NTNU – Trondheim
Norwegian University of
Science and Technology

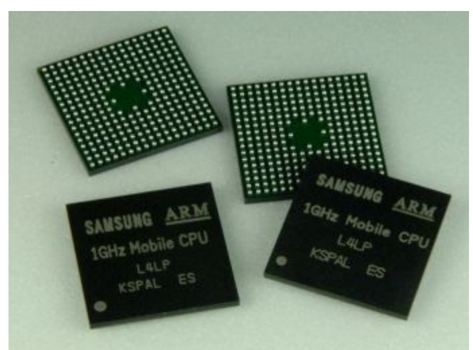
Three-address code (TAC)

On our way toward the bottom

We have a gap to bridge:

Source program

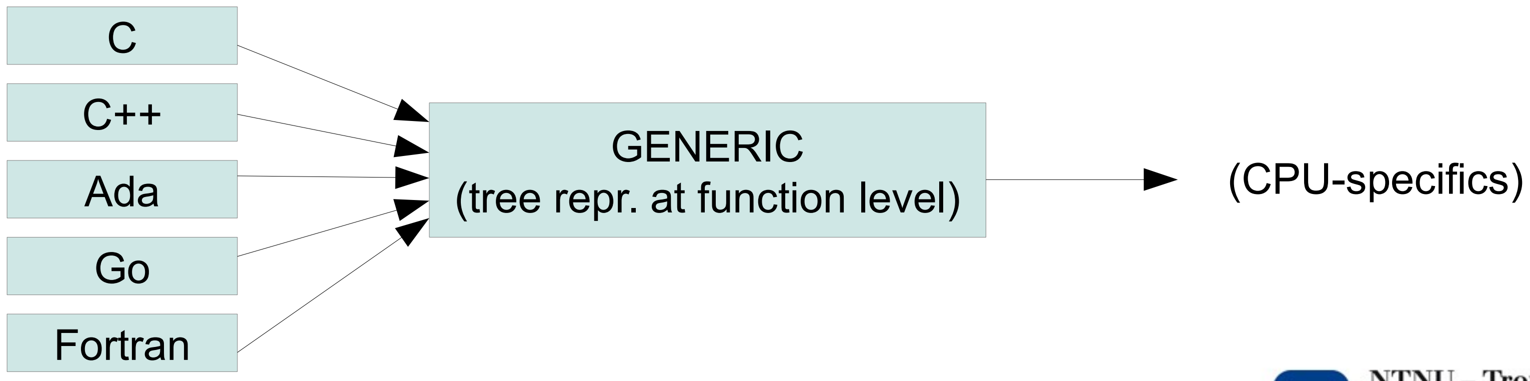
- Words
- Grammar
- Semantics



Program should do the same thing on all of these, but they're all different...

High-level intermediate representation (IR)

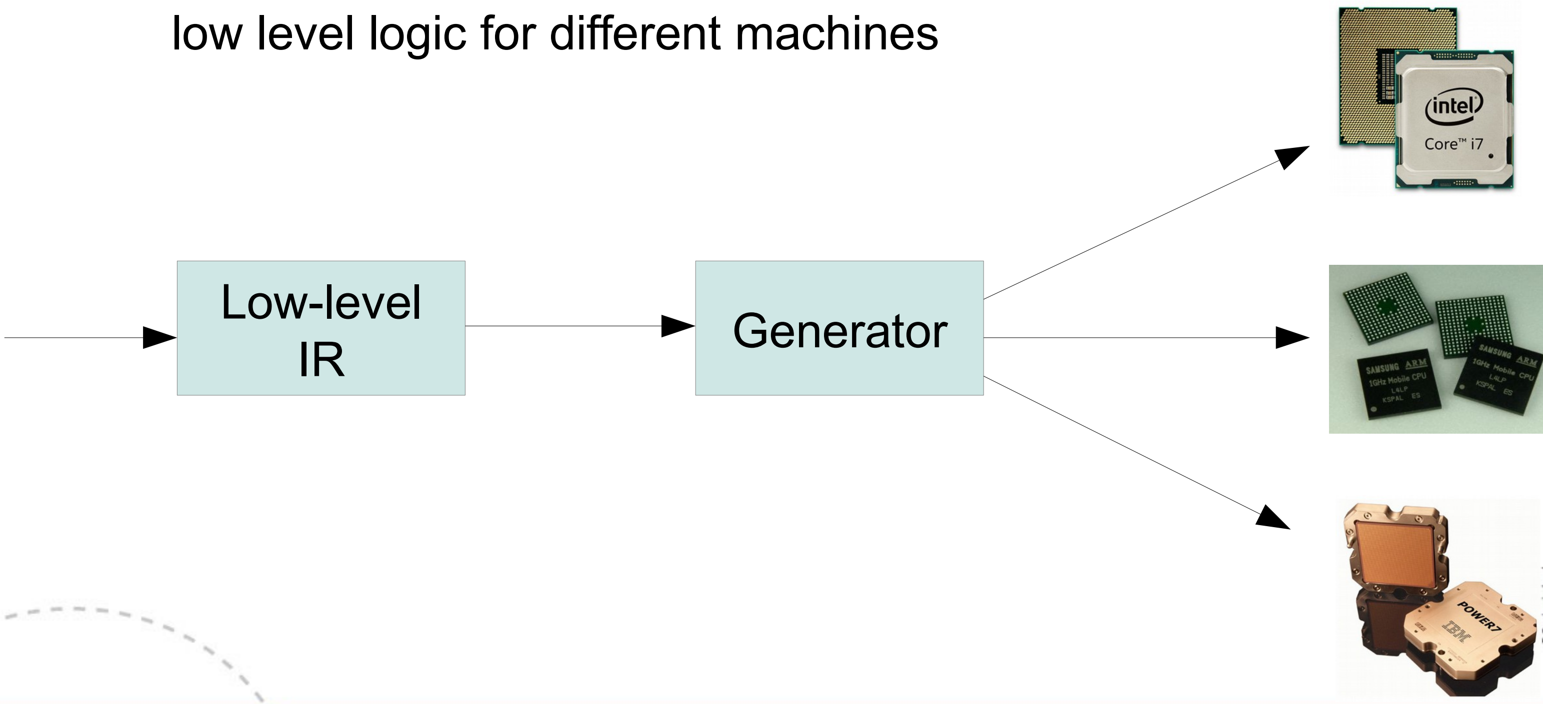
- Working from the syntax tree (or similar), we can capture the program's meaning without hardware details
- If we generalize the representation a bit, we can even liberate it from the specific syntax of the source language
- The main GCC distribution gives you several front-ends (scan/parse/translate) which target the same IR



There are more, but they're not part of the main distribution (yet)...

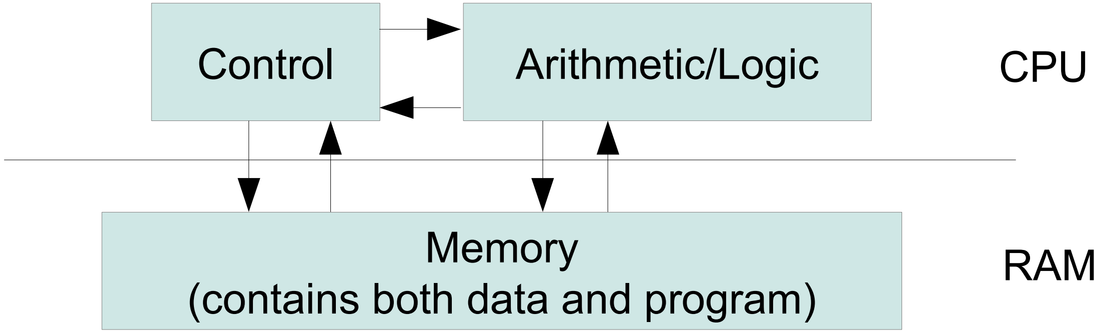
From the other end

- CPU-specific details go into things like how to store addresses, how many registers there are, if any of them have special purposes, etc. etc.
- They all have pretty similar sets of operations, though
- With an abstraction for that, we can re-use most of the low level logic for different machines



Stored-program computing

- If we ignore their implementation details, practically every* modern CPU looks like** a *von Neumann* machine, ticking along to a clock that makes it periodically
  - Fetch an instruction code (from a memory address)
  - Fetch the operands of the instruction (from a memory address)
  - Execute the instruction to obtain its result
  - Put the result somewhere clever (into a memory address)



* research contraptions and exotic experiments notwithstanding

** note that they aren't actually made this way anymore, but emulate it for the sake of programmability

There are only two things to handle

- Instructions for the control unit
- Data for the arithmetic/logic unit
  - Instructions and data are both found at memory addresses, but we can use symbolic names for those
  - Labels for instructions
  - Names for variables
- It's handy to sub-categorize the instructions into

Binary operations
Unary operations
Copy operations
Load/store operations

↑

Math, logic,
data movement

Unconditional jumps
Conditional jumps
Procedure calls

↑

Control flow

TAC is a low-level IR

- It's "three-address" because each operation deals with at most three addresses:

Binary operations:	a = b OP c	OP is ADD, MUL, SUB, DIV…
Unary operations:	a = OP b	OP is MINUS, NEG, …
Copy:	a = b	
Load/store:	x = &y	address-of-y
	x = *y	value-at-address-y
	x[i] = y	address+offset
	...	

TAC is a low-level IR

- Control flow gets the same treatment:

Label:	L:	← named adr. of next instr.
Unconditional jump:	jump L	← go to L and get next instr.
Conditional jump:	if x goto L	← go to L if x is true
	ifFalse x goto L	← go to L if x is false
	if x < y goto L	← comparison operators
	if x >= y goto L	
	if x != y goto L	
	…	
Call and return:	param x	← x is a parameter in next call
	call L	← almost like jump (more later)
	return	← to where the last call came from

Internal representation

- With at most three locations in each operation, they can be written as entries in a 4-column table (quadruples):

op	arg1	arg2	result
mul	x	x	t1
mul	y	y	t2
add	t1	t2	t3
copy	t3		z

- This is one (possible) translation of z = (x*x) + (y*y)

It can be trimmed down still

- Three columns (triples) suffice if we treat the intermediate results as places in the code
- We could decouple the instruction index from the position index (indirect triples)

(Instr. #)	op	arg1	arg2
(0)	mul	x	x
(1)	mul	y	y
(2)	add	(0)	(1)
(3)	copy	z	(2)

One can imagine any number of implementations,
the TAC part is that each instruction deals with 3 locations...

Static Single Assignment

- Programs are at liberty use the same variable for different purposes in different places:

```
z = (x*x) + (y*y);    // Get a sum of squares
if ( z > 1 )  // We're only interested in distances > 1
    z = sqrt(z);      // Get the distance from (0,0) to (x,y)
```

  - A compiler might make use of how z plays two different parts here
  - It can also introduce as many intermediate variables as it likes:

    $z_1 = (x*x) + (y*y);$

    if ($z_1 > 1$)

        $z_2 = \mathrm{sqrt}(z_1);$

    $z_3 = \Phi\ (\ z_1,\ z_2\ )$

  - This makes it explicit that z_1 and z_2 are different values computed at different points, and that the value of z_3 will be one or the other
  - *We* can read that from the source code, a *compiler* needs a representation to recognize it

Translations into low IR

- We have two intermediate representations
- We need a systematic way to translate one into the other
- Suppose we let

  e denote a construct from high IR

  T [e]   denote its translation into low IR

  t = T [e]   denote the assigment that puts the outcome of T[e] in t

  to have a notation which can capture nested applications of a translation

Simple operations

- Disregarding how complicated the contents of e1, e2 are, this generally translates

  t = T [e1 op e2]

  into

    t1 = T [e1]

    t2 = T [e2]

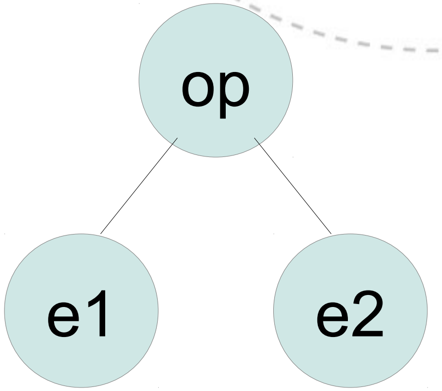
    t = t1 op t2

- In other words,

    First, (recursively) translate e1 and store its result

    Next, (recursively) translate e2 and store its result

    Finally, combine the two stored results



This linearizes the program

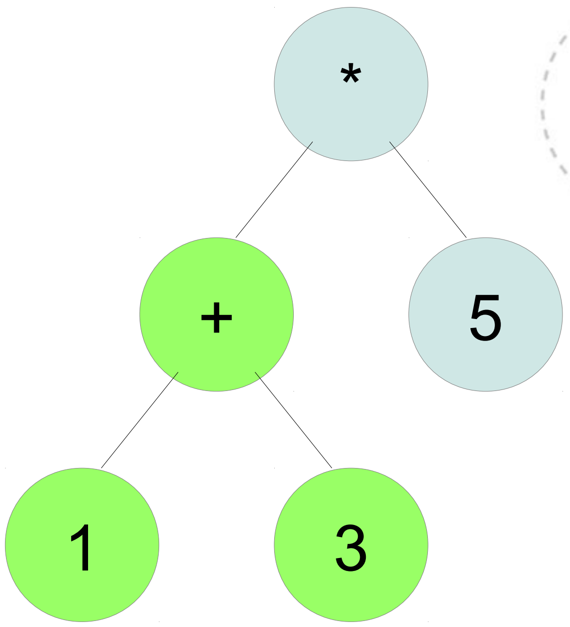
- In terms of a syntax tree, we're laying out its parts in depth-first traversal order:

  t1 = 1

  t2 = 3

  t = 1 + 3

  (from the bottom, where arguments are values)



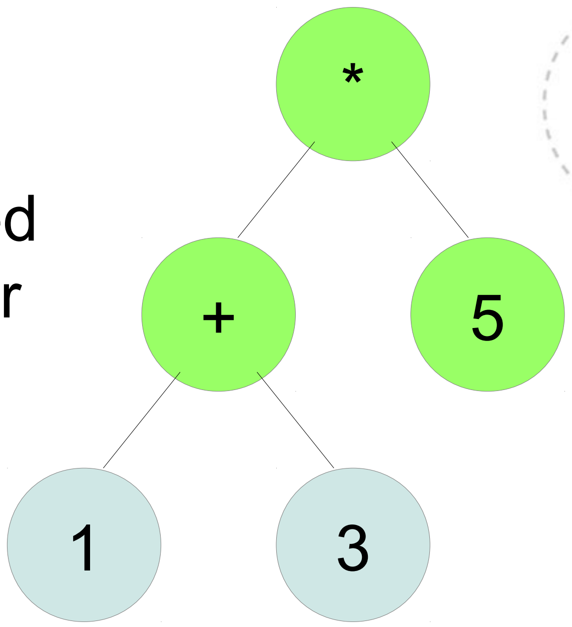
This linearizes the program

- Evaluate one part after another

```
t1 = 1
t2 = 3
t3 = 1 + 3
```

```
t4 = t3
t5 = 5
t6 = t3 * 5
```

Same pattern applied to sub-trees, in order



This linearizes the program

- Combine the local parts which represent sub-trees:

t1 = 1

t2 = 3

t3 = 1 + 3

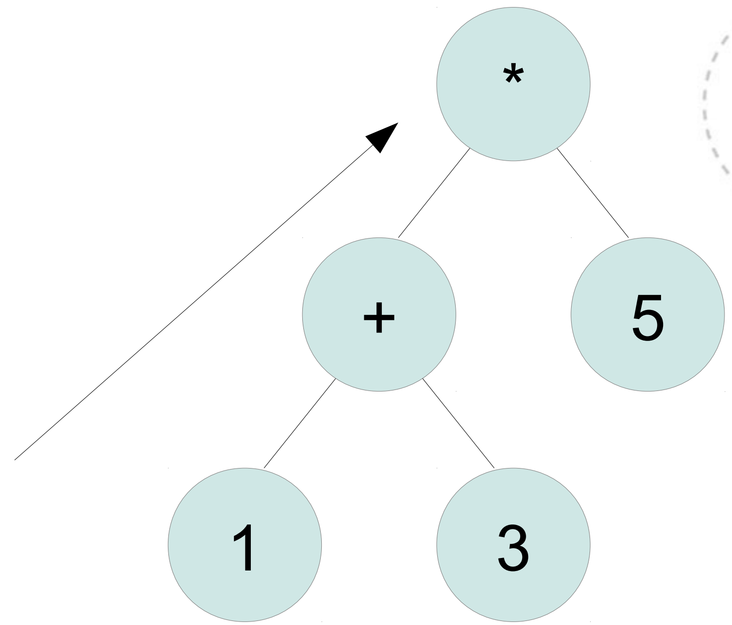
t4 = t3

t5 = 5

t6 = t3 * 5

t = t6

Final result is the whole expression



Nested expressions

- Combine the local parts which represent sub-trees:

```
t1 = 1
t2 = 3
t3 = 1 + 3          T [ 1 + 3 ]
t4 = t3                                T [ (1+3) * 5 ]     t = T[(1+3)*5]
t5 = 5              T [ t3 * 5 ]
t6 = t3 * 5

t = t6
```

Statement sequences

- These are straightforward since they are already sequenced:

  T [s1; s2; s3; …; sn] becomes

      T [s1]
      T [s2]
      T [s3]
      …
      T [sn]

- Just translate one statement after the other, and append their translations in order

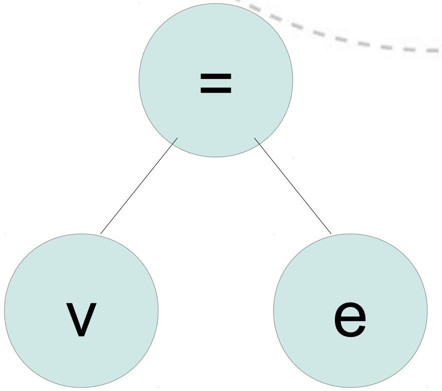
Assignments

- T [v = e] requires us to

  Obtain the value of e
  Put the result into v

  Since e is already (recursively) handled,
  T [v = e] becomes
  t = T [e]
  v = t

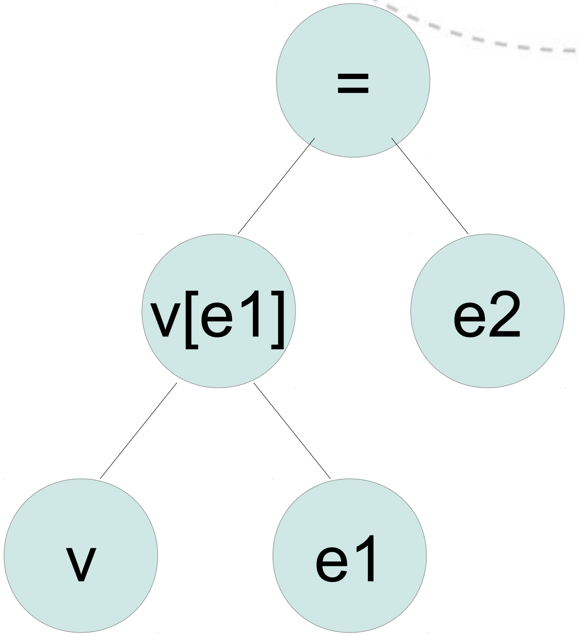
  (or just
  v = T [e]
  if it's convenient to recognize the shortcut)



Array assignment

- T [v[e1] = e2] requires us to
  - Compute the index e1
  - Compute the expression e2
  - Put the result into v[e1]

  t1 = T [e1]
  t2 = T [e2]
  v [t1] = t2

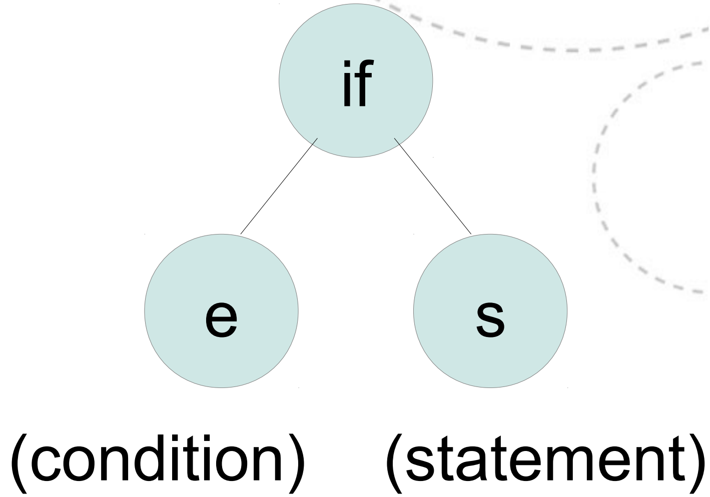


Conditionals

- These require control flow

  T [if (e) then s] becomes

  t1 = T [e]
  ifFalse t1 goto Lend
  T [s]
  Lend:
  (transl. of next statement comes here)

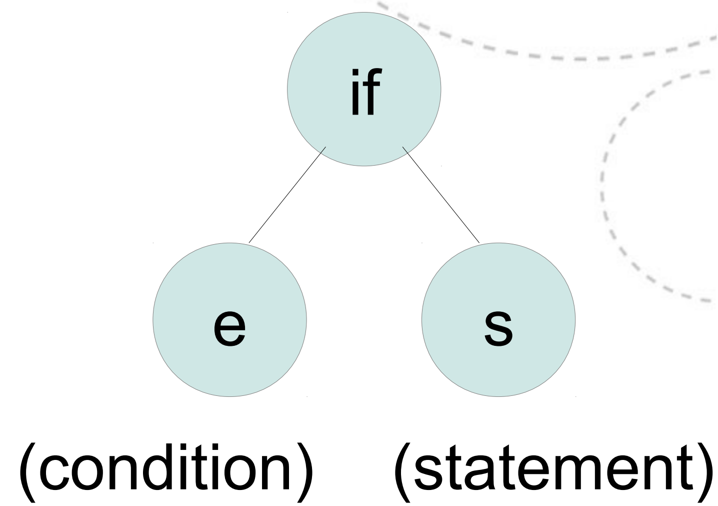


Conditionals

- If e is true, control goes through s
- If e is false, control skips past it

t1 = true *t1 = false*

```
t1 = T [ e ]
ifFalse t1 goto Lend
T [ s ]
Lend:
```



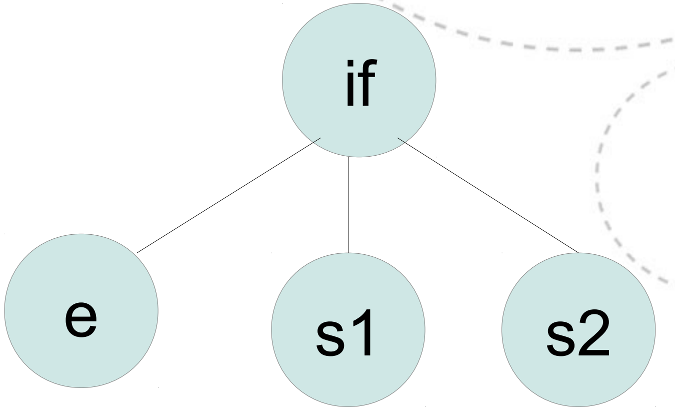
Conditionals + else

- You can probably guess this one:

t1 = true

t1 = false

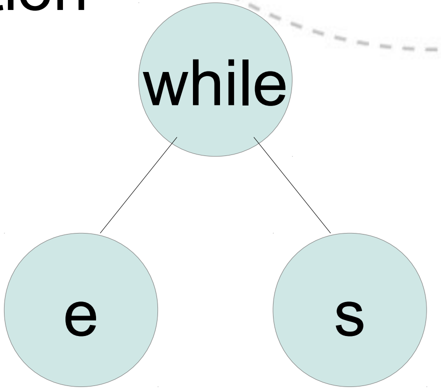
```
t1 = T [ e ]
ifFalse t1 goto Lelse
T [ s1 ]
jump Lend
Lelse:
T [ s2 ]
Lend:
```



Loops (in while flavor)

- The condition must be tested every iteration

  T [while (e) do s] becomes



t1 = true

t1 = false

```
Ltest:
t1 = T [ e ]
ifFalse t1 goto Lend
T [ s ]
jump Ltest
Lend:
```

Loops are loops

- For the sake of completeness,

```
for(i=0; i<10; i++) {
    stuff();
}
```

```
i = 0;
while (i<10) {
    stuff();
    i = i + 1
}
```

```
do {
    stuff();
} while (x);
```

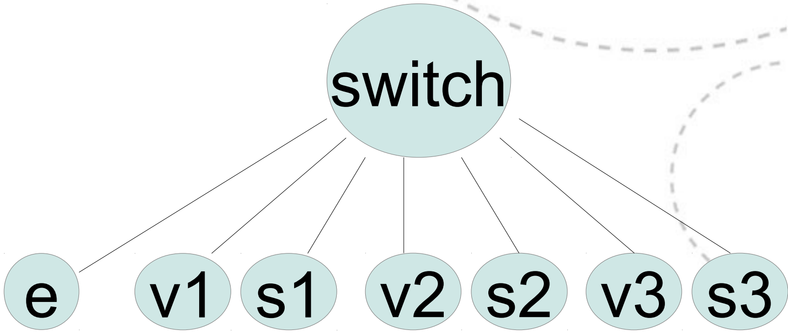
```
stuff();
while (x) {
    stuff();
}
```

Different kinds of loops are equivalent to the point of *syntactic sugar,* whatever form your compiler likes best works also for the others

Switch
(if-elseif style)

T [switch (e) { case v1:s1,…, case vn: sn }]
can become

```
t = T[e]
ifFalse (t=v1) jump L1
T[s1]
L1:
ifFalse (t=v2) jump L2
T[s2]
L2:
…
ifFalse (t=vn) jump Lend
T [ sn ]
Lend:
```



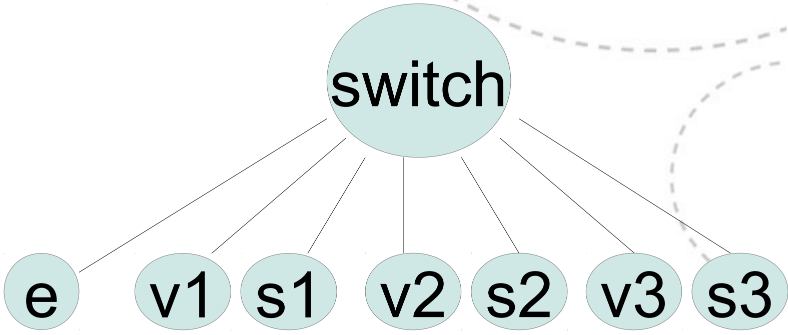
NTNU – Trondheim
Norwegian University of
Science and Technology

Switch
(by jump table)

T [switch (e) { case v1:s1,…, case vn: sn }]

can also become

t = T[e]
jump **table[t]**
Lv1:
T[s1]
Lv2:
T[s2]
…
Lvn:
T [sn]
Lend:



provided that the compiler can generate a table which maps v1,…,vn into the target addresses Lv1, … Lvn for the jumps

(We didn't talk about computed jumps, but labels are just addresses which can be calculated. I mention this because it's probably what you'll see if you disassemble your favourite compiler's interpretation of a switch statement.)

Labeling scheme

- Labels must be unique
- This can handled by numbering the statements that generate them:

```
if ( e1 ) then s1;
if ( e2 ) then s2;

becomes

t1 = T[e1]
ifFalse t1 goto Lend1
T[s1]
Lend1:
t2 = T[e2]
ifFalse t2 goto Lend2
T[s2]
Lend2:
(...and so on...)
```

Nested statements

if (e1) then if (e2) then a = b

requires a little care, nesting (as with expressions) gives

```
t1 = T [ e1 ]
ifFalse (t1) goto Lend1
t2 = T [ e2 ]
ifFalse (t2) goto Lend2
t3 = b
a = t3
Lend2:
Lend1:
```

Outer if (#1)

Inner if (#2)

Statement

The counting scheme must behave like a stack

(to generate end-labels in matching order with construct beginnings)

Those were the basics

- You can surely work out similar patterns for many statement types of your own invention

  or try some from your favourite language
- Things we didn't talk about
  - Redundant code after translation

    (Artifacts we want the low IR to expose, so that we can remove them)
  - Procedure call and return

    (Should be decorated with little background in CPU architecture)
- These are for next time