CS 346: Intermediate Code Generation

Resource: Textbook

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "*Compilers: Principles, Techniques, and Tools*", Addison-Wesley, 1986.

Intermediate Code Generation

- *Front end of compiler*: translates a source program into an intermediate representation
- Details of the back end are left to the back end
- Benefits include:
  - Retargeting
  - Machine-independent code optimization

Intermediate Code Generation

- Intermediate codes
  - machine independent
  - close to machine instructions
- Given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator

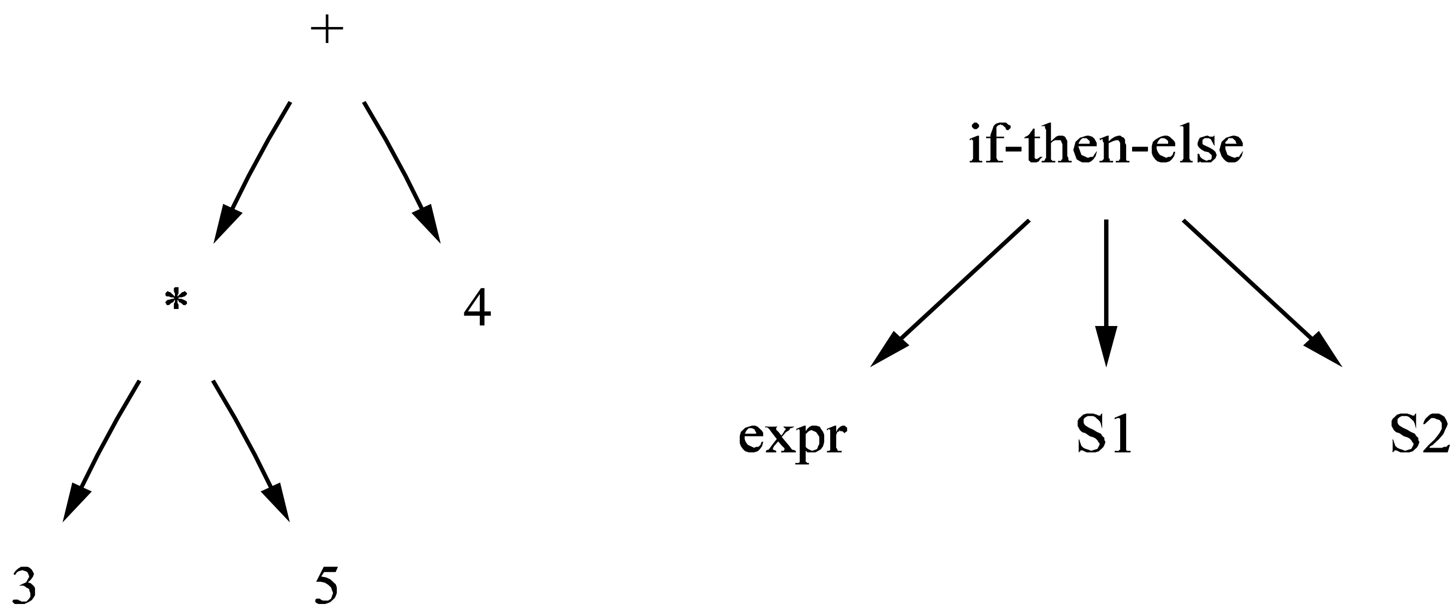
- Many different intermediate representations exist, and the designer of the compiler decides this intermediate language
  - syntax trees
  - postfix notation
  - three-address code (*Quadraples*)
    - quadraples are close to machine instructions, but they are not actual machine instructions
  - some programming languages have well defined intermediate languages
    - *java* – java virtual machine
    - *prolog* – warren abstract machine

Syntax Trees

- (Abstract) Syntax Trees
  - Condensed form of parse tree
  - Useful for representing language constructs
  - Operators and keywords appear as internal nodes

- Syntax-directed translation can be based on *syntax trees* as well as *parse trees*

Syntax Tree Examples



Implementing Syntax Trees

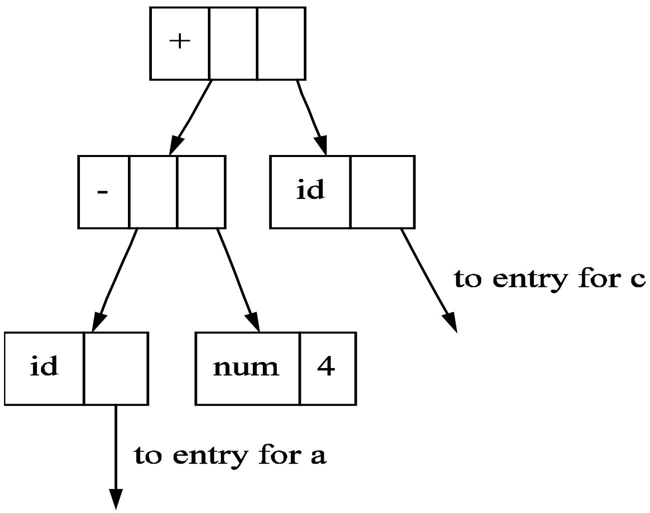
- Each node can be represented by a record with several fields
- Example: node representing an operator used in an expression
  - One field indicates the operator and others point to records for nodes representing operands
  - Operator is referred to as the “label” of the node

- If being used for translation, records can have additional fields for attributes

Syntax Trees for Expressions

- Functions will create nodes for the syntax tree
  - `mknode (op, left, right)` – creates an operator node with label `op` and pointers `left` and `right` which point to operand nodes
  - `mkleaf(id, entry)` – creates an identifier node with label `id` and a pointer to the appropriate symbol table entry
  - `mkleaf(num, val)` – creates a number node with label `num` and value `val`
- Each function returns pointer to created node

Example: a - 4 + c



p_1 := mkleaf(**id**, p_a);
P_2 := mkleaf(num, 4);
p_3 := mknode('-', p_1, p_2);
p_4 := mkleaf(**id**, p_c);
p_5 := mknode('+', p_3, p_4);

Constructing Trees for Expressions

Production	Semantic Rules
E → E_1 + T	E.np := mknode('+', E_1.np, T.np)
E → E_1 – T	E.np := mknode('-', E_1.np, T.np)
E → T	E.np := T.np
T → (E)	T.np := E.np
T → id	T.np := mkleaf(id, id.entry)
T → num	T.np := mkleaf(num, value)

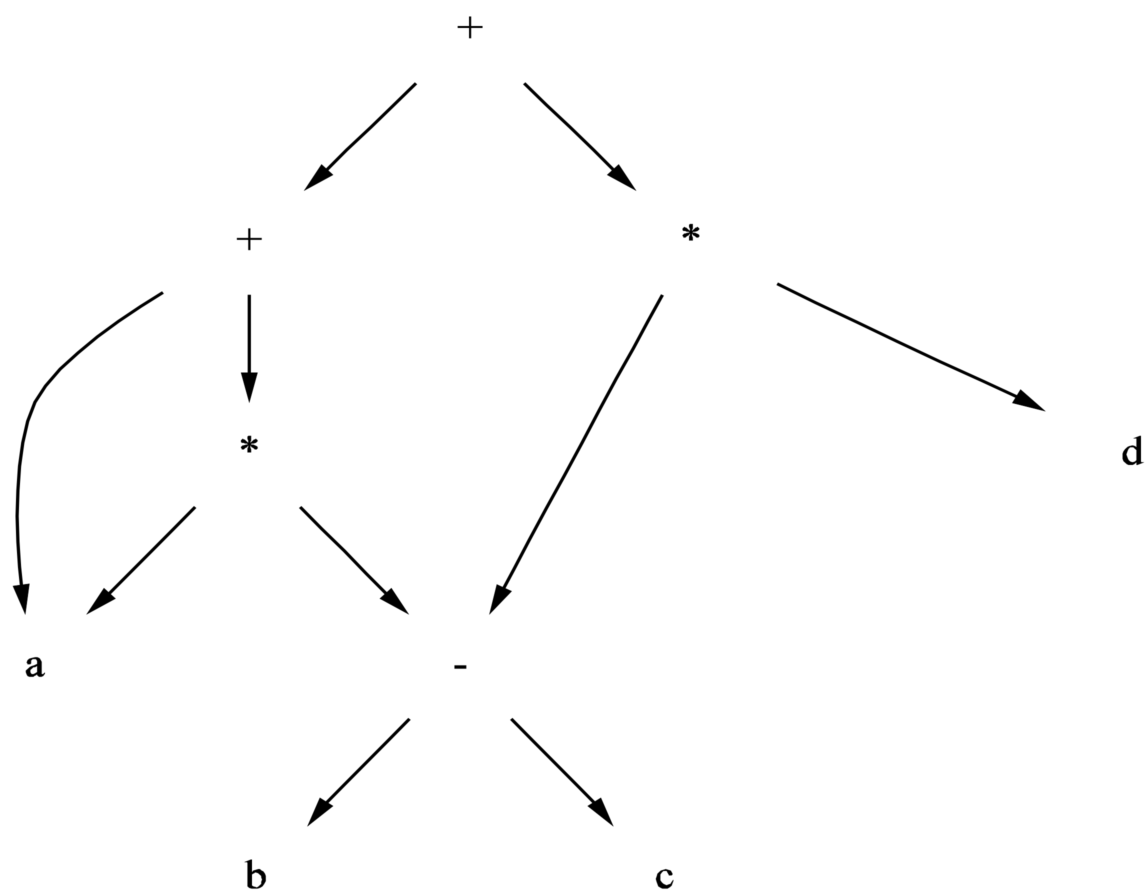
Directed Acyclic Graphs

- Called a *DAG* for short
- Convenient for representing expressions
- Similar to syntax trees:
  - Every sub-expression will be represented by a node
  - Interior nodes represent operators
  - Children represent operands

- DAG: Unlike syntax trees, nodes may have more than one parent (while representing common sub-expressions)
- Syntax tree: tree for common sub-expressions replicated as many times as they appear in the original expression

Thus, DAG generates more efficient codes for evaluating expressions !

Example: `a + a * (b – c) + (b – c) * d`



Steps for constructing DAG

p_1=Leaf (id, entry-a)

p_2=Leaf (id, entry-a)=p_1

p_3=Leaf (id, entry-b)

p_4=Leaf (id, entry-c)

p_5=Node ('-', p_3 , p_4)

p_6=Node ('*', p_1 , p_5)

p_7=Node ('+', p_1 , p_6)

p_8=Leaf (id, entry-b)= p_3

p_9=Leaf (id, entry-c)= p_4

p_{10}=Node ('-', p_3 , p_4)= p_5

p_{11}=Leaf (id, entry-d)

p_{12}=Node ('*', p_5 , p_{11})

p_{13}=Node ('*', p_7 , p_{12})

Three-Address Code (Quadruples)

- A quadraple:

  `x := y op z`

  where x, y and z are names, constants or compiler-generated temporaries; **op** is any operator

- Alternative notation (much better notation because it looks like a machine code instruction):

  `op  y,z,x`

  apply operator `op` to `y` and `z`, and store the result in `x`

- We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result)

Three-Address Statements

Binary Operator: op y,z,result or result := y op z

where op is a binary arithmetic or logical operator. This binary operator is applied to y and z, and the result of the operation is stored in result.

Ex: add a,b,c

gt a,b,c

sub a,b,c

Unary Operator: op y,,result or result := op y

where op is an unary arithmetic or logical operator. This unary operator is applied to y, and the result of the operation is stored in result

Ex: uminus a,,c

not a,,c

Three-Address Statements (cont.)

Move Operator: mov y,,result or result := y

where the content of y is copied into result.

Ex: mov a,,c

Unconditional Jumps: jmp ,,L or goto L

jump to the three-address code with the label L, and the execution continues from that statement.

Ex: jmp ,,L1 // jump to L1

jmp ,,7 // jump to the statement 7

Three-Address Statements (cont.)

Conditional Jumps: `jmp` ***`relop`*** `y,z,L` or `if y` ***`relop`*** `z goto L`

Jump to the three-address code with the label `L` if the result of `y relop z` is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement

Ex:

```
jmpgt   y,z,L1    // jump to L1 if y>z
jmpgte  y,z,L1    // jump to L1 if y>=z
jmpe    y,z,L1    // jump to L1 if y==z
jmpne   y,z,L1    // jump to L1 if y!=z
```

Relational operator can also be a unary operator.

```
jmpnz   y,,L1     // jump to L1 if y is not zero
jmpz    y,,L1     // jump to L1 if y is zero
jmpt    y,,L1     // jump to L1 if y is true
jmpf    y,,L1     // jump to L1 if y is false
```

Three-Address Statements (cont.)

Procedure Parameters: `param x,,` or `param x`

Procedure Calls: `call p,n,` or `call p,n`

where `x` is an actual parameter, we invoke the procedure `p` with `n` parameters

Ex:

```
param x1,,
param x2,,
                 ➔ p(x1,...,xn)
param xn,,
call  p,n,
```

```
f(x+1,y) ➔     add   x,1,t1
               param t1,,
               param y,,
               call  f,2,
```

Three-Address Statements (cont.)

Indexed Assignments:

```
mov y[i],,x    or   x := y[i]
mov x,,y[i]    or   y[i] := x
```

Address and Pointer Assignments:

```
movaddr y,,x   or   x := &y
movcont y,,x   or   x := *y
```

Generating Three-Address Code

- Temporary names are made up for the interior nodes of a syntax tree
- The synthesized attribute S.code represents the code for the assignment S
- The nonterminal E has attributes:
  - E.place: name that will hold the value of E
  - E.code: sequence of three-address statements evaluating E
- The function newtemp returns a sequence of distinct names
- The function newlabel returns a sequence of distinct labels

Syntax-Directed Translation into Three-Address Code

Production	Semantic Rules
S → id := E	S.code := E.code || gen(id.place ':=' E.place)
E → E_1 + E_2	E.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)
E → E_1 * E_2	E.place := newtemp; E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)

Syntax-Directed Translation into Three-Address Code

Production	Semantic Rules
E → -E_1	E.place := newtemp; E.code := E_1.code || gen(E.place ':=' 'uminus' E_1.place)
E → (E_1)	E.place := E_1.place; E.code := E_1.code
E → id	E.place := id.place; E.code := ''

Syntax-Directed Translation into Three-Address Code (alternative representation)

$S \rightarrow \textbf{id} := E$	S.code = E.code || gen('mov' E.place ',,' id.place)
$E \rightarrow E_1 + E_2$	E.place = newtemp();
	E.code = E_1.code || E_2.code || gen('add' E_1.place ',' E_2.place ',' E.place)
$E \rightarrow E_1 * E_2$	E.place = newtemp();
	E.code = E_1.code || E_2.code || gen('mult' E_1.place ',' E_2.place ',' E.place)
$E \rightarrow - E_1$	E.place = newtemp();
	E.code = E_1.code || gen('uminus' E_1.place ',,' E.place)
$E \rightarrow (E_1)$	E.place = E_1.place;
	E.code = E_1.code
$E \rightarrow \textbf{id}$	E.place = **id**.place;
	E.code = null

Triples

- Triples refer to a temporary value by the position of the statement that computes it
  - Statements can be represented by a record with only three fields: *op*, *arg1*, and *arg2*
  - Avoids the need to enter temporary names into the symbol table

- Contents of *arg1* and *arg2*:
  - Pointer into symbol table (for programmer defined names)
  - Pointer into triple structure (for temporaries)

Triples Example

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	t2	(3)
(5)	assign	a	(4)

Quadruples vs. Triples

- Quadruples need temporary names into the symbol table

- Quadruples:
  - Instructions that use a temporary *t* does not require any change if the instructions computing *t* moved
  - Useful for optimizing compilers as instructions often need to be moved around
- Triples: Result of an operation is referred by its position, so moving an instruction may require to change all the references to that results
  - *Solution: Indirect triples* (maintain a list of pointers to triples, rather than listing of all triples)
  - Java is analogous to indirect triple representation

Indirect Triple Representation

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

Three Address Codes - Example

```
x:=1;
y:=x+10;
while (x<y) {
    x:=x+1;
    if (x%2==1) then y:=y+1;
    else y:=y-2;
}
```

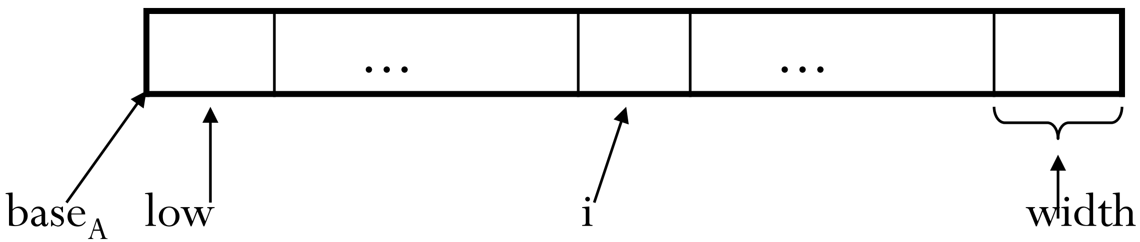
➔

```
01: mov   1,,x
02: add   x,10,t1
03: mov   t1,,y
04: lt    x,y,t2
05: jmpf  t2,,17
06: add   x,1,t3
07: mov   t3,,x
08: mod   x,2,t4
09: eq    t4,1,t5
10: jmpf  t5,,14
11: add   y,1,t6
12: mov   t6,,y
13: jmp   ,,16
14: sub   y,2,t7
15: mov   t7,,y
16: jmp   ,,4
17:
```

Arrays

- Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive locations

A one-dimensional array **A**:



$base_A$ is the address of the first location of the array A,

width is the width of each array element

low is the index of the first array element

location of A[i] ➔ $base_A+(i-low)*width$

Arrays (cont.)

$\texttt{base}_A\texttt{+(i-low)*width}$

can be re-written as $\underbrace{\texttt{i*width}}_{\text{should be computed at run-time}} + \underbrace{\texttt{(base}_A\texttt{-low*width)}}_{\text{can be computed at compile-time}}$

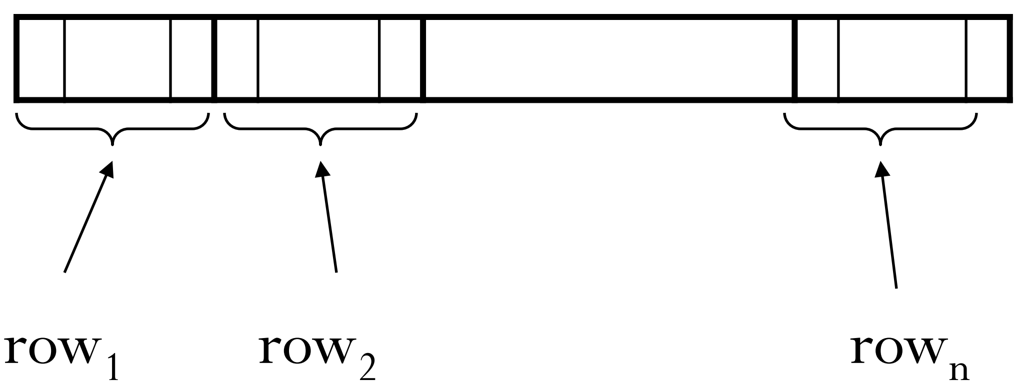
- So, the location of A[i] can be computed at the run-time by evaluating the formula i*width+c where c is ($base_A$-low*width) which is evaluated at compile-time

- Intermediate code generator should produce the code to evaluate this formula i*width+c (one multiplication and one addition operation)

Two-Dimensional Arrays

- A two-dimensional array can be stored in
  - either **row-major** (*row-by-row*) or
  - **column-major** (*column-by-column*)
- Most of the programming languages use **row-major** method

- Row-major representation of a two-dimensional array:



Two-Dimensional Arrays (cont.)

- The location of $A[i_1,i_2]$ is

$$base_A + ((i_1-low_1)*n_2+i_2-low_2)*width$$

$base_A$ is the location of the array A
low_1 is the index of the first row
low_2 is the index of the first column
n_2 is the number of elements in each row
width is the width of each array element

- Again, this formula can be re-written as

$$\underbrace{((i_1*n_2)+i_2)*width}_{\text{should be computed at run-time}} + \underbrace{(base_A-((low_1*n_1)+low_2)*width)}_{\text{can be computed at compile-time}}$$

Multi-Dimensional Arrays

- In general, the location of $A[i_1,i_2,\ldots,i_k]$ is

  $((\ \ldots\ ((i_1*n_2)+i_2)\ \ldots)*n_k+i_k)*width\ +\ (base_A-((\ldots((low_1*n_1)+low_2)\ldots)*n_k+low_k)*width)$

- So, the intermediate code generator should produce the codes to evaluate the following formula (to find the location of $A[i_1,i_2,\ldots,i_k]$) :

  $((\ \ldots\ ((i_1*n_2)+i_2)\ \ldots)*n_k+i_k)*width\ +\ c$

- To evaluate the $((\ \ldots\ ((i_1*n_2)+i_2)\ \ldots)*n_k+i_k$ portion of this formula, we can use the recurrence equation:

  $e_1 = i_1$

  $e_m = e_{m-1} * n_m + i_m$

Translation Scheme for Arrays –Ex.1

- A one-dimensional `double` array `A` : 5..100

  ➔ n_1=95 width=8 (double) low_1=5

- Intermediate codes corresponding to `x := A[y]`

```
mov    c,,t1          // where c=base_A-(5)*8
mult   y,8,t2
mov    t1[t2],,t3
mov    t3,,x
```

Translation Scheme for Arrays – Ex. 2

- A two-dimensional `int` array `A` : 1..10x1..20

  ➔ $n_1=10$ $n_2=20$ width=4 (integers) $low_1=1$ $low_2=1$

- Intermediate codes corresponding to `x := A[y,z]`

```
mult  y,20,t1
add   t1,z,t1
mov    c,,t2
mult  t1,4,t3
mov   t2[t3],,t4
mov   t4,,x
```

// where $c=base_A-(1*10+1)*4$

$((i_1*n_2)+i_2)*width + (base_A-((low_1*n_1)+low_2)*width)$

Translation Scheme for Arrays – Ex. 3

- A three-dimensional int array A : 0..9x0..19x0..29

  ➔ $n_1=10$ $n_2=20$ $n_3=30$ width=4 (integers) $low_1=0$ $low_2=0$ $low_3=0$

- Intermediate codes corresponding to x := A[w,y,z]

```
mult  w,20,t1
add   t1,y,t1
mult  t1,30,t2
add   t2,z,t2
mov   c,,t3            // where c=base_A-((0*10+0)*30+0)*4
mult  t2,4,t4
mov   t3[t4],,t5
mov   t5,,x
```

$((\ldots((i_1*n_2)+i_2)\ldots)*n_k+i_k)*width$ + $(base_A-((\ldots((low_1*n_1)+low_2)\ldots)*n_k+low_k)*width)$

Boolean Expressions

- Boolean expressions compute logical values
- Often used with flow-of-control statements
- Methods of translating boolean expression:
  - Numerical methods:
    - True is represented as 1 and false is represented as 0
    - Nonzero values are considered true and zero values are considered false
  - Flow-of-control methods:
    - Represent the value of a boolean by the position reached in a program
    - Often not necessary to evaluate the entire expression

Numerical Methods

- Expressions evaluated left to right using 1 to denote true and 0 to donate false
  - Example: `a or b and not c`

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

  - Another example: `a < b`

```
100:  if a < b goto 103
101:  t : = 0
102:  goto 104
103:  t : = 1
104:  …
```

Numerical Methods

Production	Semantic Rules

E $\rightarrow$ E$_1$ or E$_2$	`E.place := newtemp;`
	`emit(E.place ':=' E`$_1$`.place 'or'`
	`E`$_2$`.place)`

E $\rightarrow$ E$_1$ and E$_2$	`E.place := newtemp;`
	`emit(E.place ':=' E`$_1$`.place 'and'`
	`E`$_2$`.place)`

E $\rightarrow$ not E$_1$	`E.place := newtemp;` `emit(E.place ':=' 'not' E`$_1$`.place)`
E $\rightarrow$ (E$_1$)	`E.place := E1.place;`

Numerical Methods

Production	Semantic Rules

`E → id₁ relop id₂`	`E.place := newtemp;`
	`emit('if' id₁.place relop.op`
	`    id₂.place 'goto' nextstat+3);`
	`emit(E.place ':=' '0');`
	`emit('goto' nextstat+2);`
	`emit(E.place ':=' '1');`

`E → true`	`E.place := newtemp;` `emit(E.place ':=' '1')`
`E → false`	`E.place := newtemp;` `emit(E.place ':=' '0')`

Example: `a<b or c<d and e<f`

```
100:  if a < b goto 103
101:  t1 := 0
102:  goto 104
103:  t1 := 1
104:  if c < d goto 107
105:  t2 := 0
106:  goto 108
107:  t2 := 1
108:  if e < f goto 111
109:  t3 := 0
110:  goto 112
111: t3 := 1
112:  t4 := t2 and t3
113:  t5 := t1 or t4
```

Flow-of-Control

- Function *newlabel* will return a new symbolic label each time it is called

- Two attributes for each boolean expression:
  - `E.true:` label to which control flows if `E` is true
  - `E.false:` label to which control flows if `E` is false

- Attribute `S.next` of a statement `S`:
  - Inherited attribute whose value is the label attached to the first instruction to be executed after the code for `S`
  - Used to avoid jumps to jumps

Flow-of-Control

Production	Semantic Rules
S → if E then S_1	E.true := newlabel; E.false := S.next; S1.next := S.next; S.code := E.code || label(E.true) || S_1.code

Flow-of-Control

Production	Semantic Rules

S → if E then S_1 else S_2	E.true := newlabel;
	E.false := newlabel;
	S1.next := S.next;
	S2.next := S.next;
	S.code := E.code ||
	  label(E.true) ||
	  S_1.code ||
	  gen('goto' S.next) ||
	  label(E.false) ||
	  S_2.code

Flow-of-Control

Production	Semantic Rules
S → while E do S_1	S.begin := newlabel; E.true := newlabel; E.false := S.next; S1.next := S.begin; S.code := label(S.begin) || E.code || label(E.true) || S_1.code || gen('goto' S.begin)
$S \rightarrow S_1 S_2$	S1.next=newlabel() S2.next=S.next S.code=S1.code || label(S1.next) || S2.code

Flow-of-Control (Ex-1)

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

```
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:
```

Code is not optimal

Redundant goto: goto L3 (as the label of the very next instruction)

Two goto L1 instructions can also be eliminated

Flow-of-Control

More optimized codes

```
If x<100 goto L2
ifFalse x> 200 goto L1
ifFalse x!=y goto L1
L2: x=0
L1:
```

Handling different boolean expressions

- **Roles of boolean expressions**
  - Alter the flow of control
    - Example (x < y)
  - Can be evaluated for its value
    - Examples: x=true; x=a<b

- **Handling techniques**
  - Use two passes:
    1. construct a complete syntax tree for the i/p
    2. traverse the tree in depth-first manner
    3. compute the translations specified by the semantic rules
  - Use one pass for statements, but two passes for expressions:

    e.g: *while (E) S1: Translate E before S1 is executed*

    For E's translation: build syntax tree and then depth-first traversal

Two-Pass Implementation

- Problems in generating codes for boolean expressions and flow of control statements
  - Matching a jump instruction with the target of jump (or)
  - May not know the labels to which control must flow at the time a jump is generated
  - Affect boolean expressions and flow control statements

- Example: *If (B) S*
  - Contains a jump to the instruction following code for S (*B: false*)
  - One pass implementation: B must be translated before S is examined
  - *What is the target of goto that jumps over code S?*
    - *Pass labels as inherited attributes to where relevant jump instructions generated*
    - *Separate pass needed to bind labels*

Backpatching (a complementary approach)

- Lists of jumps passed as synthesized attributes
- Leave targets of jumps temporarily unspecified
- Add each such statement to a list of goto statements whose labels will be filled in later
- This filling in of labels is called *backpatching*

- Can be used to generate the code in a single pass
- Translations generated are basically the same as before except for labels
- Instructions generated into an instruction array
- Labels are indices into the array

One-pass code generation using backpatching

- For a nonterminal B:
  - B.truelist: list of jump (conditional and unconditional) instructions into which label should be inserted when B is true
  - B.falselist: list of instructions that eventually get the label to which control goes when B is false
  - Generate code for B
  - Don't specify the jumps to the true and false exits
  - Label filed is kept unfilled
- Incomplete jumps placed on lists pointed to by *B.truelist* and *B.falselist*

- *S.nextlist*: denotes a lists of jumps to the instruction that follow the code for statement S

Lists of Labels

- Lists of labels (statements with jumps to labels) are maintained
- New functions used to manipulate the lists
  - `makelist(i)`
    - Creates a new list containing only `i`, an index into the array of instructions
    - Returns a pointer to the new list
  - `merge(p1, p2)`
    - Concatenates two lists (pointed out by p1 and p2) of labels
    - Returns a pointer to the new list
  - `backpatch(p, i)` – inserts `i` as target label for each statement on the list pointed to by `p`

Backpatching of Boolean Expressions

$$
\begin{aligned}
E \rightarrow\ & E_1 \texttt{ or } M\ E_2 \\
\mid\ & E_1 \texttt{ and } M\ E_2 \\
\mid\ & \texttt{not } E_1 \\
\mid\ & (E_1) \\
\mid\ & \texttt{id}_1 \texttt{ relop } \texttt{id}_2 \\
\mid\ & \texttt{true} \\
\mid\ & \texttt{false} \\
\\
M \rightarrow\ & \varepsilon
\end{aligned}
$$

Backpatching of Boolean Expressions (controlled by bottom-up parsing)

- Translation scheme suitable for producing codes during bottom-up parsing
  - The new marker (M) has an associated semantic action
    - Picks up, at appropriate times, the index of the next instruction to be generated

- Synthesized attributes of nonterminal E :
  - *E.truelist*: a list of statements that jump when expression is true
  - *E.falselist*: a list of statements that jump when expression is false

Example: $E \rightarrow E_1$ and $M\ E_2$

- E_1 is false:
  - E is also false
  - Statements on E_1.falselist become part of E.falselist
- E_1 is true:
  - Still need to test E_2
  - Target for statements on E_1.truelist must be the beginning of code generated for E_2
  - Target is obtained using the marker M

Syntax-Directed Translation

Production	Semantic Rules
$E \rightarrow E_1$ or $M\ E_2$	backpatch(E_1.falselist, M.instr); E.truelist := merge(E_1.truelist, E_2.truelist); E.falselist := E_2.falsetlist
$E \rightarrow E_1$ and $M\ E_2$	backpatch(E_1.truelist, M.instr); E.truelist := E_2.truelist; E.falselist := merge(E_1.falselist, E_2.falselist)
$E \rightarrow$ not E_1	E.truelist := E_1.falselist; E.falselist := E_1.truelist
$E \rightarrow (E_1)$	E.truelist := E_1.truelist; E.falselist := E_1.falselist

Syntax-Directed Translation

Production	Semantic Rules
$E \rightarrow id_1$ relop id_2	E.truelist := makelist(nextinstr); E.falselist := makelist(nextinstr+1); emit('if' id_1.place relop.op id_2.place 'goto _'); emit('goto _')
$E \rightarrow$ true	E.truelist := makelist(nextinstr); emit('goto _')
$E \rightarrow$ false	E.falselist := makelist(nextinstr); emit('goto _')
$M \rightarrow \varepsilon$	M.instr := nextinstr

An Example

`a<b or c<d and e<f`

- First, `a<b` will be reduced, generating:

```
100:  if a < b goto _
101:  goto _
```

- Next, the marker `M` in $E \rightarrow E_1$ or M E_2 will be reduced, and `M.instr` will be set to `102`
- Next, `c<d` will be reduced, generating:

```
102:  if c < d goto _
103:  goto _
```

An Example

- Next, the marker `M` in $E \rightarrow E_1$ `and` $M\ E_2$ will be reduced, and `M.instr` will be set to `104`
- Next, `e<f` will be reduced, generating:

```
104:  if e < f goto _
105:  goto _
```

- Next, we reduce by $E \rightarrow E_1$ `and` $M\ E_2$
  - Semantic action calls `backpatch({102}, 104)`
  - E_1`.truelist` contains only `102`
  - Line `102` now reads: `if c <d goto 104`

An Example

- Next, we reduce by $E \rightarrow E_1$ `or` M E_2
  - Semantic action calls `backpatch({101}, 102)`
  - E_1`.falselist` contains only `101`
  - Line `101` now reads: `goto 102`

- Statements generated so far:

```
100:  if a < b goto _
101:  goto 102
102:  if c < d goto 104
103:  goto _
104:  if e < f goto _
105:  goto _
```

- Rest goto instructions will have their addresses filled in later

Annotated Parse Tree