Course Script

INF 5110: Compiler construction

INF5110, spring 2018

Martin Steffen

Contents

Chapter
Intermediate code generation

What is it about?

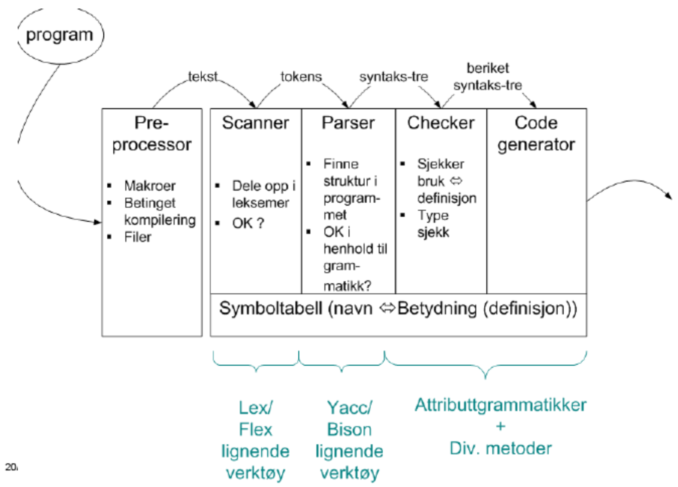
Learning Targets of this Chapter

1. intermediate code
2. three-address code and P-code
3. translation to those forms
4. translation between those forms

Contents

9.1 Intro

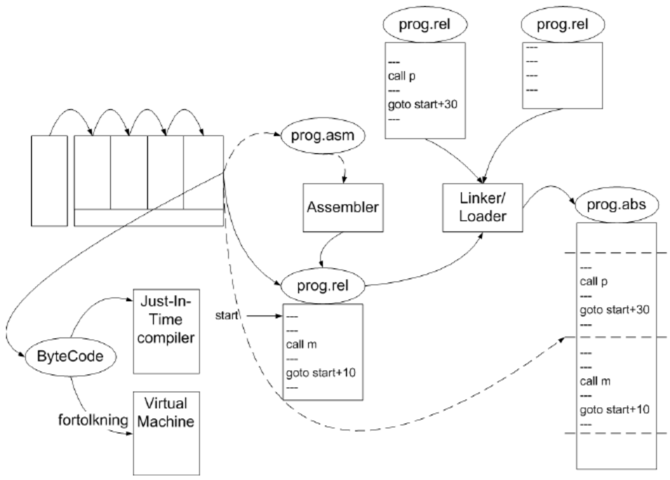
Schematic anatomy of a compiler[1]



[1]This section is 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`[2]
- *byte code* (specifically in Java)
  - a form of intermediate code, as well
  - executable on the JVM
  - in .NET/C♯: *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*)

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

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:
    - 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** (TAIC)
     - 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 a *stack* (with postfix operations)
- *many* variations and elaborations for both kinds
  - addresses *symbolically* or represented 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

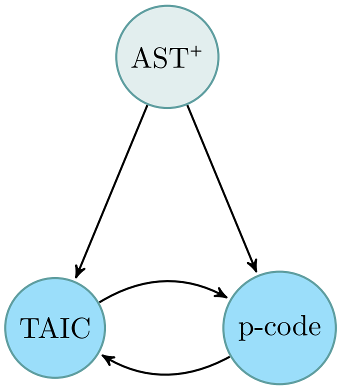
– ...

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 one) we ignore for now: *register allocation*

Picture



9.3 Three address code

Three-address code

- common (form of) IR

TA: Basic format

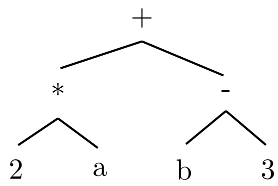
$$x = y \textbf{ op } z$$

- 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
```

TAIC 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 TAIC

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
```

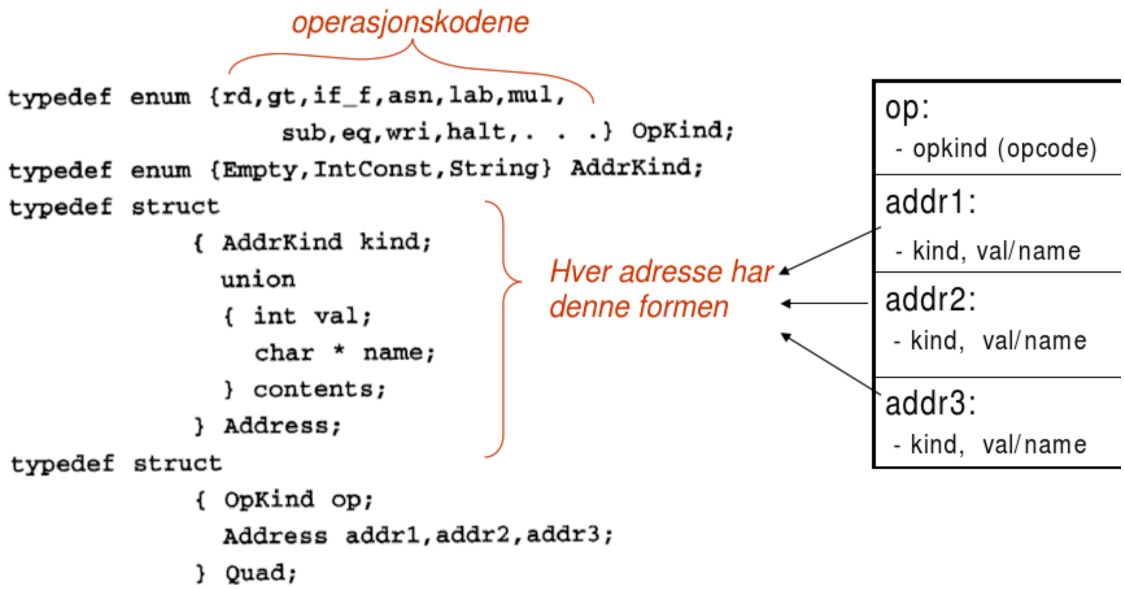
Target: TAIC

```
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 TAIC (in C)



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)

```
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
```

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

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
```

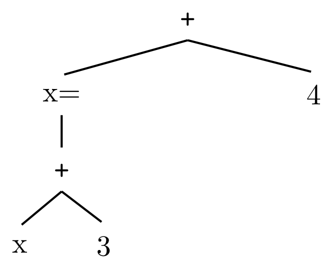
9.5 Generating P-code

Expression grammar

Grammar

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

`(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*.

[4] one can use the a-grammar formalism also to describe the treatment of ASTs, not concrete syntax trees/parse trees.

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`

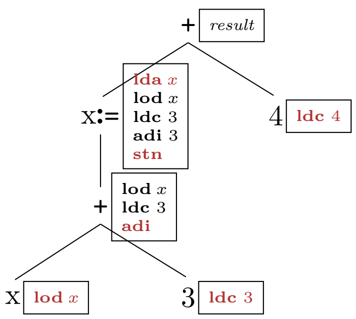
A-grammar

- "string" concatenation: ++ (construct separate instructions) and ˆ (construct one instruction)[5]

productions/grammar rules	semantic rules
$exp_1 \rightarrow \mathbf{id} = exp_2$	exp_1 .pcode = **"lda"**ˆ**id**.strval ++ exp_2 .pcode ++ **"stn"**
$exp \rightarrow aexp$	exp .pcode = $aexp$ .pcode
$aexp_1 \rightarrow aexp_2 + factor$	$aexp_1$ .pcode = $aexp_2$ .pcode ++ $factor$ .pcode ++ **"adi"**
$aexp \rightarrow factor$	$aexp$ .pcode = $factor$ .pcode
$factor \rightarrow (\ exp\ )$	$factor$ .pcode = exp .pcode
$factor \rightarrow \mathbf{num}$	$factor$ .pcode = **"ldc"**ˆ**num**.strval
$factor \rightarrow \mathbf{id}$	$factor$ .pcode = **"lod"**ˆ**num**.strval

(x = x + 3) + 4

Attributed tree



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

"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

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.

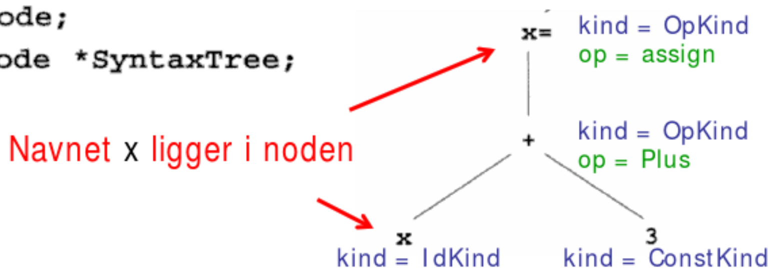
Two-stage translation

```
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)

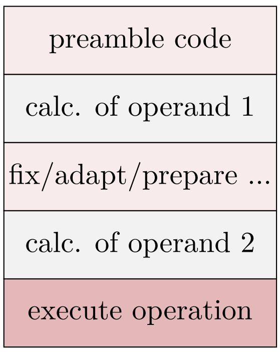
```
procedure genCode(T: treenode)
begin
 if   T ≠ nil
 then
  ``generate code to prepare for code for  left child''  // prefix
  genCode (left child of T);  // prefix ops
  ``generate code to prepare for code for right child''  //infix
   genCode (right child of T); // infix ops
  ``generate code to 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



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);   ← 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;
    }
  }
}
```

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]

[6]That's one possibility of a semantics of assignments (C, Java).

  - 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	→	$\mathbf{id} = exp_2$	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	=	""

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
}
```

[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.

- “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)

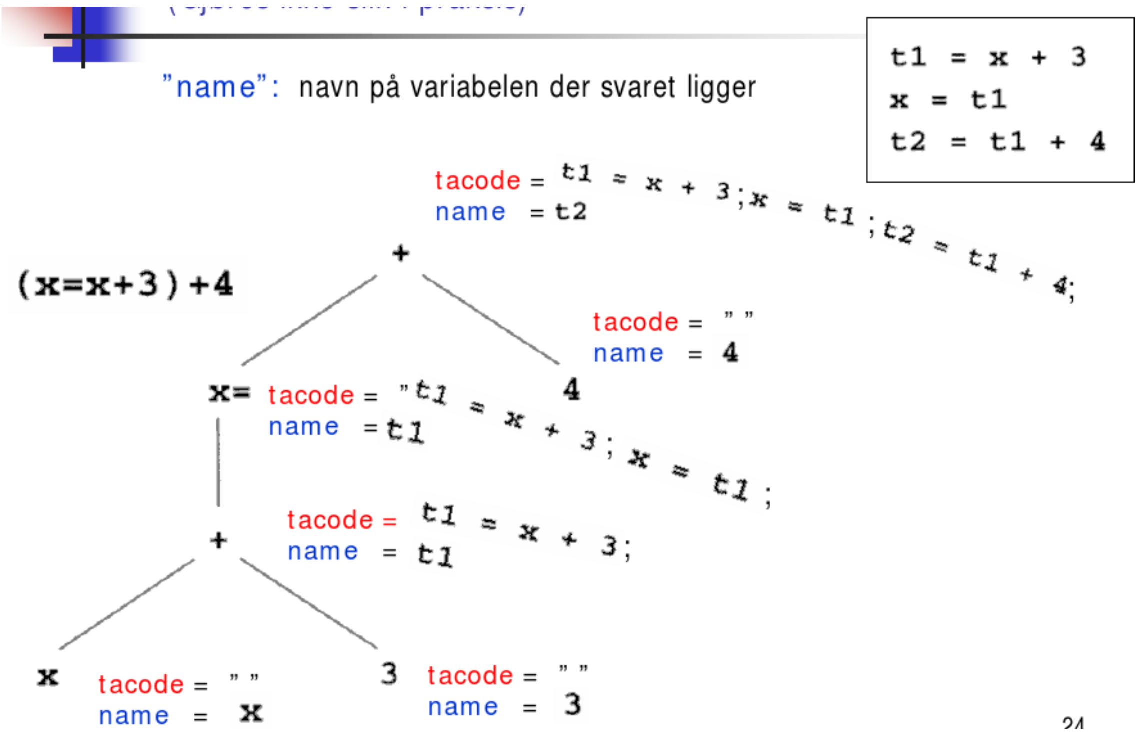
```
String genCodeTA() { String s1,s2; String t = NewTemp();
  s1 = left.GenCodeTA();
  s2 = right.GenCodeTA();
  emit (t + "=" + s1 + op + s2);
  return t
}
```

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



- note: room for optimization

9.7 Basic: From P-code to 3A-Code and back: static simulation & macro expansion

"Static simulation"

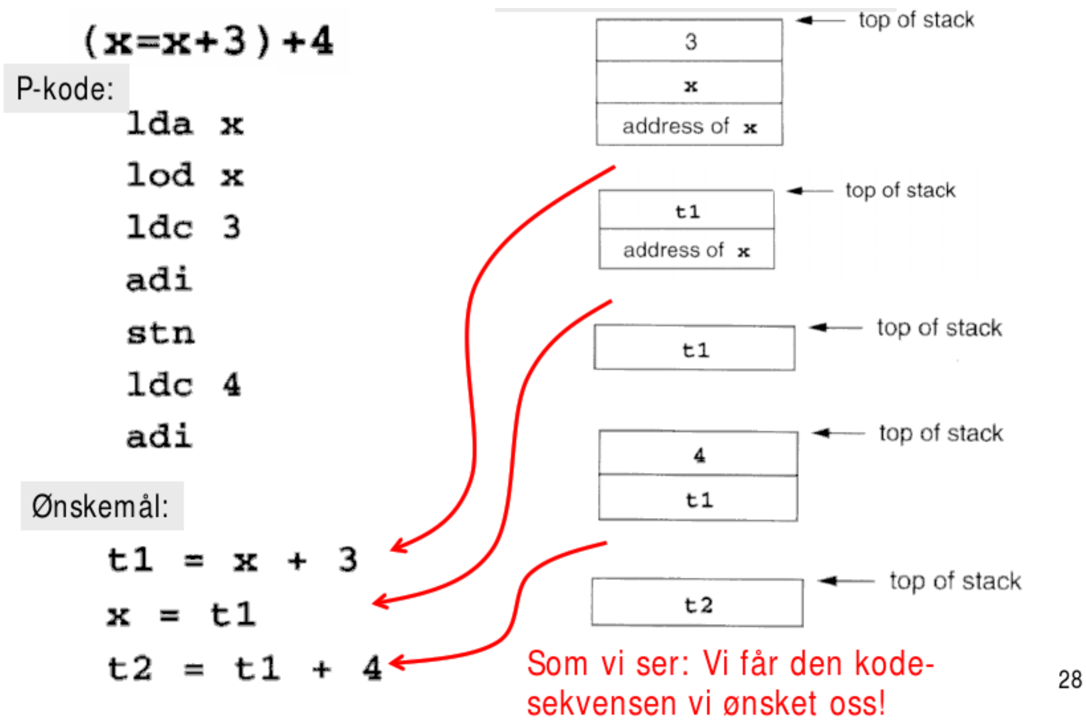
- *illustrated* by transforming P-code ⇒ 3AC
- restricted setting: straight-line code
- cf. also *basic blocks* (or elementary blocks)
  - code without branching or other control-flow complications (jumps/conditional jumps...)
  - often considered as basic building block for static/semantic analyses,
  - 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 ⇒ 3AC 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 ⇒ 3AC: illustration



P-code ⇐ 3AC: 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 3AC 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 ⇐ 3AC ((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 ⇒ 3AC ⇒ p-code

- as seen: *detour* via 3AC 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 3AC
  - brainlessly into another *linear* structure (P-code), but
  - "statically simulate" it into a more *fancy* structure (a *tree*)

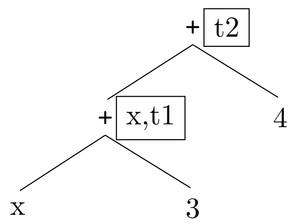
"Static simulation" into tree form (sketch)

- 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
```

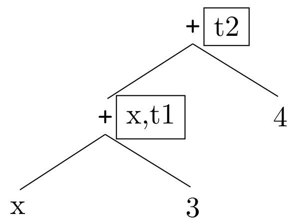
Tree



note: instruction `x = t1` from 3AC: does *not* lead to more nodes in the tree

P-code generation from the generated tree

Tree from 3AIC



Direct code = indirect code

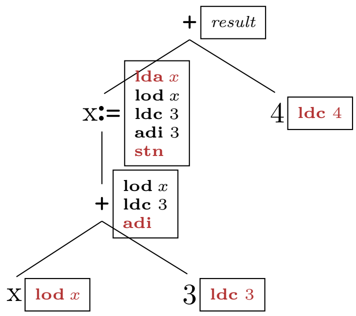
```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi    ;  +
```

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 there *names* (symbols)
- but in the end: variables are represented by *adresses*
- 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*

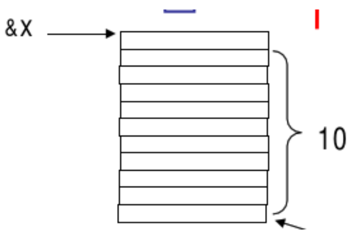
Address calculations in 3AIC: `x[10] = 2`

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

Code

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

Picture

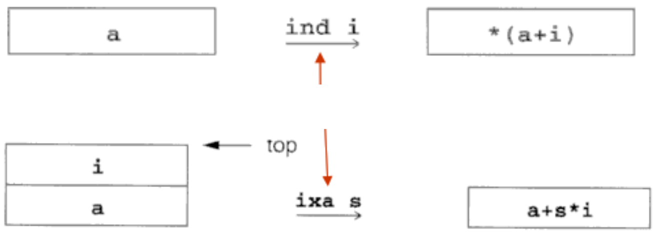


Rest

- 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

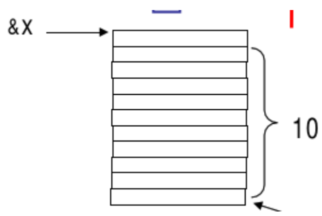


- `ixa i`: integer *scale* factor (here factor 1)

Code

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

Picture



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

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

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
```

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
```

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
```

[9]Still in 3AIC format. Apart from the "readable" notation, it's just two op-codes, say =[] and []=.

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`

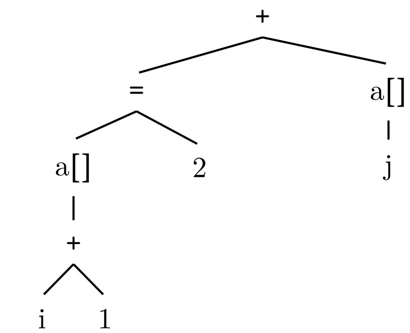
```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
lda a
lod j
ldc 2
mpi
ixa elem_size(a)
ind 0
ldc 3
adi
sto
```

Extending grammar & data structures

- extending the previous grammar

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

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



Code generation for P-code

```
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");
```

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)

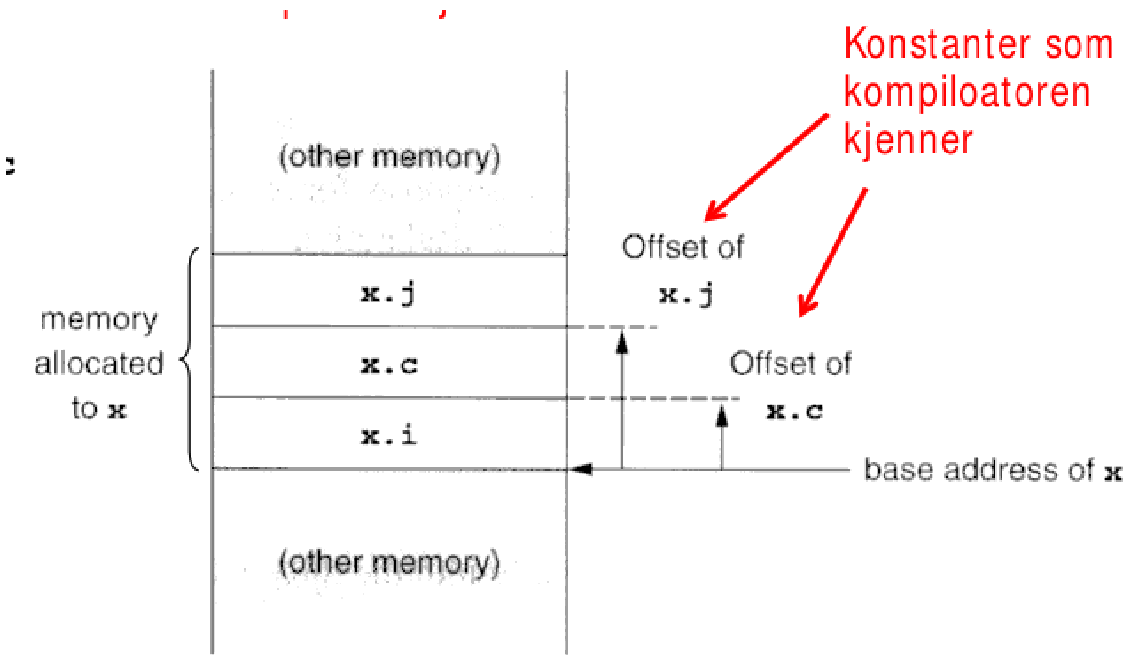
```
    case ConstKind:
      if (isAddr) emitCode("Error");
      else {
        sprintf(codestr,"%s %s","lds",t->strval);
        emitCode(codestr);
      }
      break;
    case IdKind:
      if (isAddr)
        sprintf(codestr,"%s %s", "lda",t->strval);
      else
        sprintf(codestr,"%s %s", "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 humandly 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 side 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 certain hardware instructions where . 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.

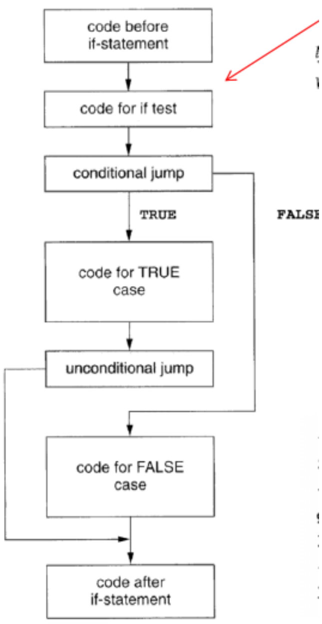
Loops and conditionals: linear code arrangement

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

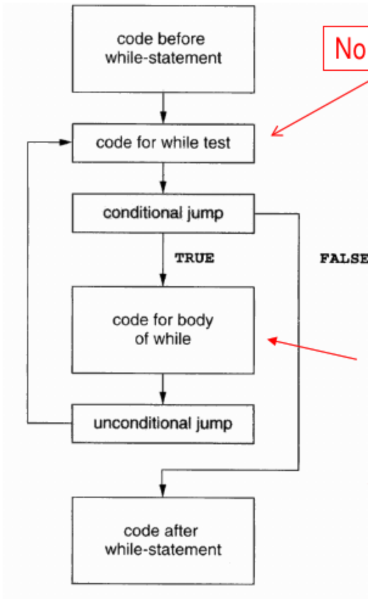
- challenge:
  - high-level syntax (AST) well-structured (= tree) which implicitly (via its structure) determines complex control-flow beyond SLC
  - low-level syntax (3AIC/P-code): rather flat, linear structure, ultimately just a *sequence* of commands

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

$$\textbf{if } (E) \textbf{ then } S_1 \textbf{ else } S_2$$

3AIC for conditional

```
<code to eval E 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 L2
```

3 new op-codes:

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

Jumps and labels: while

$$\textbf{while } (E)\ S$$

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 $\land$ and $\lor$ 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)

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

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

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

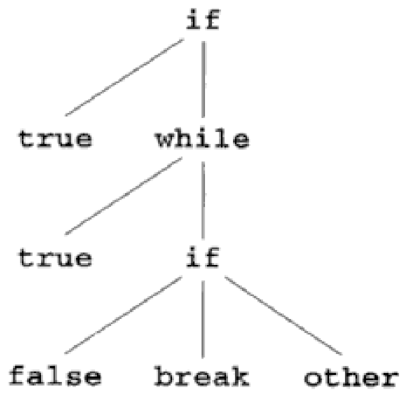
- note: simplistic expressions, only *true* and *false*

```
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
```

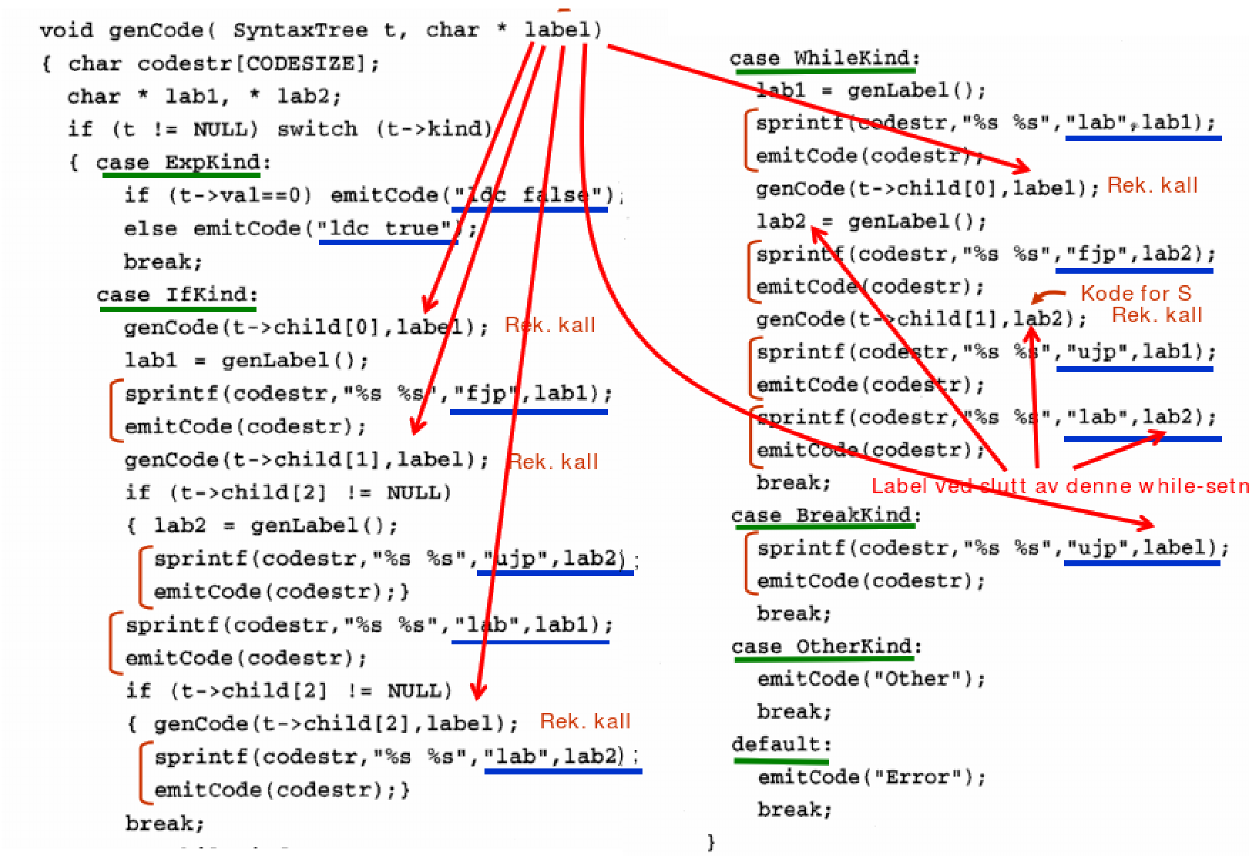
Code generation

- 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)

```
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

Alternative P/3A-Code generation for conditionals

- Assume: **no break** in the language for simplicity
- focus here: conditionals
- not covered of [**?**]

```
......
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
        }
}                     Eneste viktige forskjell er kall på ny metode her. Se neste foil
... more cases ...
```

Alternative 3A-Code generation for boolean expressions

Vi bryr oss ikke med retur-navnet, siden de alltid vil hoppe ut

```
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
```