Intermediate Code Generation

a.y. 2022-2023

Compiler front-end

Phases of the front-end:

- Lexical analysis
- Parsing
- Semantic analysis (static checking: operators are applied to compatible operands; return-statement or break-statement enclosed within a while-statement; ecc.)
- Intermediate code generation

Semantic analysis and intermediate code generation can be specified by syntax-directed translations

Many translation schemes can be implemented during parsing, all of them can be implemented by creating an abstract syntax tree and then walking it

Intermediate representation

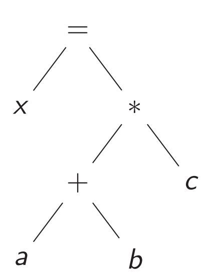
Many possibilities:

- Graph-like structures: abstract syntax trees or directed acyclic graphs
- Three-address code: $x = y \ op \ z$
- Another language (C is favoured by its flexibility)

Three-address code

Linearised representation of syntax trees

Example:



Provide temporary names for interior nodes

$$
\begin{aligned}
t_1 &= a + b \\
t_2 &= t_1 * c \\
x &= t_2
\end{aligned}
$$

Desirable representation for target-code generation and optimization

Three-address code

A variety of instructions allowed:

- $a1 = a2 op\ a3$
- $a1 = op\ a2$
- $a1 = a2$
- $a1 = a2[a3]$
- $a1[a2] = a3$
- ...
- *goto L*
- *if a goto L*
- ...

Intermediate code

For instance

```
if ( x < 100 || x > 200 && x != y ) x=0;
```

Can be translated to

```
     IF x < 100 GOTO L2
     GOTO L3
L3:  IF x > 200 GOTO L4
     GOTO L1
L4:  IF x != y GOTO L2
     GOTO L1
L2:  x = 0
L1:
```

Intermediate code

For instance

```
if ( x < 100 || x > 200 && x != y ) x=0;
```

Can be translated to

	IF x $<$ 100 GOTO L2
	GOTO L3
L3:	IF x $>$ 200 GOTO L4
	GOTO L1
L4:	IF x != y GOTO L2
	GOTO L1
L2:	x $=$ 0
L1:	

Or to

	IF x $<$ 100 GOTO L2
	IFFALSE x $>$ 200 GOTO L1
	IFFALSE x != y GOTO L1
L2:	x $=$ 0
L1:	

Syntax-directed translation of expressions

$$S \to id = E$$

$$E \to E_1 + E_2$$

$$E \to -E_1$$

$$E \to (E_1)$$

$$E \to id$$

Syntax-directed translation of expressions

Goal: emit three-adress code

Attributes and auxiliary functions:

- $E.addr$ denotes the temporary holding the value of E
- $S.code$, $E.code$ denote the code emitted for S and E
- $gen(str)$ emits the string str
- $newtemp()$ generates a new temporary
- Symbol $\triangleright$ stands for the concatenation of intermediate-code fragments

Syntax-directed translation of expressions

$S \rightarrow id = E$ $\quad$ $S.code = E.code \triangleright$

$\qquad gen(table.get(id)\ '='\ E.addr)$

$E \rightarrow E_1 + E_2$ $\quad$ $E.addr = newtemp()$

$E.code = E_1.code \triangleright E_2.code \triangleright$

$\qquad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$

Syntax-directed translation of expressions

$$E \rightarrow -E_1 \qquad E.addr = newtemp()$$
$$E.code = E_1.code \triangleright$$
$$gen(E.addr\ '=' \ '-' \ E_1.addr)$$

$$E \rightarrow (E_1) \qquad E.addr = E_1.addr$$
$$E.code = E_1.code$$

$$E \rightarrow id \qquad E.addr = table.get(id)$$
$$E.code = \ ' \ '$$

Syntax-directed translation of expressions
TRAINING

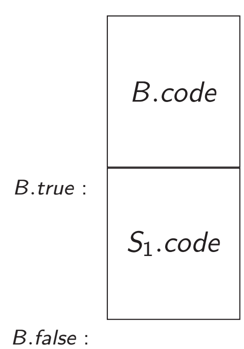
- Guess the ‘right’ tree for $a = b + -c$
- What’s the code then?

Control flow statements

$$P \rightarrow S$$
$$S \rightarrow if\ (B) S_1$$
$$S \rightarrow while\ (B)\ S_1$$
$$\ldots$$
$$B \rightarrow true$$
$$B \rightarrow false$$
$$B \rightarrow not\ B$$
$$B \rightarrow E_1\ rel\ E_2$$
$$B \rightarrow B_1 \parallel B_2$$
$$B \rightarrow B_1\ \&\&\ B_2$$
$$\ldots$$

Control flow statements

EXAMPLE: IF-THEN STATEMENT



Control flow statements

Attributes for statements:

- $S.next$, inherited, labels the beginning of the code that must be executed after S is finished
- $S.code$, synthesised, is the sequence of intermediate-code steps that implements the statement S and ends with a jump to $S.next$
- $B.true$, inherited, labels the beginning of the code that must be executed if B is true
- $B.false$, inherited, labels the beginning of the code that must be executed if B is false
- $B.code$, inherited, is the sequence of intermediate-code steps that implement the boolean condition B and jumps to $B.true$ if B is true, and to $B.false$ otherwise

Control flow statements

Notice:

Booleans have two distinct roles:

- Alter the flow of control (boolean conditions)
- Compute logical values (boolean expressions, as right-hand sides of assignments)

What we consider here is their translation in the context of control flow statements

Seen as expressions, they are translated analogously to what is done for arithmetic expressions, with logical operators rather than arithmetic operators

The grammar can have distinct nonterminals to distinguish these two roles

Control flow statements

$$P \to S \qquad S.next = newlabel();$$
$$P.code = S.code \triangleright label(S.next)$$

Where:

$newlabel()$ generates a new label at each invocation

$label(L)$ attaches label L to the next three-address instruction to be generated

Control flow statements

$$P \to S \qquad S.next = newlabel();$$
$$P.code = S.code \triangleright label(S.next)$$

$S.next$ is the label of the instruction executed when S is finished

If after performing some statements in S we need to jump at another location then we must set up an explicit GOTO instruction

Control flow statements

$S \rightarrow if\ (B)\ S_1$

$$B.true = newlabel()$$
$$B.false = S.next$$
$$S_1.next = S.next$$
$$S.code = B.code \triangleright label(B.true) \triangleright S_1.code$$

Control flow statements

$S \rightarrow while\ (B)\ S_1$

$$B.true = newlabel()$$
$$B.false = S.next$$
$$S_1.next = newlabel()$$
$$S.code = label(S_1.next) \triangleright B.code \triangleright$$
$$label(B.true) \triangleright S_1.code \triangleright$$
$$gen(GOTO\ S_1.next)$$

Control flow statements
BOOLEAN CONDITIONS

$B \to true$ $\quad B.code = gen(GOTO\ B.true)$

$B \to false$ $\quad B.code = gen(GOTO\ B.false)$

$B \to not\ B_1$ $\quad B_1.true = B.false$
$B_1.false = B.true$
$B.code = B_1.code$

$B \to E_1\ rel\ E_2$ $\quad B.code = E_1.code \triangleright E_2.code \triangleright$
$gen(IF\ E_1.addr\ relop\ E_2.addr\ GOTO\ B.true) \triangleright$
$gen(GOTO\ B.false)$

Control flow statements
BOOLEAN CONDITIONS

$B \to B_1 \parallel B_2$ $\quad B_1.true = B.true$

$B_1.false = newlabel()$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code \triangleright$

$label(B1.false) \triangleright B_2.code$

Control flow statements

BOOLEAN CONDITIONS

$$B \rightarrow B_1 \ \&\& \ B_2 \qquad \begin{aligned} & B_1.true = newlabel() \\ & B_1.false = B.false \\ & B_2.true = B.true \\ & B_2.false = B.false \\ & B.code = B_1.code \triangleright \\ & \qquad\qquad label(B1.true) \triangleright B_2.code \end{aligned}$$

Control flow statements

BOOLEAN CONDITIONS

The above translation generates codes like the one on the left

```
     IF x < 100 GOTO L2
     GOTO L3
L3:  IF x > 200 GOTO L4
     GOTO L1
L4:  IF x != y GOTO L2
     GOTO L1
L2:  x = 0
L1:
```

```
     IF x < 100 GOTO L2
     IFFALSE x > 200 GOTO L1
     IFFALSE x != y GOTO L1
L2:  x = 0
L1:
```

We can organize the translation to get the code on the right instead, and so avoid redundant gotos

Control flow statements

AVOIDING REDUNDANT GOTOS

Under the hypothesis that if the boolean condition is true then the next instruction to execute is always the next in sequence, we can reformulate the SDD for

$P \to S$

$S \to if\ (B) S_1 \mid while\ (B)\ S_1$

...

$B \to true \mid false \mid not\ B \mid E_1\ rel\ E_2 \mid B_1 \parallel B_2 \mid B_1 \ \&\&\ B_2$

...

Using a special label "fall" which does not generate gotos (just fall down to the next instruction in sequence).

Control flow statements

AVOIDING REDUNDANT GOTOS

$P \to S$	$S.next = newlabel();$ $P.code = S.code \triangleright label(S.next)$
$S \to if\ (B)\ S_1$	$B.true = fall$ $B.false = S.next$ $S_1.next = S.next$ $S.code = B.code \triangleright S_1.code$

Control flow statements
AVOIDING REDUNDANT GOTOS

$S \to while\ (B)\ S_1$	$B.true = fall$ $B.false = S.next$ $S_1.next = newlabel()$ $S.code =$ $\quad label(S_1.next) \triangleright B.code \triangleright$ $\quad S_1.code \triangleright gen(GOTO\ S_1.next)$
$B \to true$	$if\ \ B.true \neq fall$ $then\ gen(GOTO\ \ B.true)$
$B \to false$	$gen(GOTO\ \ B.false)$

Control flow statements
AVOIDING REDUNDANT GOTOS

$B \to not\ \ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \to E_1\ rel\ E_2$	$B.code = E_1.code \triangleright E_2.code \triangleright st$

Where, for $test = E_1.addr\ relop\ E_2.addr$

$$
\begin{aligned}
st \quad = \quad & if\ \ B.true \neq fall\ \ and\ \ B.false \neq fall \\
& then\ gen(IF\ \ test\ \ GOTO\ \ B.true) \triangleright gen(GOTO\ \ B.false) \\
& elseif\ \ B.true \neq fall \\
& \quad then\ gen(IF\ \ test\ \ GOTO\ \ B.true) \\
& \quad elseif\ \ B.false \neq fall \\
& \quad\quad then\ gen(IFFALSE\ \ test\ \ GOTO\ \ B.false) \\
& \quad\quad else\ skip
\end{aligned}
$$

Control flow statements
AVOIDING REDUNDANT GOTOS

$$
\begin{array}{ll}
B \to B_1 \parallel B_2 & B_1.true = \mathit{if}\ \ B.true \neq \mathit{fall} \\
& \qquad\qquad \mathit{then}\ \ B.true \\
& \qquad\qquad \mathit{else}\ \ newlabel() \\
& B_1.false = \mathit{fall} \\
& B_2.true = B.true \\
& B_2.false = B.false \\
& B.code = \\
& \quad \mathit{if}\ \ B.true \neq \mathit{fall} \\
& \quad \mathit{then}\ B_1.code \triangleright B_2.code \\
& \quad \mathit{else}\ \ B_1.code \triangleright B_2.code \triangleright label(B_1.true)
\end{array}
$$

Control flow statements
AVOIDING REDUNDANT GOTOS

$$
\begin{array}{ll}
B \to B_1\ \&\&\ B_2 & B_1.true = \mathit{fall} \\
& B_1.false = B.false \\
& B_2.true = B.true \\
& B_2.false = B.false \\
& B.code = B_1.code \triangleright B_2.code
\end{array}
$$

Control flow statements

AVOIDING THE SECOND PASS: BACKPATCHING

$$
\begin{array}{ll}
S \rightarrow \textit{if}\ (B)\ S_1 & B.\textit{true} = \textit{newlabel}() \\
 & B.\textit{false} = S.\textit{next} \\
 & S_1.\textit{next} = S.\textit{next} \\
 & S.\textit{code} = B.\textit{code} \triangleright \textit{label}(B.\textit{true}) \triangleright S_1.\textit{code}
\end{array}
$$

When the code for B is generated, $S.\textit{next}$ is not known yet

Problem: match jump instructions and target

Two passes are needed for the evaluation

Control flow statements

BACKPATCHING FOR BOOLEAN EXPRESSIONS

Strategy:

Generate incomplete jumps ($GOTO\ \_$)

Use synthesized attributes ($B.\textit{truelist}$, $B.\textit{falselist}$) for boolean expressions that keep track of the list of incomplete jump instructions where labels are still missing

Insert the missing label when the target is known:
$\textit{backpatch}(\textit{list\_of\_incomplete\_jumps}, \textit{address\_to\_jump\_to})$

For example:

$$
\begin{array}{ll}
S \rightarrow \textit{if}\ (B)\ M\ S_1 & \textit{backpatch}(B.\textit{truelist}, M.\textit{instr}); \\
 & S.\textit{nextlist} = \textit{merge}(B.\textit{falselist}, S_1.\textit{nextlist});
\end{array}
$$

Control flow statements
BACKPATCHING FOR BOOLEAN EXPRESSIONS

Why M in $S \to if\ (B)\ M\ S_1$?

To take the index of the next instruction

$$M \to \epsilon \qquad M.instr = nextinstr;$$

Control flow statements
BACKPATCHING FOR BOOLEAN EXPRESSIONS

Assume instructions are generated into an array and target labels are indexes of the array

- $B.truelist$: list of jump instructions where we have to insert the label to which the control goes if B is true
- $B.falselist$: list of jump instructions where we have to insert the label to which the control goes if B is false
- $nextinstruction$: holds the index of the following instruction
- $makelist(i)$: creates a list containing index i, returns a pointer to the list
- $merge(p_1, p_2)$: concatenates the lists pointed by p_1 and p_2, returns a pointer to the list
- $backpatch(p, i)$: inserts label i as target of each of the incomplete instructions in the list pointed by p

Control flow statements

BACKPATCHING FOR BOOLEAN EXPRESSIONS

$M \rightarrow \epsilon$	$M.instr = nextinstr;$
$B \rightarrow true$	$B.truelist = makelist(nextinstr);$ $gen(GOTO\ \_)$
$B \rightarrow false$	$B.falselist = makelist(nextinstr);$ $gen(GOTO\ \_)$

Control flow statements

BACKPATCHING FOR BOOLEAN EXPRESSIONS

$B \rightarrow not\ B_1$	$B.truelist = B_1.falselist;$ $B.falselist = B_1.truelist;$
$B \rightarrow E_1\ rel\ E_2$	$B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $gen(IF\ E_1.addr\ relop\ E_2.addr\ GOTO\ \_);$ $gen(GOTO\ \_);$

Control flow statements

BACKPATCHING FOR BOOLEAN EXPRESSIONS

$B \rightarrow B_1 \parallel M\ B_2$	$backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist;$
$B \rightarrow B_1\ \&\&\ M\ B_2$	$backpatch(B_1.truelist, M.instr);$ $B.truelist = B_2.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist);$

Control flow statements

BACKPATCHING FOR BOOLEAN EXPRESSIONS

$S \rightarrow if\ (B)\ M\ S_1$	$backpatch(B.truelist, M.instr);$ $S.nextlist = merge(B.falselist, S_1.nextlist);$
$S \rightarrow while\ M_1\ (B)\ M_2\ S_1$	$backpatch(S_1.nextlist, M_1.instr);$ $backpatch(B.truelist, M_2.instr);$ $S.nextlist = B.falselist;$ $gen(GOTO\ M_1.instr);$

Addressing Array Elements

- The main issue in generating code for arrays is the computation of the address of their elements
- Elements are stored in consecutive locations to ease accessing them

Addressing Array Elements

Two main strategies to store arrays:

- *Row-major order.* Change the rigthmost index: for a 2×4 matrix, first store $A[0][0]$, then $A[0][1]$, then $A[0][2]$, , then $A[0][3]$, then $A[1][0]$, then $A[1][1]$, then $A[1][2]$, then $A[1][3]$
  - for $i = 0$ to 1
    for $j = 0$ to 3
        A[i][j] $=$1
  - The assignment steps through memory in sequential order
- *Column-major order.* Change the leftmost index: for the same 2×3 matrix, first store $A[0][0]$, then $A[1][0]$, then $A[0][1]$, then $A[1][1]$, then $A[0][2]$, then $A[1][2]$, then $A[0][3]$, then $A[1][3]$

Addressing Array Elements

ROW-MAJOR ORDER

- The main issue in generating code for arrays is the computation of the address of their elements
- Elements are stored in consecutive locations to ease accessing them
- If elements are $0, 1, \ldots, n$, *base* is the relative address of the storage allocated for the one-dimensional array, and w is the width of each element, then the *ith* element starts at

$$base + i * w$$

- For two-dimensional arrays, if w_1 is the width of an entire row, and w_2 the width of an element in a row, then $A[i_1][i_2]$ starts at then the *ith* element starts at

$$base + i_1 * w_1 + i_2 * w_2$$

Syntax-directed translation of arrays

$$S \to id = E \mid L = E$$

$$E \to E + E \mid id \mid L$$

$$L \to id[E] \mid L[E]$$

To generate, e.g.,

- $a[b] = c$
- $a[b][c] = d[e]$
- $a = b[c]$

Syntax-directed translation of arrays

Attributes:

- $L.addr$ denotes a temporary used in computing the due offset
- $L.array$ is a reference to the symbol table entry for the array name; the entry contains the *base* address of the array ($L.array\_base$) and the width of its elements ($L.array\_ewidth$)
- $L.width$ the width of the subarray generated by L

Syntax-directed translation of arrays

$S \to L = E$

Generate an instruction to assign the value denoted by E to the location denoted by the reference L

$L.addr$ holds the offset from the base, which is in $L.array\_base$

Hence the location for L is $L.array\_base[L.addr]$

$$S \to L = E$$

$$gen(L.array\_base \ '[' \ L.addr \ ']' \ ' =' \ \ E.addr)$$

Syntax-directed translation of arrays

$$E \rightarrow L \qquad E.add = newtemp()$$
$$gen(E.addr\ '=' \ L.array\_base\ '[' L.addr ']')$$

Syntax-directed translation of arrays

$$L \rightarrow id[E] \qquad L.array = table.get(id)$$
$$L.width = L.array\_ewidth$$
$$L.addr = newtemp()$$
$$gen(L.addr\ '=' \ E.addr\ '*' \ L.width)$$

Syntax-directed translation of arrays

$L \to L_1[E]$

$$L.array = L_1.array$$

$$L.width = L_1.width$$

$$L.addr = newtemp()$$

$$t = newtemp()$$

$$gen(t\ '='\ E.addr\ '*'\ L.width)$$

$$gen(L.addr\ '='\ L_1.addr\ '+'\ t)$$

Syntax-directed translation of arrays

TRAINING

Assuming that b is a 2×3 array of integers, that a, i, j are integers, and that the width of integers is 4, translate

$$a + b[i][j]$$