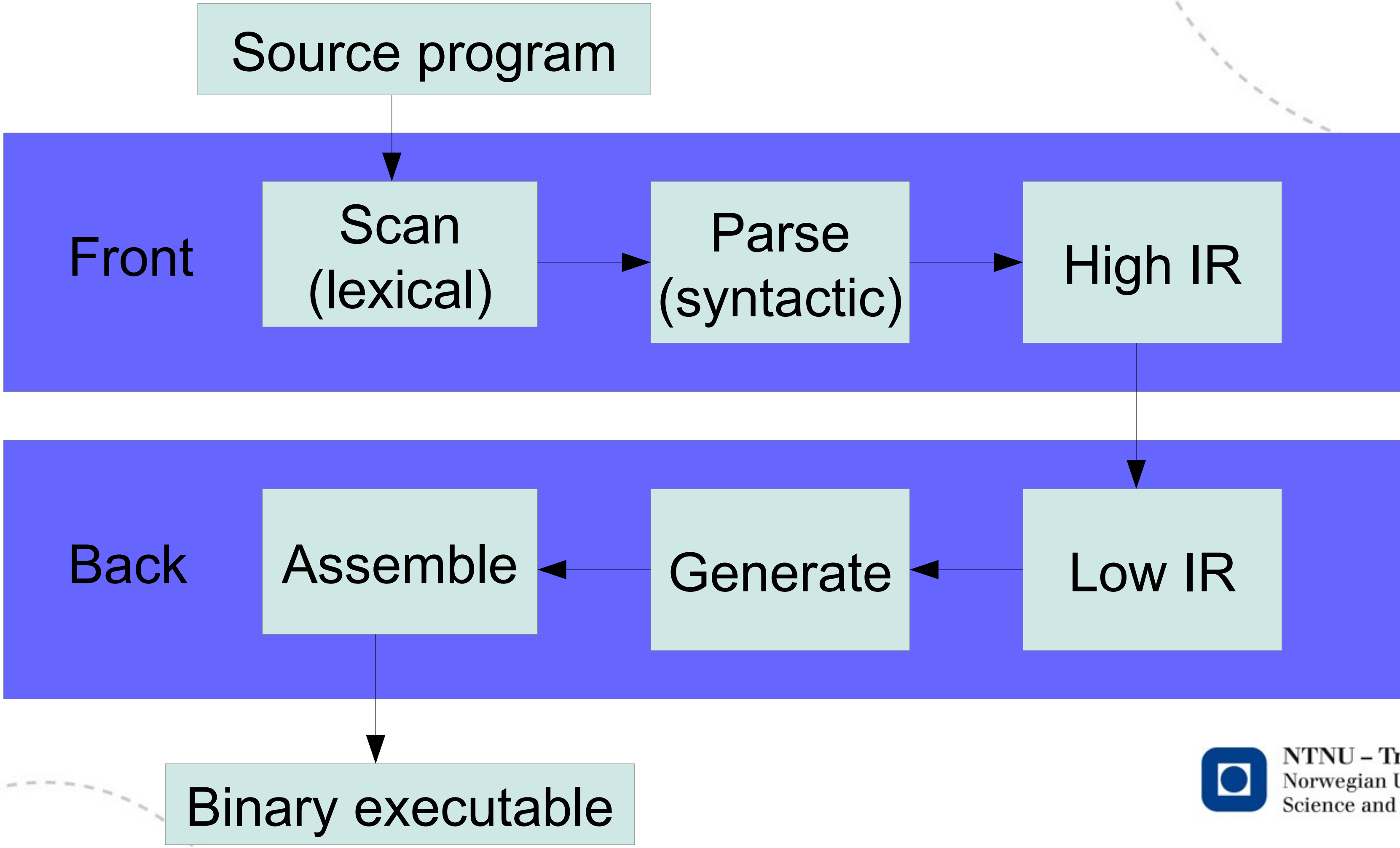
NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4205 Grand Summary, pt. 1

An overall view (of little detail)



Lexical analysis

- Lexical analysis covers splitting of text into
  - Tokens (symbolic values for what kind of word we see)
  - Lexemes (the text which is the actual recognized word)
- That is, things like
  - Language keywords (fixed strings of predefined words)
  - Operators (typically, short strings of funny characters)
  - Names (alphanumeric strings)
  - Values (integers, floating point numbers, string literals...)
- Why does it happen?
  - Technically, this could all be defined syntactically
  - This would inflate the grammar for no good reason
  - Choosing an appropriate dictionary and separating it in a scanner makes design easier

NTNU – Trondheim
Norwegian University of
Science and Technology

Lexical analysis

- What happens?
  - Characters are grouped into indivisible lumps, in pairs of *token* values and *lexemes*
  - The token value is just an arbitrary number, which can be used for a placeholder in a grammar, but says nothing about the text which produced it.
  - The lexeme is the text matching the token, it says nothing about the grammatical role of the word, but everything about which particular instance from a class of words we are dealing with
- How does it happen?
  - Deterministic finite state automata are simulated with the source program as input, changing state on each read character
  - There is a 1-1 correspondence between DFA and *regular expressions*

DFA & regular expressions

- Regular expressions are defined in terms of
  - Literal characters, and groups of them
  - Closures (zero-or-more *, “Kleene closure”), (one-or-more, +)
  - Selection (either-or, |)
- Character classes denote the transitions between states (arcs in a directed graph representation of DFA)
- Kleene closure is an edge from a state to itself
  - One-or-more follows by prepending one state
- Selection is nodes where two branches in the graph diverge from one another

NFA and DFA

- When multiple edges leave an FA state on the same symbol (or equivalently, an FA state may have transitions taken without input), it is a lot easier to construct an automaton for a given class of words
- This breaks the simple DFA simulation algorithm, as the automaton is now NFA (Nondeterministic FA)
  - With two transitions possible, two paths in the graph diverge – if only one of them ends in accept, that one should be taken, but we will not know until later which one it is, if any
- Still, the family of languages recognized by these two classes of automata is the same
  - That is, the *regular languages*

NFA, DFA equivalence

- We can demonstrate this equivalence by constructing mappings between NFA, DFA and reg. ex.
- Reg. ex. turn into NFA because there is an NFA construct for every element of basic reg. ex. (character classes, selection, Kleene closure)
  - A class of N characters becomes N arcs with one char. Each
  - Selection is constructed inductively: the NFA of one alternative and the NFA of the other are connected by introducing start and end states with transitions-on-nothing (epsilon) at the front and back
  - Zero-or-more is similarly created with a back arc from the tail of a construct to its beginning, and an epsilon arc from start to an end state
- This is the *McNaughton-Thompson-Yamada* algorithm
  - Formerly known as *Thompson's Construction,* but we wouldn't want to sell McNaughton and Yamada short.

NFA, DFA equivalence

- Turning an NFA into a DFA is a matter of taking sets of states reachable on no input, and lumping them together into new states
  - The *epsilon-closure* of a state is the set of states thus reachable
  - All transitions on a symbol from the e-closure of a state implies a new e-closure at its destination
  - These closures are turned into single states of a DFA
- This is the *subset construction*
- There is also an algorithm for direct simulation of NFA, which essentially computes e-closures as we go along
  - Know that it is there / how it operates

NFA, DFA equivalence

- We know now that
  - Regular expressions turn into NFA
  - NFA turn into DFA
- Add to this
  - DFA are already NFA, they just happen to have 0 e-transitions
  - We can turn DFA back into reg.ex. - branches are selection, loops are closures
- Know that these things are the same, be able to pun between them
  - If you feel that it is easier to memorize the systematic algorithms to do so, please go ahead
  - If you see the equivalence by common sense, that is ok too

Minimizing states

- DFA states are equivalent if there is a subset of states which share in and out edges
- These can be merged together without making a difference to the program
- The grouping is a recursive split wherever there are distinguishable states in a group

How do we write programs?

- Use a regular expression library or generator
  - Yes, it's doable by hand
  - It's a waste of effort to do so except in very special circumstances
- On the practical side, we've worked with Lex, know how to deal with it
  - Where are tokens defined?
  - Where does the lexeme go?
  - How are these two transferred to external code?
- It is as important to be able to read and interface to this sort of thing as it is to write it
  - Given a scanner in Lex, know what to do with it, or how to change it

Syntactic analysis (parsing)

- Lexically, a language is just a pile of words
- Syntax gives structure in terms of which words can appear in which capacity
  - Mostly dealt with in terms of sequencing in programming languages
- Context-Free Grammars give a notation to identify this sort of structure, forming trees from streams of tokens
- We have a number of systematic ways to perform this construction
  - None of them do arbitrary grammars
  - Since the languages we analyze are synthetic, the problems can be avoided by designing them so as to be easy to parse
  - It is mostly simpler to devise a different way of expressing something than to adapt the parsing scheme

Ambiguity and CFG

- A single grammar can admit multiple tree representations of the same text
- That makes it *ambiguous,* and it is a problem to computers because they aren't very clever about context (and none can be found in the grammar)
- This cannot really be fixed – if two trees are valid, then they are both valid
- It can be worked around by adding some rule which consistently picks one interpretation over the other

  (Essentially adding a very primitive idea of context)

Parsing

- What happens?
  - Some tree structure is suggested to match the structure of a token stream, and verified to be accurate
  - Verification can be done by predicting the tree and verifying the stream (predictive parsing, top-down)
  - Verification can be done by constructing the tree after seeing the stream, and checking that it corresponds to the grammar (shift/reduce parsing, bottom-up)
- Why does it happen?
  - Grammar is a general theory of language structure, so all our languages contain special cases of it
  - The more generally we can manipulate the common elements of every language, the less trouble it is to describe each particular one

Parsing: how?

- Top-down:
  - Start with no tree, check a little bit of the token stream
  - Expand the tree with an educated guess about which tokens will appear soon
  - Read as many as the guess permits, then guess again until finished
- Bottom-up:
  - Start with no tree, read tokens onto a stack until they form the bottom/left corner of a tree (shift)
  - Pop them off, and push the top of their sub-tree instead (to remember the part which was already seen) (reduce)
  - Build the next sub-tree in the same way
  - When the sub-trees form a bigger sub-tree, reduce that too
  - Keep going until only the root of a valid tree is left on stack

What we need for top-down

- The grammar must conform so that
  - A prediction can be made by looking a small number of tokens ahead (lookahead)
  - A prediction leads to consuming some tokens, so that the small set which give the next prediction will be different from the ones which gave this one (no left-recursive constructs)
- If it is impossible to discriminate between two constructs because the lookahead is too short, *left factoring* splits the work of one prediction into two predictions with no common part
- If left recursion is present, it can be eliminated systematically
  - Note: neither of these are ambiguities – there is still a unique correct interpretation, the problem lies in how to reach it algorithmically.

Predictive parser construction

- Scheme works by *recursive descent*
  - Make prediction for (nonterminal, lookahead) pair
  - Extend tree
  - Recursively traverse new subtree, until nonterminal is encountered
  - Repeat procedure
- The corresponding grammar class is called LL(k)
  - **L**eft-to-right scan (tokens appear in reading order)
  - **L**eftmost derivation (1^{st} child is on the left)
  - **k** symbols of lookahead are needed for the prediction
- Practically, k=1 is enough for us
  - Parsing table grows with # of k-long token combinations columns
  - Pred. parsing is useful because it is easy, less point when it gets hard

Predictive parser construction

- The parsing table is easier to construct after finding the FIRST, FOLLOW properties of nonterminals
  - Really, relations on intermediate forms, but knowing them for the nonterminals alone simplifies reasoning about the grammar
- Knowing these, deriving a parsing table is a matter of following simple rules
- Knowing the parsing table, constructing code is a matter of following simple rules
  - Again: if you are given to memorizing algorithms, that's an easy way
  - If you feel that you see how the principles work, there will be no questions asking you to recall specific pseudocode from the Dragon
- Learning to do this is practice & repetition
- Not learning to do this is ill advised

What we need for bottom-up

- Bottom-up parsing is a little more general, it doesn't mind left-recursion
  - There *are* still grammars which are LL-but-not-LR for given lookaheads, but they are constructed with the purpose of proving a point, rather than being helpful
- Still, grammars need to be free of conflicts
  - *Shift/reduce conflicts* arise when the r.h.s. of a production appears on stack, but shifting some more symbols could create a different r.h.s.
  - This is analogous to the left-factoring scenario, and can be decided by choosing a favorite production to go for (multiply-first, longest-match-first, or similar)
  - *Reduce/reduce conflicts* arise when the stack state is the r.h.s. of multiple productions, and the parser cannot choose which one
  - This is a symptom of an ambiguity in the language, and strongly indicates that the grammar should be rewritten

Bottom-up basics

- The productions of a grammar imply a number of *items*, which are the productions themselves + an indicator of how far the r.h.s. has been parsed already
- The *closure* of an item results from expanding the nonterminal just after the I-am-here symbol in all the ways it can possibly be expanded
- The LR(0) automaton results from starting with the first production, and creating states from the closure of items
- Next-states and transitions follow from shifting the I-am-here marker one symbol forward (thereby changing the item)
- When the marker is at the end of a production, a reduction happens, and the parser backtracks to where it can start a new construct.

SLR, LR(1), LALR

- The LR(0) method in itself is overly restrictive: constructs with an optional tail cause shift/reduce conflicts
- SLR is the simplest modification: add a symbol of lookahead, and select whether to shift or reduce based on the FOLLOW set of the nonterminal
- LR(1) is more general, adds lookahead symbol to items, increasing number of states
- LALR strikes tradeoff, taking LR(1) approach and merging states which are identical up to the lookahead

How do we write programs?

- For bottom-up parsing, we've been using Yacc
- Translating a grammar into an automaton/table is a fairly straightforward operation
- Transforming it into a program is just as sensibly left to a generator
- Know how to read and write Yacc specifications

Semantics

- Semantics attach meaning to syntactic constructs
- Syntax-directed definition addresses the matter by attaching semantic rules to grammar specifications
- Influenced by the parsing scheme chosen:
  - Bottom-up → synthesized attributes
  - Top-down → inheritance from above/left
- Type of a variable is typically the sort of information we are attaching

Symbol tables

- Symbol tables connect the names of programmer-defined entities to their occurrences in the syntax tree
- A fundamental thing to associate with them is their type
- A *type-safe* program contains only combinations of compatible entities
  - *Strong* type systems permit only this, check and enforce it
  - *Weak* type systems relax the requirements
- Tradeoff: there are type-safe programs which cannot be automatically recognized as such
  - How many programs to allow?

Symbol tables

- Symbol tables are frequently used, require fast insert and lookup
- Three implementations suggested:
  - Array (not very useful, fixed finite set of allowed symbols)
  - Linked list (fast insertion, avg. lookup time of ½ the list length)
  - Hash table (better balance between lookup and insertion)
- How do we write programs?
  - Compute mapping from arbitrary length strings to fixed-length checksums
  - Make fixed-length array, select slot by checksum modulo length
  - Resolve conflicts by rooting linked lists in array

Type checking

- Type comparison is not a simple equality, because some types can be converted into each others' representations
- Valid conversions can be seen as inference rules in restricted natural semantics
- Deriving a judgment on the equivalence of types is constructing a proof tree based on these rules
- Don't be put out if you see one

Natural semantics

- We made a sidebar on how similar rules can characterize the execution of a program
- Attaching execution state to rules per type of statement gives a semantic specification of the language
- Program execution thus maps to a derivation of a tree also
- Don't be put out if you see this either
  - i.e. know what it means

Memory management

- Exiting the front end, we've examined what the requirements of the executing program are
- Specifically, in order to turn it into a *process image*, we need to know what one looks like
- Processes have
  - Code and an instruction counter
  - Initialized data
  - A stack
  - A heap
- Variables need to be laid out in this image in order for the code to access them correctly

Memory management & scope

- At the source program level, the location of a variable in memory is determined by where it is available in the source program
- Local things go on a stack, thrown away at the end of a scope
- Global things go directly in the process image
- Heap things never occur explicitly at the lower level, so code must be written or generated to manage them in terms of other variables which hold their references
  - Pointers or references

Objects

- We took a quick look at the implementation of objects
- The cornerstone is the need for run-time information before a function call can be resolved
- *Dispatch vectors* add a level of indirection, by specifying where to find the address of a function given a variable of a known type (instead of resolving it directly)
  - Classes can have dispatch vectors constructed at compile time, from type information
  - Interfaces specify only a (partial) layout of a dispatch vector, and disappear at compile time
  - Abstract classes mix constraints from the two
- Run time system must support this
  - Either as a general library loaded at run time to handle the housekeeping, or as code inserted by the compiler

So far, so good

- As far as I am concerned, these are the essentials we have covered from the front end
- If you have a decent grasp of what all this means, you're in good shape