y$, l)
else skip;
let $l_z \in \locsof{z}{\tablead}$)  in emit (``OP $l_z$, l'');
$\tablead := \tablead\setaddto{x}{l}$;
if   l is a register
then $\tablerd := \tablerd\setto{l}{x}$
```

Exploit liveness/next use info: recycling registers

- register descriptors: don't update themselves during code generation
- once set (e.g. as $R_0 \mapsto t$), the info stays, unless reset
- thus in step 4 for $z := x$ **op** y:

Code generation algo for $x := y$ op z

```
$l$ = getreg("i: x := y op z")   // $i$ for instructions line number/label
if $l \notin \locsof{y}{\tablead}$
then let $l_y$ =  best ($\locsof{y}{\tablead}$)
      in  emit ("$\red{\mathbf{MOV}\ l_y,\ l}$")
else skip;
let $l_z$ = best ($\locsof{z}{\tablead}$)
in emit ("$\red{\mathbf{OP}\ l_z,l}$");
$\tablead := \tablead\setwithoutto{\_}{l}$;
$\tablead := \tablead\setto{x}{l}$;
$\tablerd := \tablerd\setto{l}{x}$;

if  $\lnot \tableliveat{i}{y}$ and $\tablead(y) = r$ then $\tablerd := \tablerd\setwithoutto{r}{y}$
if  $\lnot \tableliveat{i}{z}$ and $\tablead(z) = r$ then $\tablerd := \tablerd\setwithoutto{r}{z}$
```

To exploit liveness info by recycling reg's

if y and/or z are currently

- *not live* and are
- in *registers*,

⇒ "wipe" the info from the corresponding register descriptors

- side remark: for address descriptor
  - no such "wipe" needed, because it won't make a difference (y and/or z are not-live anyhow)
  - their address descriptor wont' be consulted further in the block

getreg algo: $x := y$ op z

- goal: return a location for x
- basically: check possibilities of register uses,
- starting with the "cheapest" option

Do the following steps, in that order

1. **in place:** if x is in a register already (and if that's fine otherwise), then return the register

2. **new register:** if there's an unsused register: return that

3. **purge filled register:** choose more or less cleverly a filled register and save its content, if needed, and return that register

4. **use main memory:** if all else fails

getreg algo: $x := y$ op z in more details

1. if
   - y in register R
   - R holds *no alternative names*
   - y is *not live* and has no next use after the 3AIC instruction
   - $\Rightarrow$ return R
2. else: if there is an **empty** register R': return R'
3. else: if
   - x has a next use [or operator requires a register] $\Rightarrow$
     - find an occupied register R
     - store R into M if needed (MOV R, M))
     - don't forget to update M 's address descriptor, if needed
     - return R
4. else: x not used in the block *or* no suituable occupied register can be found
   - return x as location L

- choice of purged register: *heuristics*
- remember (for step 3): registers may contain value for > 1 variable ⇒ *multiple* MOV's

Sample TAIC

```
d := (a-b) + (a-c) + (a-c)
```

```
t := a − b
u := a − c
v := t + u
d := v + u
```

line	a	b	c	d	t	u	v
[0]	$L(1)$	$L(1)$	$L(2)$	D	D	D	D
1	$L(2)$	$L(\perp)$	$L(2)$	D	$L(3)$	D	D
2	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	$L(3)$	$L(3)$	D
3	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	D	$L(4)$	$L(4)$
4	$L(\perp)$	$L(\perp)$	$L(\perp)$	$L(\perp)$	D	D	D

Code sequence

Code sequence

	3AIC	2AC	reg. descr.		addr. descriptor						
			R_0	R_1	a	b	c	d	t	u	v
[0]			⊥	⊥	a	b	c	d	t	u	v
1	t := a − b	**MOV** a, R0	$[a]$		$[R_0]$						
		SUB b, R0	t		$\cancel{R_0}$				R_0		
2	u := a − c	**MOV** a, R1	·	$[a]$	$[R_0]$						
		SUB c, R1		u	$\cancel{R_0}$					R_1	
3	v := t + u	**ADD** R1, R0	v	·					$\cancel{R_0}$		R_0
4	d := v + u	**ADD** R1, R0	d					R_0			$\cancel{R_0}$
		MOV R0, d									
			R_i: unused		all var's in "home position"						

- address descr's: "home position" not explictely needed.
- e.g. variable a always to be found "at a ", as indicated in line "0".
- in the table: only *changes* (from top to bottom) indicated
- after line 3:
  - t **dead**
  - t resides in R_0 (and nothing else in R_0)

  → **reuse** R_0
- Remark: info in [brackets]: "ephemeral"

10.5 Ignore for now

10.6 Global analysis

10.7 From "local" to "global" data flow analysis

- data stored in variables, and "flows from definitions to uses"
- **liveness** analysis
  - one *prototypical* (and important) data flow analysis
  - so far: *intra-block* = straight-line code
- related to
  - *def-use* analysis: given a "definition" of a variable at some place, where it is (potentially) used
  - *use-def*: (the inverse question, "reaching definitions"
- other similar questions:
  - has a value of an expression been calculated before ("available expressions")
  - will an expression be used in all possible branches ("very busy expressions")

10.8 Global data flow analysis

- block-local
  - block-local analysis (here liveness): *exact* information possible
  - block-local liveness: *1 backward scan*
  - important use of liveness: *register allocation*, temporaries typically don't survive blocks anyway
- **global**: working on complete CFG

2 complications

- **branching**: *non-determinism*, unclear which branch is taken
- **loops** in the program (loops/cycles in the graph): simple *one pass* through the graph does not cut it any longer

- *exact* answers no longer possible (undecidable)

⇒ work with safe **approximations**

- this is: general characteristic of DFA

10.9 Generalizing block-local liveness analysis

- *assumptions* for block-local analysis
  - all program variables (assumed) *live* at the end of each basic block
  - all temps are assumed *dead* there.
- now: we do better, info across blocks

at the end of each block:

which variables **may** be used in subsequent block(s).

- **now:** re-use of temporaries (and thus corresponding registers) across blocks possible
- remember local liveness algo: determined liveness status per var/temp *at the end* of each "line/instruction"

We said that "now" a re-use of temporaries is possible. That is in contrast to the block *local* analysis we did earlier, before the code generation. Since we had a local analysis only, we had to work with assumptions converning the variables and temporaries at the end of each block, and the assumptions were "worst-case", to be on the safe side. Assuming variables live, even if actually they are not, is safe, the opposite may be unsafe. For temporaries, we assumed "deadness". So the code generator therefore, under this assumption, must not reuse temporaries across blocks.

One might also make a parallel to the "local" liveness algorithm from before. The problem to be solved for liveness is to determined the status for each variable *at the end of each block*. In the local case, the question was analogous, but for the "end of each line". For sake of making a parallel one could consider each line as individual block. Actually, the global analysis would give *identical* results also there. The fact that one "lumps together" maximal sequences of straight-line code into the so-called *basic blocks* and thereby distinguishing between local and global levels is a matter of efficiency, not a principle, theoretical distinction. Remember that basic blocks can be treated in one single path, whereas the whole control-flow graph cannot: do to the possibility of loops or cycles there, one will have to treat "members" of such a loop potentially more than one (later we will see the corresponding algorithm). So, before addressing the global level with its loops, its a good idea to "pre-calculate" the data-flow situation per block, where such treatment requies one pass for each individual block to get an *exact* solution. That avoid potential line-by-line *recomputation* in case a basic block neeeds to be treated multiple times.

10.10 Connecting blocks in the CFG: *inLive* and *outLive*

- CFG:
  - pretty conventional graph (nodes and edges, often designated start and end node)
  - *nodes* = basic blocks = contain straight-line code (here 3AIC)
  - being conventional graphs:
    - conventional representations possible
    - E.g. nodes with lists/sets/collections of immediate *successor nodes* plus immediate *predecessor nodes*
- remember: local liveness status

  - can be different *before* and *after* one single instruction
  - liveness status *before* expressed as dependent on status *after*

  ⇒ **backward** scan
- Now per block: *inLive* and *outLive*

Loops vs. cycles

As a side remark. Earlier we remarked that loops are closely related to cycles in a graph, but not 100% the same. Some forms of analyses resp. algos assume that the only cycles in the graph are *loops*. However, the techniques presented here work generally, i.e., the worklist algorithm in the form presented here works just fine also in the presence of general cycles. If one had no cycles, no loops. special strategies or variations of the worklist algo could exploit that to achieve better efficiency. We don't pursue that issue here. In that connection it might also be mentioned: if one had a program *without* loops, the best strategy would be *backwards*. If one had straight-line code (no loops and no branching), the algo corresponds directly to "local" liveness, explained earlier.

10.11 *inLive* and *outLive*

- tracing / approximating set of live variables[6] at the *beginning* and *end* per basic block
- *inLive* of a block: depends on
  - *outLive* of that block and
  - the SLC inside that block
- *outLive* of a block: depends on *inLive* of the *successor* blocks

Approximation: To err on the safe side

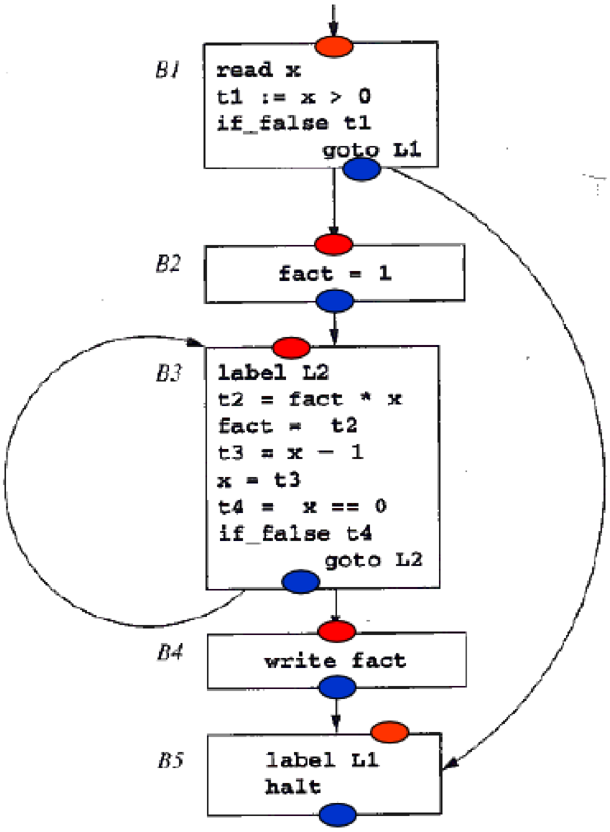
Judging a variable (statically) live: always *safe*. Judging wrongly a variable *dead* (which actually will be used): **unsafe**

- goal: **smallest** (but **safe**) possible sets for *outLive* (and *inLive*)

[6] To stress "approximation": *inLive* and *outLive* contain sets of *statically* live variables. If those are dynamically live or not is undecidable.

10.12 Example: Faculty CFG

CFG picture



Explanation

- *inLive* and *outLive*
- picture shows arrows as *successor nodes*
- needed *predecessor nodes* (reverse arrows)

node/block	predecessors
B_1	$\varnothing$
B_2	$\{B_1\}$
B_3	$\{B_2, B_3\}$
B_4	$\{B_3\}$
B_5	$\{B_1, B_4\}$

10.13 Block local info for global liveness/data flow analysis

- 1 CFG per procedure/function/method
- as for SLC: algo works **backwards**
- for each block: underlying block-local liveness analysis

3-valued block local status per variable

result of block-local live variable analysis

1. *locally live* on entry: variable used (before overwritten or not)
2. *locally dead* on entry: variable overwritten (before used or not)
3. status not locally determined: variable neither assigned to nor read locally

- for efficiency: *precompute* this info, before starting the global iteration ⇒ avoid *recomputation* for blocks in loops

Precomputation

We mentioned that, for efficiency, it's good to *precompute* the local data flow per local block. In the smallish examples we look at in the lecture or exercises etc.: we don't pre-compute, we often do it simply on-the-fly by "looking at" the blocks' of SLC.

10.14 Global DFA as iterative "completion algorithm"

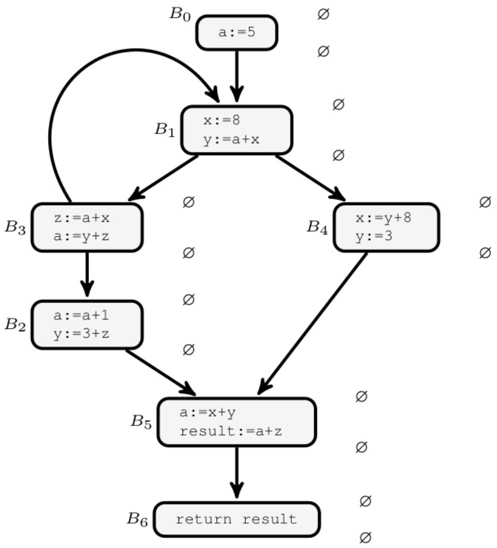
- different names for the general approach
  - *closure* algorithm, *saturation* algo
  - *fixpoint* iteration
- basically: a big loop with
  - **iterating** a step approaching an intended solution by making current approximation of the solution *larger*
  - **until** the solution stabilizes
- similar (for example): calculation of first- and follow-sets
- often: realized as *worklist algo*
  - named after central data-structure containing the "work-still-to-be-done"
  - here possible: worklist containing nodes untreated wrt. liveness analysis (or DFA in general)

10.15 Example

```
    a := 5
L1: x := 8
    y := a + x
    if_true x=0 goto L4
    z := a + x          // B3
    a := y + z
    if_false a=0  goto L1
    a := a + 1          // B2
    y := 3 + x
L5  a := x + y
    result := a + z
    return result       // B6
L4: a := y + 8
    y := 3
    goto L5
```

10.16 CFG: initialization

Picture



- *inLive* and *outLive*: *initialized* to ∅ everywere
- note: start with (most) *unsafe* estimation
- extra (return) node
- but: analysis here *local per procedure*, only

10.17 Iterative algo

General schema

Initialization start with the "minimal" estimation (∅ everywhere)

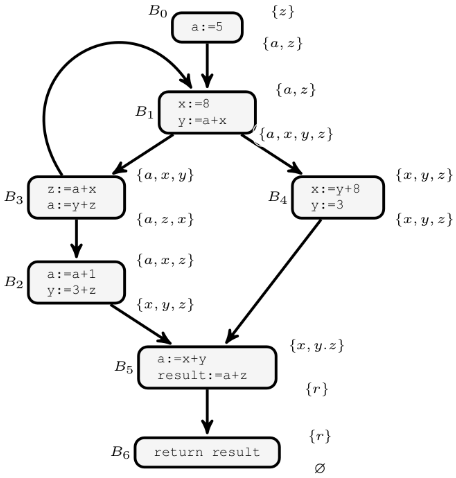
Loop pick one node & update (= enlarge) liveness estimation in connection with that node

Until finish upon stabilization (= no further enlargement)

- order of treatment of nodes: in princple arbitrary[7]
- in tendency: following edges **backwards**
- comparison: for linear graphs (like inside a block):
  - no repeat-until-stabilize loop needed
  - 1 simple backward scan enough

[7] There may be more efficient and less efficient orders of treatment.

10.18 Liveness: run



10.19 Liveness example: remarks

- the shown traversal strategy is (cleverly) backwards
- example resp. example run simplistic:
- the *loop* (and the choice of "evaluation" order):

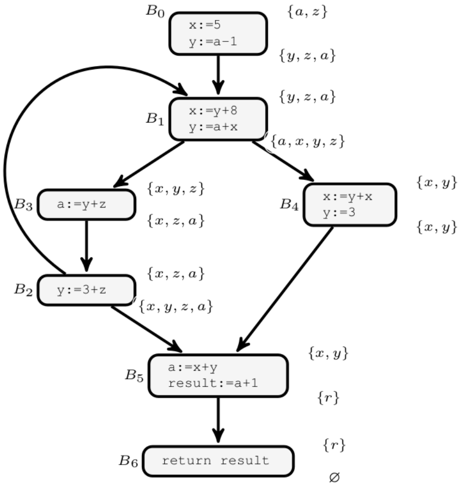
"harmless loop"

after having updated the *outLive* info for B_1 following the edge from B_3 to B_1 *backwards* (propagating flow from B_1 back to B_3) **does not increase the current solution for** B_3

- no need (in this particular order) for continuing the iterative search for stabilization
- in other examples: loop iteration cannot be avoided
- note also: end result (after stabilization) **independent from evaluation order**! (only some strategies may stabilize faster...)

In the script, the figure shows the end-result of the global liveness analysis. In the slides, there is a "slide-show" which shows step-by-step how the liveness-information propagates (= "flows") through the graph. These step-by-step overlays, also for other examples, are not reproduced in the script.

10.20 Another, more interesting, example



10.21 Example remarks

- loop: this time leads to updating estimation more than once
- evaluation order not chosen ideally

10.22 Precomputing the block-local "liveness effects"

- *precomputation* of the relevant info: efficiency
- traditionally: represented as *kill* and *generate* information
- here (for liveness)
  1. **kill**: variable instances, which are overwritten
  2. **generate**: variables used in the block (before overwritten)
  3. rests: all other variables won't change their status

Constraint per basic block (transfer function)

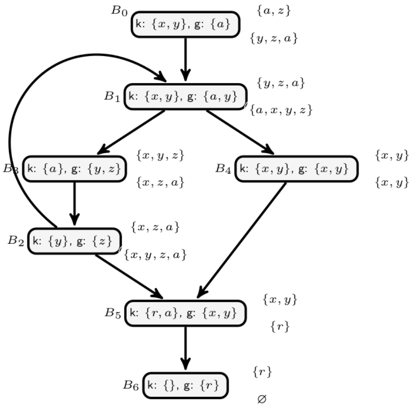
$$\mathit{inLive} = \mathit{outLive} \backslash \mathit{kill}(B) \cup \mathit{generate}(B)$$

- note:
  - order of kill and generate in above's equation
  - a variable killed in a block may be "revived" in a block
- simplest (one line) example: `x := x +1`

Order of kill and generate

As just remarked, one should keep in mind the oder of kill and generate in the definition of transfer functions. In principle, one could also arrange the opposite order (interpreting kill and generatate slightly differently). One can also define the so-called transfer function directly, without splitting into kill and generate (but for many (but not all) such a separation in kill and generate functionality is possible and convenient to do). Indeed using transfer functions (and kill and generate) works for many other data flow analyses as well, not just liveness analysis. Therefore, understanding liveness analysis basically amounts to having understood data flow analysis.

10.23 Example once again: kill and gen



Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques and Tools*. Pearson,Addison-Wesley, second edition.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

[3] Appel, A. W. (1998a). *Modern Compiler Implementation in Java*. Cambridge University Press.

[4] Appel, A. W. (1998b). *Modern Compiler Implementation in ML/Java/C*. Cambridge University Press.

[5] Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(113–124).

[6] Cooper, K. D. and Torczon, L. (2004). *Engineering a Compiler*. Elsevier.

[7] Hopcroft, J. E. (1971). An $n \log n$ algorithm for minimizing the states in a finite automaton. In Kohavi, Z., editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, New York.

[8] Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–42. Princeton University Press.

[9] Louden, K. (1997). *Compiler Construction, Principles and Practice*. PWS Publishing.

[10] Rabin, M. and Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research Developments*, 3:114–125.

[11] Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419.

Index