Chapter 9

Intermediate code generation

Course "Compiler Construction"

Martin Steffen

Spring 2021

Section

Targets

Chapter 9 “Intermediate code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2021

Chapter 9

Learning Targets of Chapter "Intermediate code generation".

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

Chapter 9

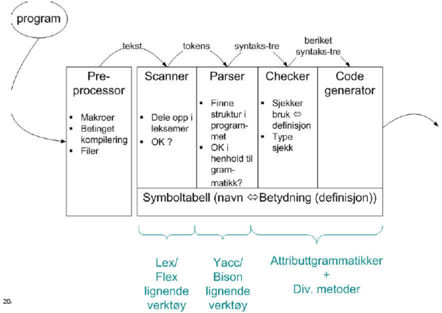
Outline of Chapter "Intermediate code generation".

Section

Intro

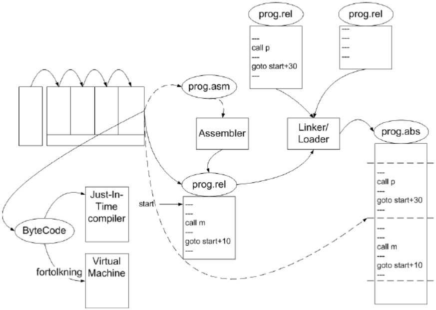
Chapter 9 “Intermediate code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2021

Schematic anatomy of a compiler



- 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: `*.o`
    - 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 on 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
    - 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" (like JVM)
- often: *stack-oriented* execution code (in post-fix format)
- also *internal* intermediate code (in compiled languages) may have stack-oriented format ("P-code")

Section

Intermediate code

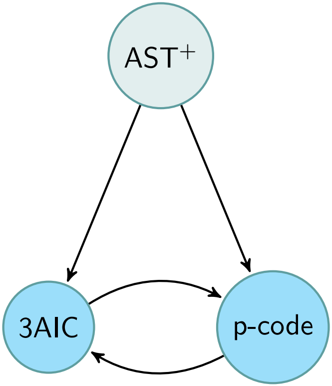
Chapter 9 “Intermediate code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2021

Use of intermediate code

- two kinds of IC covered
  1. three-address code (3AC, 3AIC)
     - generic (platform-independent) abstract machine code
     - new names for all intermediate results
     - can be seen as unbounded pool of maschine registers
     - advantages (portability, optimization . . .)
  2. P-code ("Pascal-code", cf. Java "byte code")
     - originally proposed for interpretation
     - now often translated before execution (cf. JIT-compilation)
     - intermediate results in a *stack* (with postfix operations)
- *many* variations and elaborations for both kinds
  - addresses represented *symbolically* or as *numbers* (or both)
  - granularity/"instruction set"/level of abstraction: high-level op's available e.g., for array-access or: translation in more elementary op's needed.
  - operands (still) typed or not
  - . . .

Various translations in the lecture

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



Section

Three-address (intermediate) code

Chapter 9 “Intermediate code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2021

Introduction

Three-address code is an common format, not just for intermediate code, but also for machine code. The name comes from that fact that some instructions make use of three "addresses". Not all operations use three, some use less, but the most general ones make use of 2 source addresses for the arguments, and one target address for the result. In particular, binary operations that do calculations use 3, like addition or bitwise and. See equation (1).
We mentioned before that our intermediate code does not make use of addresses and registers (which is a common thing to do for intermediate code). That means, the instructions don't literally work with 3 addresses, but rather they involve 3 variables or constants. The code also not only makes use of "ordinary" variables (like the ones that originate from the source code), but the code generation introduces *temporary variables* or *temporaries* for short to store intermediate results. At this phase there is no attempt to economize on the amount of temporaries. An unbounded

Three-address code

- common (form of) IR

TA: Basic format

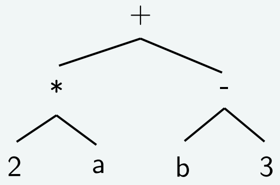
$$x = y \ \mathbf{op} \ z \tag{1}$$

- 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 (or even 0) code (stack-machine code), 2 address code
- well-suited for optimizations
- modern architectures often have 3-address code like instruction sets (RISC-architectures)

3AC example (expression)

2*a+(b-3)



Three-address code

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

alternative sequence

```
t1 = b - 3
t2 = 2 * a
t3 = t2 + t1
```

3AIC instruction set

- basic format: $x = y \,\mathbf{op}\, z$
- but also:
  - $x = \mathbf{op}\, z$
  - $x = y$
- *operators*: $+$,-,*,/, $<$, $>$, `and`, `or`
- `read` x, `write` x
- `label` L (sometimes called a "pseudo-instruction")
- conditional jumps: `if_false` x `goto` L
- t_1, t_2, t_3 .... (or `t1, t2, t3, ...`): temporaries (or temporary variables)
  - assumed: *unbounded* reservoir of those
  - note: "non-destructive" assignments (single-assignment)

Illustration: translation to 3AIC

Source

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

Target: 3AIC

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

Variations in the design of 3A-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 3A *instructions*?
  - quadruples: 3 "addresses" + the op
  - *triple* possible (if target-address (left-hand side) is always a *new temporary*)

Quadruple-representation for 3AIC (in C)

```
typedef enum {rd, gr, 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
```

Section

P-code

Chapter 9 "Intermediate code generation"
Course "Compiler Construction"
Martin Steffen
Spring 2021

P-code

- different common intermediate code / IR
- aka "one-address code"[2] or stack-machine code
- used prominently 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 for intermediate code.

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

```
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
```

Section

Generating P-code

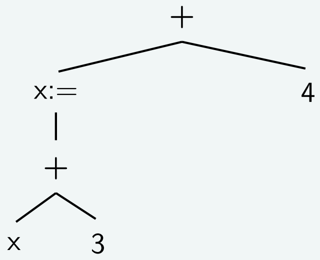
Chapter 9 “Intermediate code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2021

Assignment grammar

Grammar

$$
\begin{array}{rcl}
exp_1 & \to & \mathbf{id} := exp_2 \\
exp & \to & aexp \\
aexp & \to & aexp_2 + factor \\
aexp & \to & factor \\
factor & \to & (\, exp \,) \\
factor & \to & \mathbf{num} \\
factor & \to & \mathbf{id}
\end{array}
$$

`(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[3]

[3] 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`

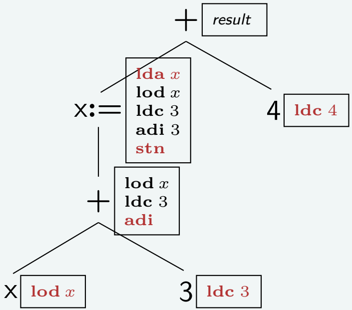
A-grammar

- "string" concatenation: ++ (construct separate instructions) and ˆ (concat one instruction)

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
$factor$	$\rightarrow$	**num**	$factor$ .pcode = "**ldc**"ˆ**num**.strval
$factor$	$\rightarrow$	**id**	$factor$ .pcode = "**lod**"ˆ**num**.strval

(x := x + 3) + 4

Attributed tree



"result" attr.

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

- note: here x:=x+3 has a 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

```
type symbol = string

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

Listing 1: Syntax of the source language (expressions with side effects)

```
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
```

Listing 2: Syntax of the target language

- 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
```

Listing 3: Code generation (interface)

```
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);;
```

Listing 4: Code generation

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;
```

Listing 5: AST in C (for expressions with assignments)

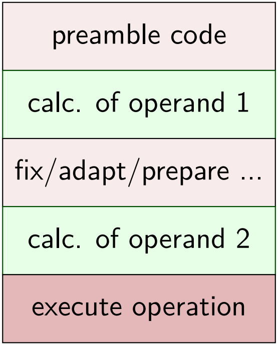
Code-generation via tree traversal (schematic)

```
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;
```

Listing 6: Schematic code generation `GenCode` in C

Code generation from AST+

- main "challenge": linearization
- here: relatively simple
- no control-flow constructs
- linearization here (see a-grammar):
  - string of p-code
  - not necessarily the ultimate choice (p-code might still need translation to "real" executable code)



Code generation

```
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;
    }
  }
}
```

Section

Generation of three-address intermediate code

Chapter 9 “Intermediate code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2021

3AIC 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: 3AIC

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

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   (* &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
```

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)
```

Listing 8: Code generation 3AIC (expressions)

Three-address code by synthesized attributes

- similar to the representation for p-code
- again: purely synthesized
- semantics of executing expressions/assignments[4]
  - 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[5]
- evaluation of expressions: *left-to-right* (as before)

---

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

[5] 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.

A-grammar

productions/grammar rules	semantic rules
$exp_1 \rightarrow \mathbf{id} = exp_2$	$exp_1\,.\mathtt{name} = exp_2\,.\mathtt{name}$ $exp_1\,.\mathtt{tacode} = exp_2\,.\mathtt{tacode} \mathbin{+\!\!+}$ $\mathbf{id}.\mathtt{strval}\,\hat{}\,\text{"="}\,\hat{}\; exp_2\,.\mathtt{name}$
$exp \rightarrow aexp$	$exp\,.\mathtt{name} = aexp\,.\mathtt{name}$ $exp\,.\mathtt{tacode} = aexp\,.\mathtt{tacode}$
$aexp_1 \rightarrow aexp_2 + factor$	$aexp_1\,.\mathtt{name} = newtemp()$ $aexp_1\,.\mathtt{tacode} = aexp_2\,.\mathtt{tacode} \mathbin{+\!\!+} factor\,.\mathtt{tacode} +$ $aexp_1\,.\mathtt{name}\,\hat{}\,\text{"="}\,\hat{}\; aexp_2\,.\mathtt{name}\,\hat{}$ $\text{"+"}\,\hat{}\; factor\,.\mathtt{name}$
$aexp \rightarrow factor$	$aexp\,.\mathtt{name} = factor\,.\mathtt{name}$ $aexp\,.\mathtt{tacode} = factor\,.\mathtt{tacode}$
$factor \rightarrow (\, exp\,)$	$factor\,.\mathtt{name} = exp\,.\mathtt{name}$ $factor\,.\mathtt{tacode} = exp\,.\mathtt{tacode}$
$factor \rightarrow \mathbf{num}$	$factor\,.\mathtt{name} = \mathbf{num}.\mathtt{strval}$ $factor\,.\mathtt{tacode} = \text{""}$
$factor \rightarrow \mathbf{id}$	$factor\,.\mathtt{name} = \mathbf{num}.\mathtt{strval}$ $factor\,.\mathtt{tacode} = \text{""}$

Another sketch of 3AI-code generation

- “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)

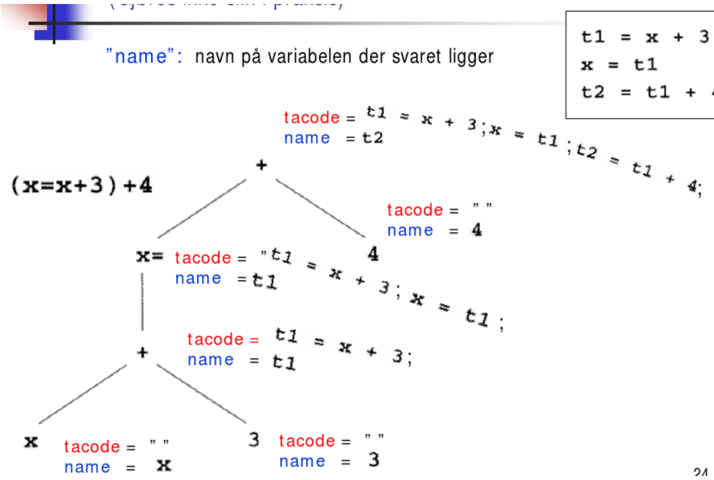
```
String genCodeTA() { String s1,s2; String t = NewTemp();
  s1 = left.GenCodeTA();
  s2 = right.GenCodeTA();
  emit (t + "=" + s1 + op + s2);
  return t
}
```

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

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



- note: room for optimization

Section

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

Chapter 9 “Intermediate code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2021

"Static simulation"

- *illustrated* by transforming p-code ⇒ 3AC
- restricted setting: straight-line code
- cf. also *basic blocks* (or elementary blocks)
  - code without branching or other control-flow complications (jumps/conditional jumps...)
  - often considered as basic building block for static/semantic analyses,
  - e.g. basic blocks as nodes in *control-flow graphs*, the "non-semicolon" control flow constructs result in the edges
- terminology: static simulation seems not widely established
- cf. *abstract interpretation*, *symbolic execution*, etc.

P-code ⇒ 3AIC via "static simulation"

- difference:
  - p-code operates on the *stack*
  - leaves the needed "temporary memory" implicit
- given the (straight-line) p-code:
  - traverse the code = list of instructions from beginning to end
  - seen as "simulation"
    - conceptually at least, but also
    - concretely: the translation can make *use* of an actual stack

From P-code ⇒ 3AIC: illustration

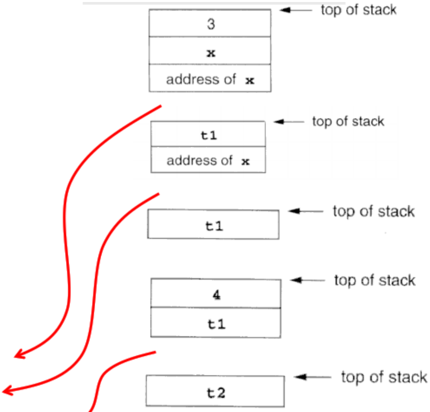
`(x=x+3)+4`

P-kode:

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

Ønskemål:

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



Som vi ser: Vi får den kode-sekvensen vi ønsket oss!

28

P-code ⇐ 3AIC: macro expansion

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

Macro for general 3AIC instruction: `a := b + c`

```
lda a
lod b;    or ``ldc b'' if b is a const
lod c:    or ``ldc c'' if c is a const
adi
sto
```

Example: P-code ⇐ 3AIC ((x:=x+3)+4)

source 3AI-code

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

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
```

cf. indirect 13 instructions vs. direct: 7 instructions

Indirect code gen: source code ⇒ 3AIC ⇒ p-code

- as seen: *detour* via 3AIC leads to sub-optimal results (code size, also efficiency)
- basic deficiency: too many *temporaries*, memory traffic etc.
- several possibilities
  - avoid it altogether, of course (but remember JIT in Java)
  - chance for *code optimization* phase
  - here: more clever "macro expansion" (but sketch only)

  the more clever macro expansion: some form of *static simulation* again
- don't macro-expand the linear 3AIC
  - brainlessly into another *linear* structure (p-code), but
  - "statically simulate" it into a more *fancy* structure (a *tree*)

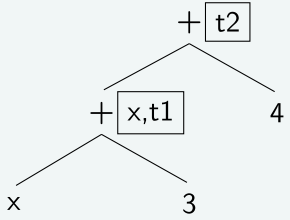
"Static simulation" into tree form (sketch)

- more fancy form of "static simulation" of 3AIC
- *result*: tree labelled with
  - operator, together with
  - variables/temporaries containing the results

Source

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

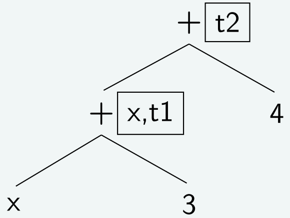
Tree



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

P-code generation from the generated tree

Tree from 3AIC



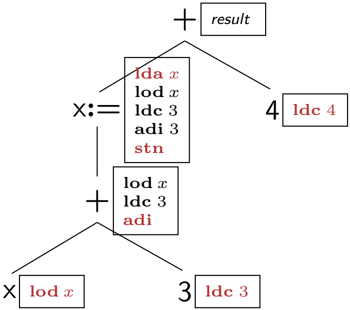
Direct code = indirect code

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

- 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)



Section

More complex data types

Chapter 9 “Intermediate code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2021

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 their *names* (symbols)
- but in the end: variables are represented by *addresses*
- more complex *address calculations* needed

addressing modes in 3AIC:

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

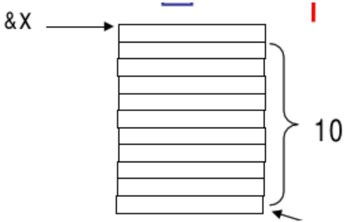
addressing modes in P-code

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

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

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

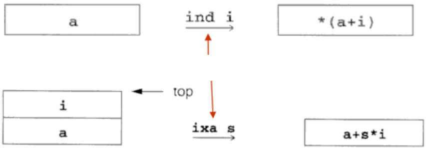
```
t1  = &x + 10
*t1 = 2
```



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

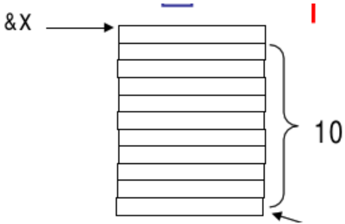
Address calculations in P-code: `x[10] = 2`

- tailor-made commands for address calculation



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

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



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)[6]

```
a + (i+1) * sizeof(int)
```

- `a` here *directly* stands for the base address

[6] 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[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 3AIC format. Apart from the "readable" notation, it's just two op-codes, say =[] and []=.

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
```

- "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  elem_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`

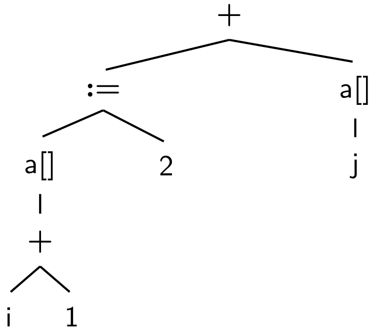
```
lda  a
lod  i
ldc  1
adi
ixa  elem_size(a)
lda  a
lod  j
ldc  2
mpi
ixa  elem_size(a)
ind  0
ldc  3
adi
sto
```

Extending 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]



P-code generation: arrays (1): op

```
void genCode (SyntaxTree t, int isAddr)  {
  char codestr[CODESIZE];
  /* CODESIZE = max length of 1 line of P-code */
  if (t != NULL) {
    switch (t->kind) {
    case OpKind:
      { switch (t->op) {
        case Plus:
          if (isAddress) emitCode("Error");   // new check
          else {                              // unchanged
            genCode(t->lchild,FALSE);
            genCode(t->rchild,FALSE);
            emitCode("adi");                  // addition
          }
          break;
        case Assign:
          genCode(t->lchild,TRUE);            //``l-value''
          genCode(t->rchild,FALSE);           //``r-value''
          emitCode("stn");
```

Listing 9: Code generation 3AIC (arrays)

P-code generation: arrays (2): "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;
```

Listing 10: Code generation 3AIC (arrays: "subs")

P-code generation: arrays (3): constants and identifiers

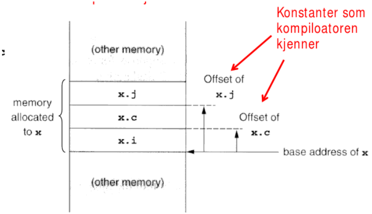
```
      case ConstKind:
        if (isAddr) emitCode("Error");
        else {
          sprintf(codestr,"%s %s","lds",t->strval);
          emitCode(codestr);
        }
        break;
      case IdKind:
        if (isAddr)
          sprintf(codestr,"%s %s", "lda",t->strval);
        else
          sprintf(codestr,"%s %s", "lod",t->strval);
        emitCode(codestr);
        break;
      default:
        emitCode("Error");
        break;
      }
    }
}
```

Listing 11: Code generation 3AIC (arrays: constants and ids)

Access to records

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

Rec x;
```

Listing 12: Sample struct type declaration



- 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*

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

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

Treenode *p;
```

Listing 13: Some sample record type declaration (binary trees)

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

3AIC

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

Structs and pointers in P-code

- 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
```

Section

Control statements and logical expressions

Chapter 9 “Intermediate code generation”
Course “Compiler Construction”
Martin Steffen
Spring 2021

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*

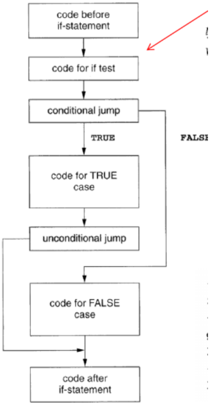
- generation of labels (cf. also temporaries)

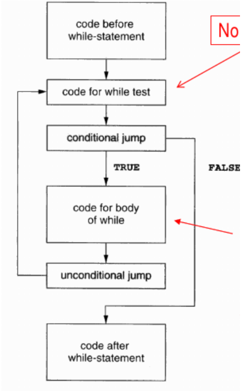
Loops and conditionals: linear code arrangement

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

- 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





INF5110 – Compiler Construction

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
// goto false branch
<code for S1>
// fall through to true branch
goto L2
// hop over false branch
label L1
<code for S2>
label L2
```

P-code for conditional

```
<code to evaluate E>
fjp L1
// got false branch
<code for S1>
// fall through to true branch
ujp L2
// hop over false branch
lab L1
<code for S2>
lab L2
```

3 new op-codes:

- **ujp**: unconditional jump ("goto")
- **fjp**: jump on false

Jumps and labels: while

while $(E)\ S$

3AIC for while

```
label L1
// label the loop header
<code to evaluate E to t1>
if_false t1 goto L2
// jump to after the loop
<code for S>
goto L1
// jump back
label L2
// label the loop exit
```

P-code for while

```
lab L1
// label the loop header
<code to evaluate E>
fjp L2
// jump to after the loop
<code for S>
ujp L1
// jump back
lab L2
// label the loop exit
```

Boolean expressions

- two alternatives for treatment
  1. as *ordinary* expressions
  2. via *short-circuiting*
- ultimate representation in HW:
  - no built-in booleans (HW is generally untyped)
  - but "arithmetic" 0, 1 work equivalently & fast
  - bitwise ops which corresponds to logical $\wedge$ and $\vee$ etc
- comparison on "booleans": $0 < 1$?
- boolean values vs. jump conditions

Short circuiting boolean expressions

```
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)

$$a \text{ and } b \quad \triangleq \quad \text{if } a \text{ then } b \text{ else false}$$
$$a \text{ or } b \quad \triangleq \quad \text{if } a \text{ then true else } b$$

```
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

Grammar for loops and conditionals

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

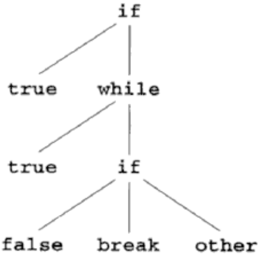
- note: simplistic expressions, only *true* and *false*

```
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;
```

Listing 14: C data structures for AST (control flow structures)

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
```

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 (old)

```
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;
  }
}
```

Rek. kall

Kode for S

Label ved slutt av denne while-setn

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 for conditional (short circuit)

```
case IfKind:
  lab_t = genLabel();
  lab_f = genLabel();
  genBoolCode(t->child[0],lab_t,lab_f);     // boolean condition
  sprintf(codestr,"%s %s", "lab", lab_t); // if-branch
  emitCode(codestr);
  genCode(t->child[1],label);
  lab_x = genLabel();
  if (t->child[2]!=NULL) {                  // does there exists an els
    sprintf(codestr,"%s %s", "ujp", lab_x);
    emitCode(codestr);
  }
  sprintf(codestr,"%s %s", "lab", lab_f); // else-branch
  emitCode(codestr);
  if (t->child[2]!=NULL) {                  // does there exists an els
    genCode(t->child[2],label);
    sprintf(codestr,"%s %s", "lab", lab_x);// post-statement label (
    emitCode(codestr);
  }
  break;
```

Listing 15: Alternative code generation for p-code (conditionals)

Code generation for bools (short circuit)

```
void genBoolCode (string lab_t, lab_f) =
  ...
  switch ... {
    case "||" : {
      String lab_x = genLabel();
      left.genBoolCode(lab_t, lab_x);
      sprintf(codestr,"%s %s", "lab", lab_x);
      emitCode(codestr);
      right.genBoolCode(lab_t, lab_f);
    }

    case "&&" : {
      String lab_x = genLabel();
      left.genBoolCode(lab_x, lab_f);
      sprintf(codestr,"%s %s", "lab", lab_x);
      emitCode(codestr);
      right.genBoolCode(lab_t, lab_f);
    }

    case "not" : {  // here just a left tree
      left.genBoolCode(lab_f, lab_t);
    }

    case "<" : {              // example for a binary relation
      String t_1, t_2, t_3; //
      t_1 = left.genIntCode();
```

Or case

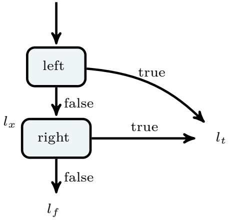


Figure: Short circuiting booleans, case "or"

References I

Chapter 10

*

[plain,t]

Course “Compiler Construction”