INF5110 – Compiler Construction

Spring 2016

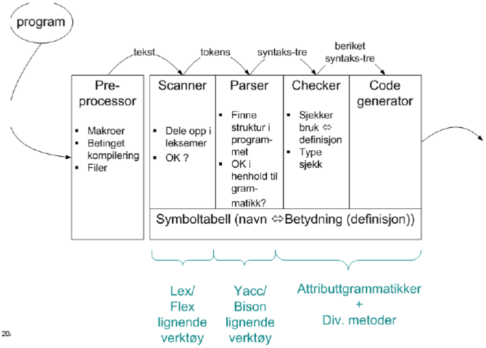
Outline

INF5110 – Compiler Construction

Intermediate code generation

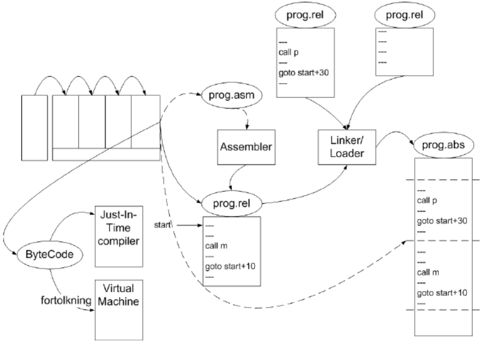
Spring 2016

[a]This section, based on slides from Stein Krogdahl, 2015.



- code generator:
  - 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`[1]
- *byte code* (specifically in Java)
  - a form of intermediate code, as well
  - executable in the JVM
  - in .NET/C♯: *CIL*
    - also called byte-code, but compiled further

[1] .exe-files include more, and "assembly" in .NET even more

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
    - eg. there are *stack frames*, static links, and push and pop, but *exact* layout of the frames is platform dependent
  - platform dependent details:
    - platform dependent code
    - filling in call-sequence / linking conventions

  done in a last step

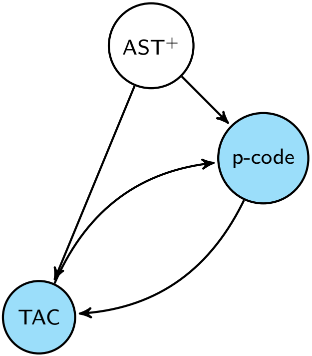
- semi-compiled well-defined format
- platform.independent
- further away from any HW, quite more high-level
- for example: Java byte code (or CIL for .NET and C$^\sharp$)
  - can be interpreted, but often compiled further to machine code ("just-in-time compiler" JIT)
- exectured (interpreted) in a "virtual 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")

Outline

Use of intermediate code

- two kinds of IC covered
  1. three-address code
     - 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", a la Java "byte code")
     - originally proposed for interpretation
     - now often translated before execution (cf. JIT-compilation)
     - intermediate results in *stack* (with postfix operations)
- *many* variations and elaborations for both kinds
  - addresses *symbolically* or represented as *numbers* (or both)
  - granularity/"instruction set"/level of abstract: high-level op's available e.g., for array-access or: translation in more elementary op's needed.
  - operands (still) typed or not
  - ...

- AST here: tree structure *after* semantic analysis, let's just call it AST$^+$ or just simply AST.
- translation AST $\Rightarrow$ P-code: appox. as in Oblig 2
- we touch upon many general problems/techniques in "translations"
- on (important) we ignore for now: *register allocation*



Outline

Three-address code

- common (form of) IR

Basic format

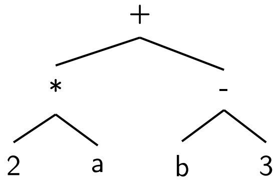
$$x = y \textbf{ op } z$$

- x, y, y: 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 codes (stack-machine code), 2 address codes.
- well-suited for optimizations
- modern archictures often have 3-address code like instruction sets (RISC-architectures)

TAC 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
```

TAC 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 TAC

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: TAC

```
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 c
end
```

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) always a *new temporary*)

Quadruple-representation for TAC (in C)

operasjonskodene

```
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;
```

Hver adresse har denne formen

op:
- opkind (opcode)

addr1:
- kind, val/name

addr2:
- kind, val/name

addr3:
- kind, val/name

P-code

- different common intermediate code / IR
- aka "one-address code"[2] or stack-machine code
- originally developed for Pascal
- remember: post-fix printing of syntax trees (for expressions) and "reverse polish notation"

[2] There's also two-address codes, but those have fallen more or less in disuse.

Example: expression evaluation 2*a+(b-3)

```
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
```

- 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

```
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
```

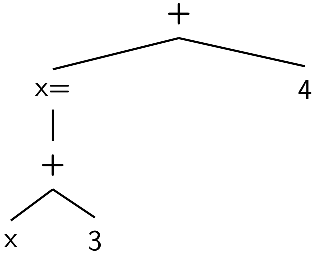
```
1  lda x       ; load address of x
   rdi         ; read an integer, store to
               ; address on top of stack (& pop it)
2  lod x       ; load the value of x
   ldc 0       ; load constant 0
   grt         ; pop and compare top two values
               ; push Boolean result
   fjp L1      ; 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
```

Expression grammar

Grammar

$$
\begin{aligned}
\mathit{exp}_1 &\rightarrow \boldsymbol{id} = \mathit{exp}_2 \\
\mathit{exp} &\rightarrow \mathit{aexp} \\
\mathit{aexp} &\rightarrow \mathit{aexp}_2 + \mathit{factor} \\
\mathit{aexp} &\rightarrow \mathit{factor} \\
\mathit{factor} &\rightarrow \boldsymbol{(}\ \mathit{exp}\ \boldsymbol{)} \\
\mathit{factor} &\rightarrow \boldsymbol{num} \\
\mathit{factor} &\rightarrow \boldsymbol{id}
\end{aligned}
$$

(x=x+3)+4



- goal: p-code as *attributes* 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

- conceptual picture only, not done line that (with A-grammars) in practice!
- not recommended at any rate (for modern/reasonably complex language): code generation *while* parsing

A-grammar for the statement/expression

- dealing here with expressions/assignment: leaves out certain complications
- in particular: control-flow complications
  - two-armed conditionals
  - loops etc.
- but: code-generation "intra-procedural" only, rest is filled in as call-sequences.
- A-grammar:
  - rather simple and straightforwad
  - only 1 *synthesized* attribute: `pcode`

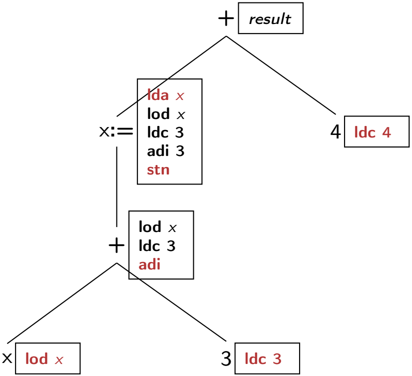
- "string" concatenation: $+\!\!+$ and ^ (inside one command)[3]

productions/grammar rules			semantic rules
exp_1	$\rightarrow$	$\textbf{\textit{id}} = exp_2$	$exp_1.\mathtt{pcode} =$ "lda" ^ $\textbf{\textit{id}}.\mathtt{strval}$ $+\!\!+$ $exp_2.\mathtt{pcode}$ $+\!\!+$ "stn"
exp	$\rightarrow$	$aexp$	$exp.\mathtt{pcode} = aexp.\mathtt{pcode}$
$aexp_1$	$\rightarrow$	$aexp_2 + factor$	$aexp_1.\mathtt{pcode} = aexp_2.\mathtt{pcode}$ $+\!\!+$ $factor.\mathtt{pcode}$ $+\!\!+$ "adi"
$aexp$	$\rightarrow$	$factor$	$aexp.\mathtt{pcode} = factor.\mathtt{pcode}$
$factor$	$\rightarrow$	$(\, exp \,)$	$factor.\mathtt{pcode} = exp.\mathtt{pcode}$
$factor$	$\rightarrow$	$\textbf{\textit{num}}$	$factor.\mathtt{pcode} =$ "ldc" ^ $\textbf{\textit{num}}.\mathtt{strval}$
$factor$	$\rightarrow$	$\textbf{\textit{id}}$	$factor.\mathtt{pcode} =$ "lod" ^ $\textbf{\textit{num}}.\mathtt{strval}$

[3] So, the result is not 100% linear. In general, one should not produce a flat string already.

(x = x + 3) + 4

Attributed tree



Result

```
lda  x
lod  x
ldc  3
adi
stn
ldc  4
adi       ;  +
```

- note: here x=x+3 has side effect *and "return" value* (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 addess, but not the top-value

Overview: p-code data structures

```
type symbol = string

type expr =
    | Var of symbol
    | Num  of int
    | Plus of expr * expr
    | Assign of symbol * expr
```

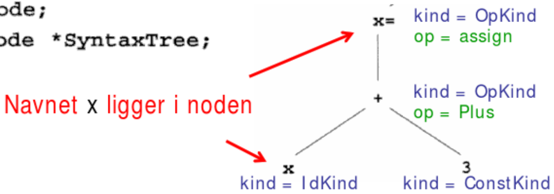
```
type instr   =
(* p-code instructions *)
     LDC of int
   | LOD of symbol
   | LDA of symbol
   | ADI
   | STN
   | STO


type tree = Oneline of instr
   | Seq of tree * tree

type program = instr list
```

- symbols:
  - here strings for simplicity
  - concretely, symbol table may be involved, or variable names already resolved in addresses etc.

```
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);;

let to_program e = linearize (to_tree e);;
```

```
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;
```

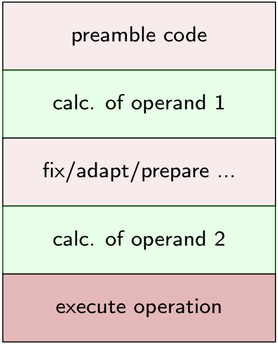


- remember though: there are more dignified ways to design ASTs . . .

Code-generation via tree traversal (schematic)

```
procedure genCode(T: treenode)
begin
 if   T ≠ 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;
```

- 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)



```
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;
```

all
all

```
            case Assign:
              sprintf(codestr,"%s %s",
                              "lda",t->strval);
              emitCode(codestr);
              genCode(t->lchild);    ← rek.kall
              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;
      }
    }
  }
```

- slightly unstructured (since AST is unstructured)

TAC 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: TAC

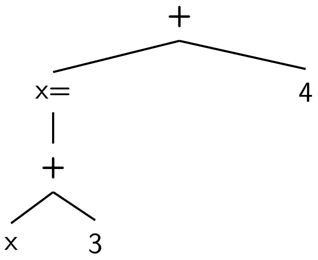
```
read x; {  input an integer }
if  0<x then
  fact := 1;
  repeat
    fact := fact * x;
    x := x -1
  until x = 0;
  write fact { output: factorial
end
```

Expression grammar

Grammar

$$
\begin{aligned}
exp_1 &\rightarrow \boldsymbol{id} = exp_2 \\
exp &\rightarrow aexp \\
aexp &\rightarrow aexp_2 + factor \\
aexp &\rightarrow factor \\
factor &\rightarrow \boldsymbol{(}\ exp\ \boldsymbol{)} \\
factor &\rightarrow \boldsymbol{num} \\
factor &\rightarrow \boldsymbol{id}
\end{aligned}
$$

(x=x+3)+4



Three-address code data structures (some)

```
type symbol = string

type expr =
    | Var of symbol
    | Num of int
    | Plus of expr * expr
    | Assign of symbol * expr
```

```
type mem =
      Var of symbol
    | Temp of symbol
    | Addr of symbol

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 * opera
(* a    := b[i] *)
    | AssignLI of operand * operand * opera
(* a[i] := b *)
    | BranchComp of cond * label
    | Halt
    | Nop

type tree = Oneline of instr
    | Seq of tree * tree
```

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)
```

- similar to the representation for p-code
- again: purely synthesized
- executing expressions/ssignments
  - 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[4]
- evaluation of expressions: *left-to-right* (as before)

[4] In the p-code, the result of evaluating expression (also assignments) ends up in the stack (at the top). This, one does not need to capture it in an attribute.

A-grammar

productions/grammar rules			semantic rules		
exp_1	$\rightarrow$	$\mathbf{id} = exp_2$	$exp_1.\texttt{name}$	$=$	$exp_2.\texttt{name}$
			$exp_1.\texttt{tacode}$	$=$	$exp_2.\texttt{tacode}$ ++ $\mathbf{id}.\texttt{strval}$ ^ "$=$" ^ $exp_2.\texttt{name}$
exp	$\rightarrow$	$aexp$	$exp.\texttt{name}$	$=$	$aexp.\texttt{name}$
			$exp.\texttt{tacode}$	$=$	$aexp.\texttt{tacode}$
$aexp_1$	$\rightarrow$	$aexp_2 + factor$	$aexp_1.\texttt{name}$	$=$	$newtemp()$
			$aexp_1.\texttt{tacode}$	$=$	$aexp_2.\texttt{tacode}$ ++ $factor.\texttt{tacode}$ + $aexp_1.\texttt{name}$ ^ "$=$" ^ $aexp_2.\texttt{name}$ ^ "$+$" ^ $factor.\texttt{name}$
$aexp$	$\rightarrow$	$factor$	$aexp.\texttt{name}$	$=$	$factor.\texttt{name}$
			$aexp.\texttt{tacode}$	$=$	$factor.\texttt{tacode}$
$factor$	$\rightarrow$	$(\ exp\ )$	$factor.\texttt{name}$	$=$	$exp.\texttt{name}$
			$factor.\texttt{tacode}$	$=$	$exp.\texttt{tacode}$
$factor$	$\rightarrow$	$\mathbf{num}$	$factor.\texttt{name}$	$=$	$\mathbf{num}.\texttt{strval}$
			$factor.\texttt{tacode}$	$=$	""
$factor$	$\rightarrow$	$\mathbf{id}$	$factor.\texttt{name}$	$=$	$\mathbf{num}.\texttt{strval}$
			$factor.\texttt{tacode}$	$=$	""

Three-address code data structures (some)

```
type symbol = string

type expr =
    | Var of symbol
    | Num of int
    | Plus of expr * expr
    | Assign of symbol * expr
```

```
type mem =
      Var of symbol
    | Temp of symbol
    | Addr of symbol

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 * opera
(* a    := b[i] *)
    | AssignLI of operand * operand * opera
(* a[i] := b *)
    | BranchComp of cond * label
    | Halt
    | Nop

type tree = Oneline of instr
    | Seq of tree * tree
```

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
}
```

- slightly more cleaned up (and less C-details) than in the book
- "return" of the two attributes
  - name of the variable (a *temorary*): officially returned
  - the code: via *emit*
- note: *postfix* emission only (in the shown cases)

- possible: add `genCode` as *method* to the nodes of the AST[5]
- e.g.: define an abstract `String genCodeTA()` in the `Exp` class (or `Node`, in general all AST nodes where needed)

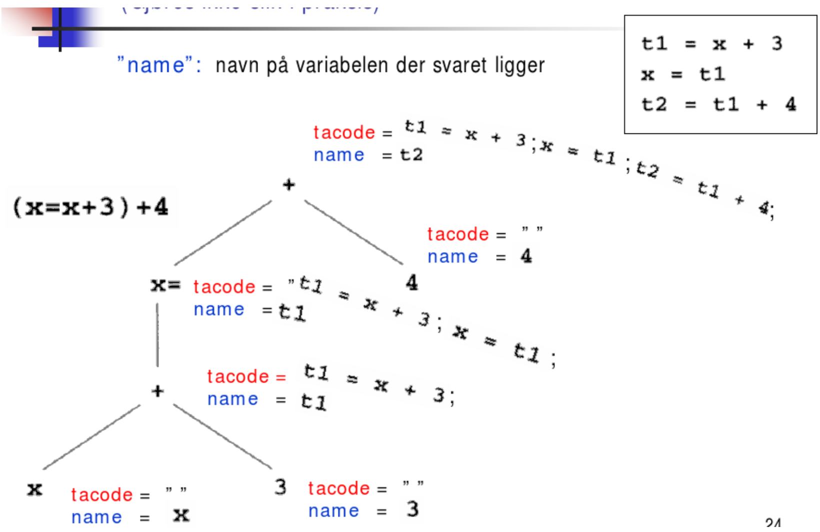
```
String genCodeTA() { String s1,s2; String t = NewTemp();
  s1 = left.GenCodeTA();
  s2 = right.GenCodeTA();
  emit (t + "=" + s1 + op + s2);
  return t
}
```

[5]Whether that is a good design from a compiler-perspective and code maintenance, cluttering 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

```
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



- note: room for optimization

- *illustrated* by transforming P-code → TA-code
- very 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 result in the edges.
- terminology: static simulation seems not widely established
- cf. *abstract interpretation*, *symbolic execution* etc.

- 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

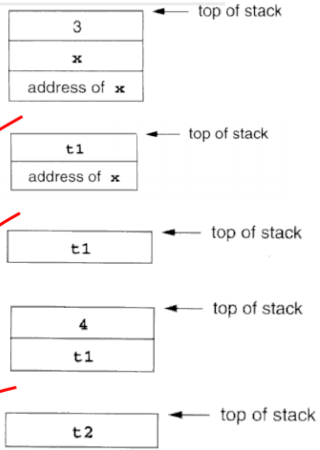
```
(x=x+3)+4
```

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

- also here: simplification, illustrating the general technique only
- main simplification:
  - register allocation
  - but: better done in just another optmization "phase"

Macro for general TAC instruction: a = b + b

```
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 ⇒ TA-code ((x=x+3)+4)

source TA-code

```
t1 = x + 3
x  = t2
t2 = t1 + 4
```

Direct P-code

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi      ;  +
```

P-code via TA-code by macro exp.

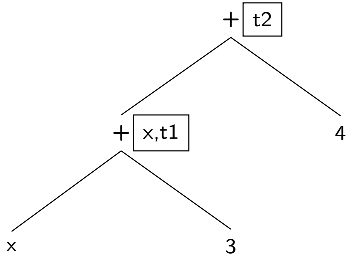
```
;−−− t1 = x + 3
lda t1
lod x
ldc 3
adi
sto
;−−− x = t1
lda x
lod t1
sto
;−−− t2 = t1 + 4
lda t2
lod t1
ldc 4
adi
sto
```

cf. indirect 13 instructions vs. direct: 7 instructions

- as seen: detour lead to sub-optimal results (code size, also efficiency,
- basic deficiency: too many *temporaries* memory traffic etc)
- several possibilities
  - avoid it altogether, of course (but JIT)
  - chance for *cope optimization* phase
  - more clever macro expansion (sketch only)

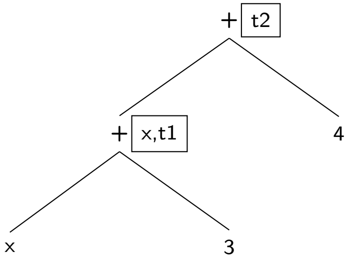
  the more clever macro expansion: some form of static simulation again
- don't macro-expand the linear TAC (via static simulation)
  - brainlessly into another linear structure (P-code), but
  - "statically simulate" it into a more fancy structure (a tree)

- more fancy form of "static simulation"
- results: tree labelled with
  - operator, together with
  - variables/tmporaries containing the results



- note: instruction `x = t1` from TAC: does *not* lead to more nodes in the tree

Tree from TAC



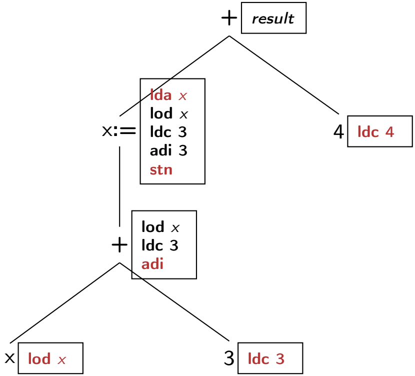
Direct code = indirect code

```
lda  x
lod  x
ldc  3
adi
stn
ldc  4
adi       ;    +
```

- with the thusly (re-)constructed tree

⇒ p-code generation
  - as before done for the AST
  - remember: code as synthesized attributes
- in a way: the "trick" was: reconstruct the essential syntactic tree structure (via "static simulation") from the TA-code

Compare: AST (with direct p-code attributes)



Outline

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

  $\Rightarrow$ 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 there *names* (symbols)
- but in the end: variables are represented by *adresses*
- more complex *address calculations* needed

addressing modes in TAC:

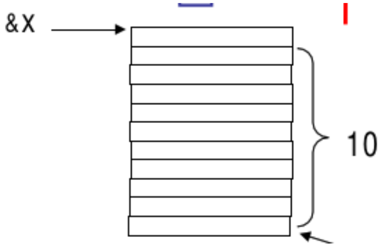
- `&x`: *address* of `x` (not for temporaries!)
- `*t`: *indirectly* via `t`

addressing modes in P-code

- `ind i`: *indirect load*
- `ixa a`: *indexed address*

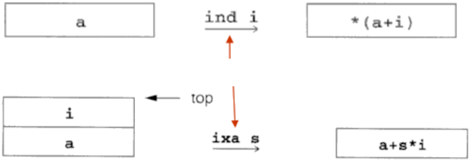
- notationally represented as in C
- "pointer arithmetic" and address calculation with the available numerical ops

```
t1  = &x + 10
*t1 = 2
```



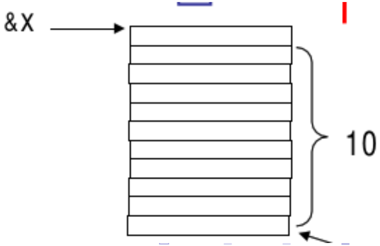
- 3-address-code data structure (e.g., quadrupel): extended

- tailor-made commands for address calculation



- ixa i: integer *scale* factor

```
lda x
ldc 10
ixa 1
ldc 2
sto
```



```
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*
- for example: for `a[i+1]` (with C-style array implementation)[6]

$$a + (i+1) * \mathrm{sizeof(int)}$$

- a here directly stands for the base address

[6] In C, arrays start at an 0-offset as the first array index is 0. Details may differ in other languages.

Array accesses in TA code

- one possible way: assume 2 addition TAC instructions
- remember: TAC can be seen as *intermediate code*, not instruction set of a particular HW!
- 2 new instructions[7]

```
t2 = a[t1] ; fetch value of array element

a[t2] = t1 ; assign to the address of an array element
```

```
a[i+1] = a[j*2] + 3;
```

```
t1     = j * 2
t2     = a[t1]
t3     = t2 + 3
t4     = i + 1
a[t4]  = t3
```

[7] Still in TAC format. Apart from the "readable" notation, it's just two op-codes, say =[] and []=.

Array accesses in TA 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
```

- "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
```

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
```

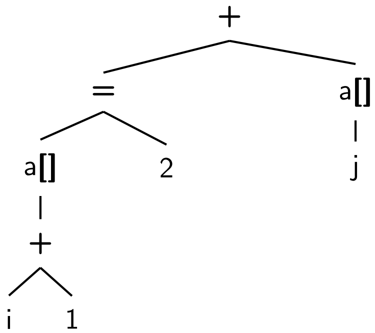
- "expanded" result for a[i+1] = a[j*2] + 3

```
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 the previous grammar

$$
\begin{array}{rcl}
exp & \rightarrow & subs = exp_2 \mid aexp \\
aexp & \rightarrow & aexp + factor \mid factor \\
factor & \rightarrow & (\, exp \,) \mid \textbf{num} \mid subs \\
subs & \rightarrow & \textbf{id} \mid \textbf{id}\, [\, exp \,]
\end{array}
$$

Syntax tree for (a[i+1]=2)+a[j]



```
void genCode (SyntaxTree, 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");
```

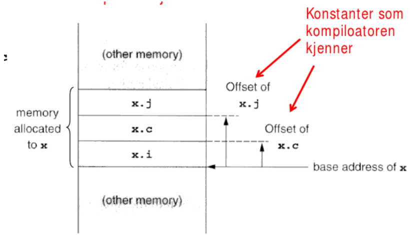
- new code, of course

```
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;
    }
  }
}
```

```
typedef struct rec {
  int i;
  char c;
  int j;
} Rec;
...

Rec x;
```



- fields with (statically known) offsets from base address
- note:
  - goal is: 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*

$\Rightarrow$ call replaced by actual off-set

- note: typically, records are implicitly references (as for objects)
- in (our version of a) TAC: 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,j)
*t1 = *t2
```

Field selection and pointer indirection in TAC

```
typedef struct treeNode {
  int val;
  struct treeNode * lchild,
  * rchild;
} treeNode
...

Treenode *p;
```

assigments involving fields:

```
p -> lchild = p;
p           = p->rchild;
```

```
t1  = p + field_access(*p,lchild)
*t1 = p
t2  = p + field_access(*p,rchild)
p   = *t2
```

- basically same basic “trick”
- make use of `field_offset(x,j)`

```
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
```

- so far: basically *straight-line code*
- intra-procedural[8] control more complex thanks to *control-statements*
  - conditionals, switch/case
  - loops (while, repeat, for ...)
  - breaks, gotos[9], 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. temporaries)

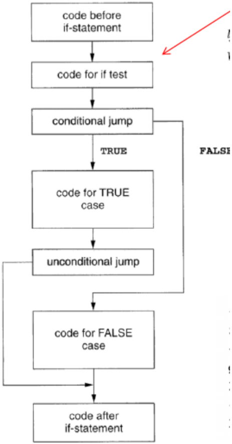
[8] "inside" a procedure. *inter*-procedural control-flow refers to calls and returns, which is handled by calling sequences (which also maintain (in the standard C-like language) the call-stack/RTE

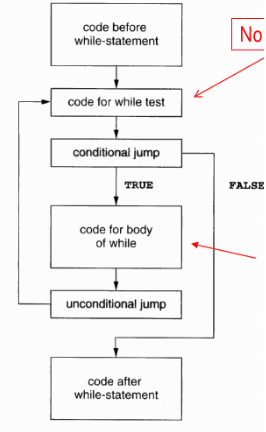
[9] gotos are almost trivial in code generation (as they are basically available at machine code level. Nonetheless, they are "considered harmful", as they mess up everything else in a compiler/language.

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

- challenge:
  - high-level syntax (AST) well-structured (= tree) which implicitly (via its structure) determines complex control-flow beyond SLC
  - low-level syntax (TAC/P-code): rather flat, linear structure, ultimately just a *sequence* of commands

Arrangement of code blocks and cond. jumps





if (E) **then** S_1 **else** S_2

TAC for conditional

```
<code to eval E1 to t1>
if_false t1 goto L1
<code for S1>
goto L2
label L1
<code for S2>
label L2
```

P-code for conditional

```
<code to evaluate E>
fjp L1
<code for S1>
ujp L2
lab L1
<code for S2>
lab L3
```

- 3 new op-codes:
  - **ujp**: unconditional jump ("goto")
  - **fjp**: jump on false
  - **lab**: label (for pseudo instructions)

Jumps and labels: while

while (E) S

TAC for while

```
label L2
<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

```
if ((p!=NULL) && p -> val==0)) ...
```

```
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
```

- 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)

$$
\begin{aligned}
a \textbf{ and } b &\triangleq \textbf{if } a \textbf{ then } b \textbf{ else false} \\
a \textbf{ or } b &\triangleq \textbf{if } a \textbf{ then true else } b
\end{aligned}
$$

- new op-codes
  - **equ**
  - **neq**

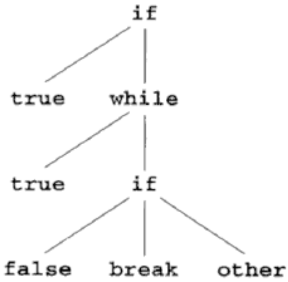
$$\begin{array}{rcl} \textit{stmt} & \rightarrow & \textit{if-stmt} \mid \textit{while-stmt} \mid \textbf{\textit{break}} \mid \textbf{\textit{other}} \\ \textit{if-stmt} & \rightarrow & \textbf{if}\ \textbf{(}\ \textit{exp}\ \textbf{)}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\ \textit{while-stmt}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

- 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
```




```
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
```

- 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
  - labels can (also) be seen as *nodes* in the *control-flow graph*
  - `genCode` generates labels while traversing the AST
  - ⇒ implicit generation of the CFG
  - also possible:
    - separate the CFG first
    - as (just another) IR
    - generate code from there

```
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);  Rek. kall
      lab1 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab1);
      emitCode(codestr);
      genCode(t->child[1],label);  Rek. kall
      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);  Rek. kall
        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);  Rek. kall
      lab2 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab2);
      emitCode(codestr);             Kode for S
      genCode(t->child[1],lab2);     Rek. kall
      sprintf(codestr,"%s %s","ujp",lab1);
      emitCode(codestr);
      sprintf(codestr,"%s %s","lab",lab2);
      emitCode(codestr);
      break;   Label ved slutt av denne while-set
    case BreakKind:
      sprintf(codestr,"%s %s","ujp",label);
      emitCode(codestr);
      break;
    case OtherKind:
      emitCode("Other");
      break;
    default:
      emitCode("Error");
      break;
}
```

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

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

- boolean expressions contain only two (official) values: true and false
- as stated: boolean expressions are often treated special: via short-circuiting
- short-cicruiting especially for boolean expressions in *conditionals* and *while*-loops and similar
- 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
```

TAC

```
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
```

```
void genCode(TreeNode t, String label){        Til denne labelen skal en "break" i kildeprogrammet gå
  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

TA-Code generation for conditionals

```
......
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 ...
```

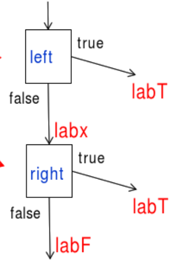
Eneste viktige forskjell er kall på ny metode her. Se neste foil

TA-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 konstrusksjonen 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

For "||":



References I

[Aho et al., 2007] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007).
Compilers: Principles, Techniques and Tools.
Pearson,Addison-Wesley, second edition.

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986).
Compilers: Principles, Techniques and Tools.
Addison-Wesley.