

Intermediate Code Generation

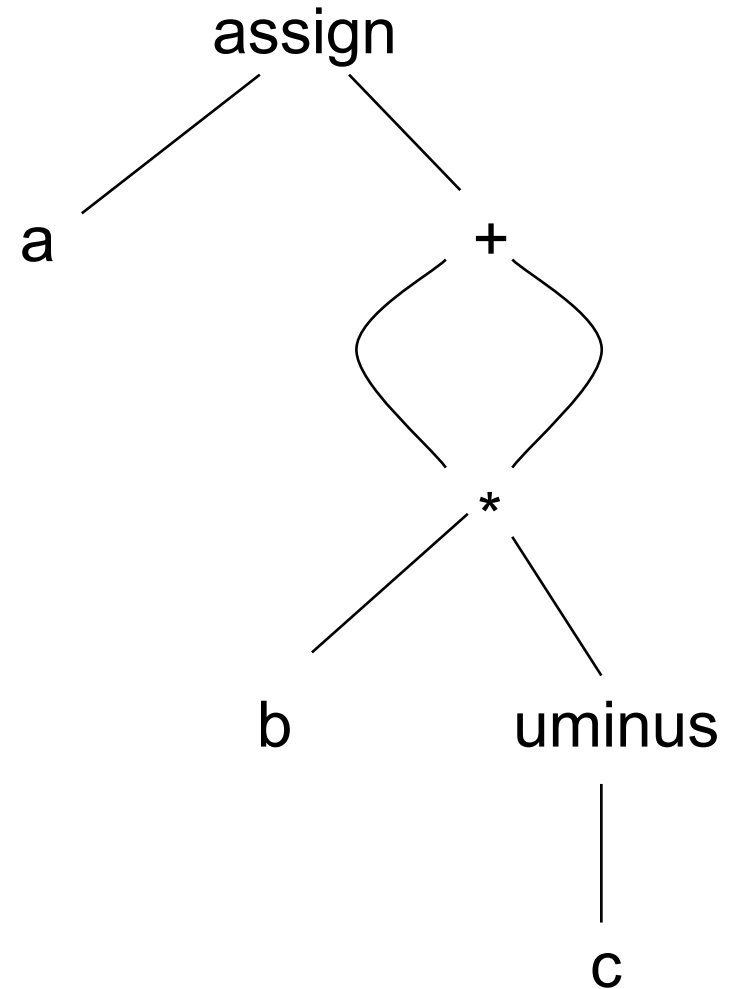
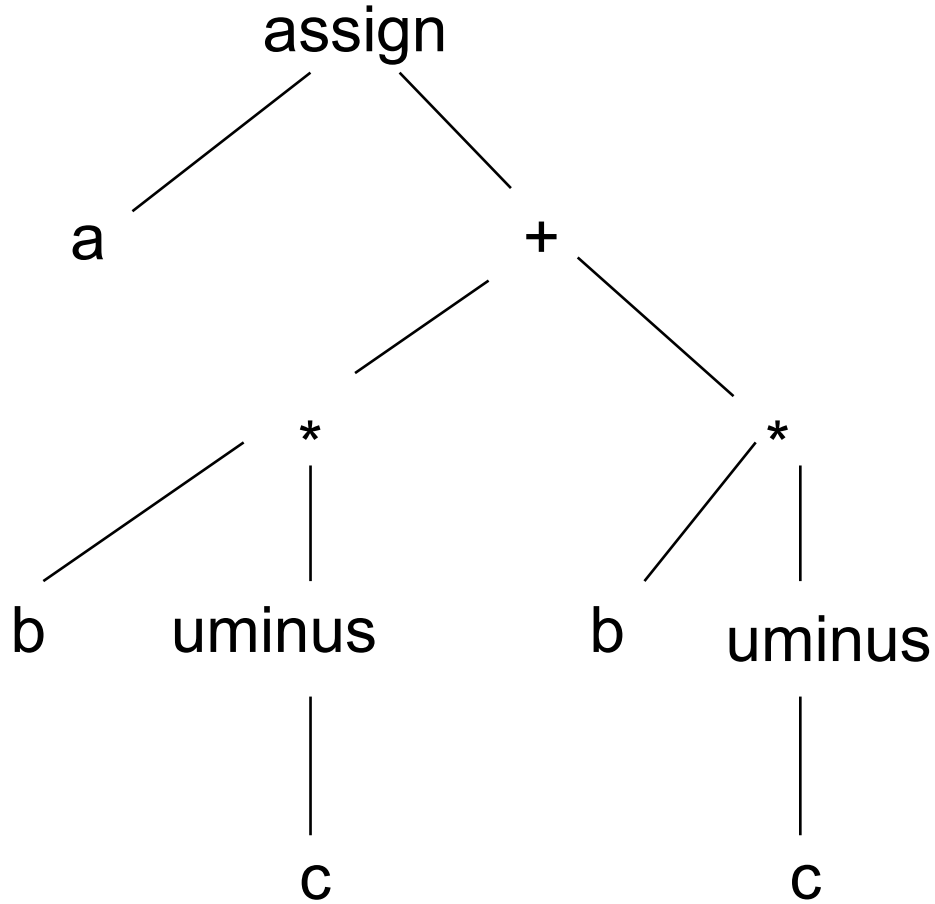
Intermediate Code Generation

- Translating source program into an “intermediate language”
 - Simple
 - CPU Independent,
 - ... yet, close in spirit to machine language
- Benefits
 - Retargeting is facilitated
 - Machine independent code optimization can be applied

Intermediate Code Generation

- Intermediate codes are machine independent codes, but they are close to machine instructions
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator
- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language
 - Syntax trees can be used as an intermediate language
 - Postfix notation can be used as an intermediate language
 - Three-address code (Quadruples) can be used as an intermediate language
 - we will use quadruples to discuss intermediate code generation
 - quadruples 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
 - In fact, there are byte-code emulators to execute instructions in these intermediate languages

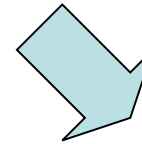
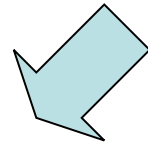
A Tree and A DAG as Intermediate Languages



- Pro: easy restructuring of code
and/or expressions for
intermediate code optimization
- Con: memory intensive

Postfix Notation

a := b * -c + b * -c



a b c uminus * b c uminus * + assign

Postfix notation represents operations on a stack

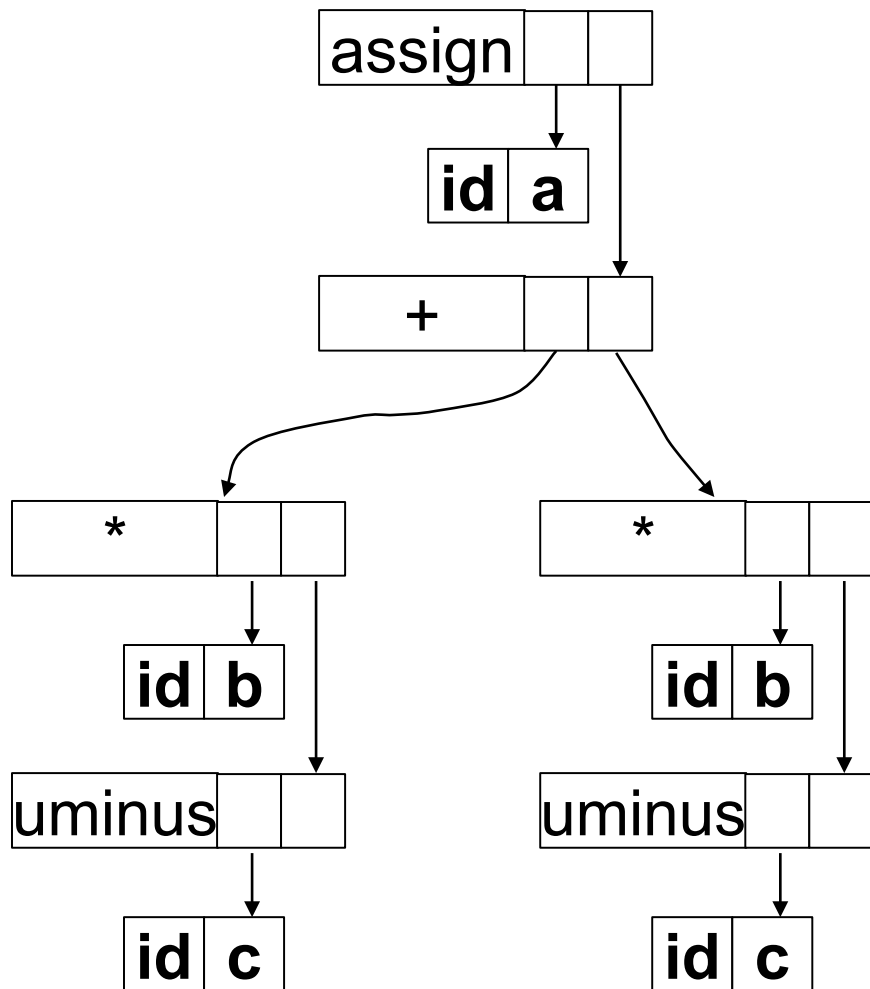
Pro: easy to generate

Con: stack operations are more difficult to optimize

Bytecode (for example)

```
iload 2      // push b
iload 3      // push c
ineg         // uminus
imul         // *
iload 2      // push b
iload 3      // push c
ineg         // uminus
imul         // *
iadd         // +
istore 1     // store a
```

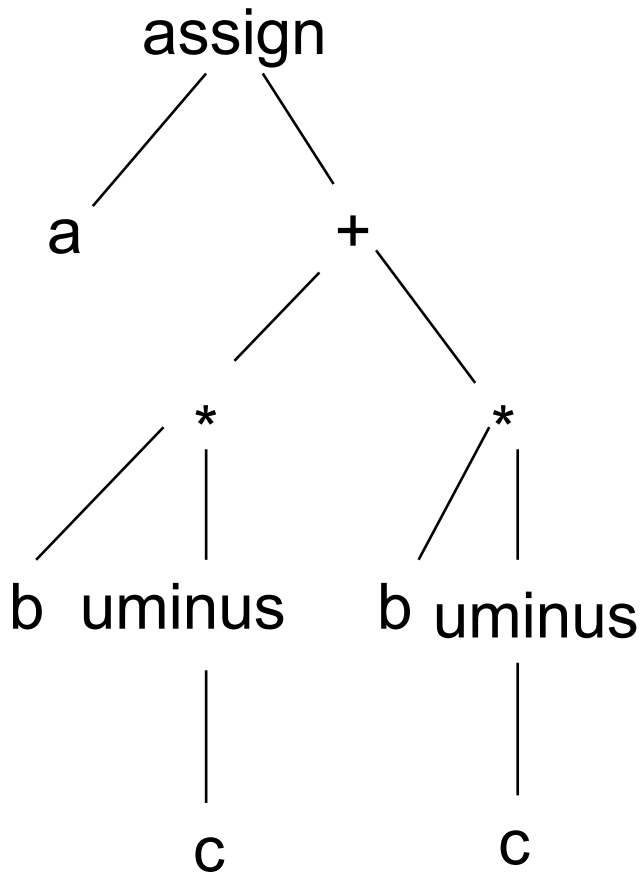
Two Representations of A Syntax Tree



0	id	b		
1	id	c		
2	uminus		1	
3	*		0	2
4	id	b		
5	id	c		
6	uminus		5	
7	*		4	6
8	+		3	7
9	id	a		
10	assign		9	8
11	...			

Three-Address Code

`x := y op z`

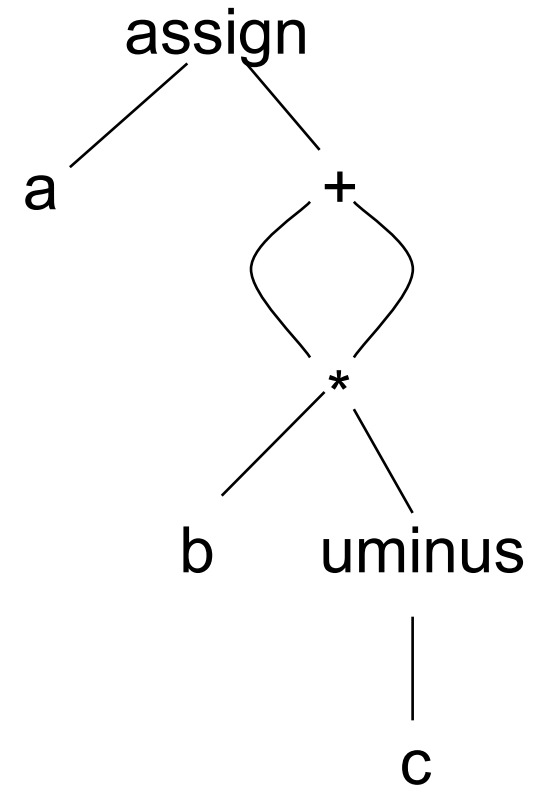


Tree

```
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Dag

```
t1 := -c
t2 := b * t1
t5 := t2 + t2
a := t5
```



Three-address code is a linearization of the tree/DAG

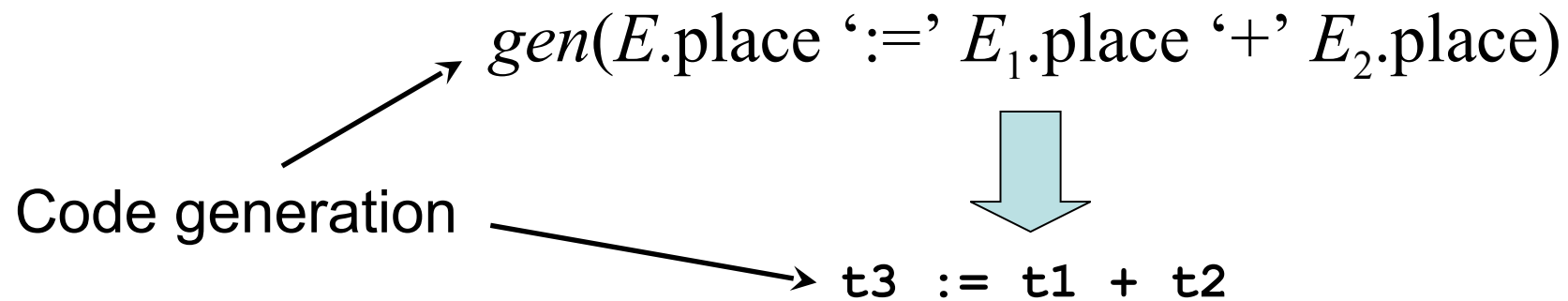
Types of Three-Address Code Instructions

- Binary operations:
 $x := y \text{ op } z$
- Unary operations:
 $x := \text{op } y$
- Copy instructions:
 $x := y$
- Conditional jumps:
 $\text{if } x \text{ relop } y \text{ goto } L$
- Procedure calls:
 $\text{param } x_1$
 $\text{param } x_2$
 \dots
 $\text{param } x_n$
 $\text{call } p, n$
- Index assignments:
 $x := y[i], x[i] := y$
- Address and pointer assignments:
 $x := \&y, x := *y, *x := y$

Syntax-Directed Translation Into 3-address Code

- First deal with assignments and simple expressions
- Use attributes
 - *E.place*: the name that will hold the value of *E*
 - Identifier will be assumed to already have the place attribute defined
 - *E.code*: hold the three address code statements that evaluate *E*
- Use function `newtemp` that returns a new temporary variable that we can use
- Use function `gen` to generate a single three address statement given the necessary information (variable names and operations)

The *gen* Function



Code Generation for Expressions

Production	Semantic Rules
$S \rightarrow \mathbf{id} := E$	$S.code := E.code \parallel \text{gen}(\mathbf{id}.place \text{ ':=' } E.place)$
$E \rightarrow E_1 + E_2$	$E.place := \text{newtemp};$ $E.code := E_1.code \parallel E_2.code \parallel \text{gen}(E.place \text{ ':=' } E_1.place \text{ '+' } E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := \text{newtemp};$ $E.code := E_1.code \parallel E_2.code \parallel \text{gen}(E.place \text{ ':=' } E_1.place \text{ '*' } E_2.place)$
$E \rightarrow -E_1$	$E.place := \text{newtemp};$ $E.code := E_1.code \parallel \text{gen}(E.place \text{ ':=' 'uminus' } E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow \mathbf{id}$	$E.place := \mathbf{id}.place$ $E.code := ""$

Code Generation for *while* Loops

Production	Semantic Rules
$S \rightarrow \text{while } E \text{ do } S_1$	<pre> S.begin:=newlabel; S.after:=newlabel; S.code:=gen(S.begin ':') E.code gen('if' E.place '=' '0' 'goto' S.after) S₁.code gen('goto' S.begin) gen(S.after ':') </pre>

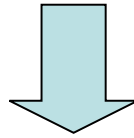
S.begin :

<i>E.code</i>
if <i>E.place</i> = 0 goto <i>S.after</i>
<i>S₁.code</i>
goto <i>S.begin</i>
<i>S.after :</i>

S.after :

Example

```
i := 2 * n + k  
while i do  
  i := i - k
```



```
t1 := 2  
t2 := t1 * n  
t3 := t2 + k  
i := t3  
L1: if i = 0 goto L2  
    t4 := i - k  
    i := t4  
    goto L1  
L2:
```

Declarations

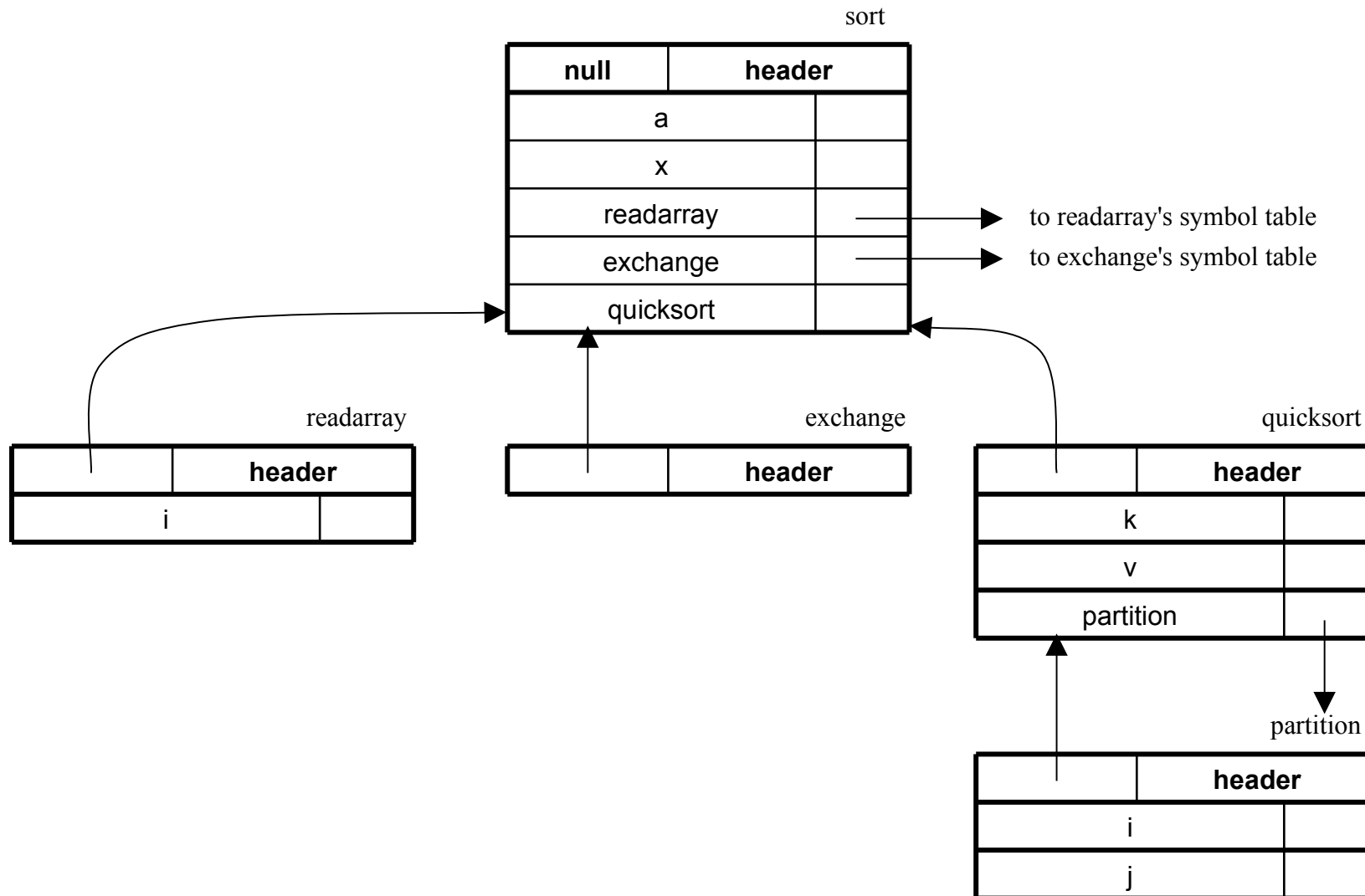
$P \rightarrow \{ \textit{offset} := 0 \} D$
 $D \rightarrow D ; D$
 $D \rightarrow \mathbf{id} : T \quad \{ \textit{addtype}(\mathbf{id.place}, T.type, \textit{offset});$
 $\quad \quad \quad \textit{offset} := \textit{offset} + T.width \}$
 $T \rightarrow \mathbf{char} \quad \{ T.type := \textit{char}; T.width := 1 \}$
 $T \rightarrow \mathbf{integer} \quad \{ T.type := \textit{integer}; T.width := 4 \}$
 $T \rightarrow \mathbf{real} \quad \{ T.type := \textit{real}; T.width := 8 \}$
 $T \rightarrow \wedge T_1 \quad \{ T.type := \textit{pointer}(T_1.type); T.width := 4 \}$
 $T \rightarrow \mathbf{array} [\mathbf{num}] \mathbf{of} T_1$
 $\quad \quad \quad \{ T.type := \textit{array}(1..\mathbf{num.val}, T_1.type);$
 $\quad \quad \quad T.width := \mathbf{num.val} * T_1.width \}$

name	type	offset
a	<i>char</i>	0
b	<i>pointer(real)</i>	1
c	<i>real</i>	5

Nested Procedure Declarations

- For each procedure we should create a symbol table.
 - *mktable(previous)* – create a new symbol table where *previous* is the parent symbol table of this new symbol table
 - *addtype(symtable,place,type,offset)* – update an entry for a variable in the given symbol table.
 - *enterproc(symtable,name,newsymbtable)* – create a new entry for the procedure in the symbol table of its parent.
 - *addwidth(symtable,width)* – puts the total width of all entries in the symbol table into the header of that table.
- We will have two stacks:
 - *tblptr* – to hold the pointers to the symbol tables
 - *offset* – to hold the current offsets in the symbol tables in *tblptr* stack.

Symbol Table for Nested Procedures



Nested Procedure Declarations

Production	Semantic Rules
$P \rightarrow MD$	addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset)
$M \rightarrow \epsilon$	t:=mhtable(nil); push(t, tblptr); push(0,offset)
$D \rightarrow D_1; D_2$	
$D \rightarrow \text{proc id}; ND_1; S$	t:=top(tblptr); addwidth(t,top(offset)); pop(tblptr); pop(offset); enterproc(top(tblptr), id .name, t)
$D \rightarrow \text{id}: T$	addtype(top(tblptr), id .place, T.type, top(offset)); top(offset) := top(offset)+T.width
$N \rightarrow \epsilon$	t:=mhtable(top(tblptr)); push(t,tblptr); push(0,offset)

Code Generation for Expressions

Production	Semantic Rules
$S \rightarrow \text{id} := E$	emit(id.place ':=' E.place)
$E \rightarrow E_1 + E_2$	E.place:=newtemp; emit(E.place ':=' E ₁ .place '+' E ₂ .place)
$E \rightarrow E_1 * E_2$	E.place:=newtemp; emit(E.place ':=' E ₁ .place '*' E ₂ .place)
$E \rightarrow - E_1$	E.place:=newtemp; emit(E.place ':=' 'uminus' E ₁ .place)
$E \rightarrow (E_1)$	E.place := E ₁ .place
$E \rightarrow \text{id}$	E.place:=id.place

Addressing of Array Elements

$$A[\text{low}..\text{high}] \quad A[i] \quad \text{base} + (i - \text{low}) * w$$

$$i * w + (\text{base} - \text{low} * w)$$

$$A[\text{low}_1..\text{high}_1, \text{low}_2..\text{high}_2] \quad A[i_1, i_2] \quad n_2 = \text{high}_2 - \text{low}_2 + 1$$

$$\text{base} + ((i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2) * w$$

$$((i_1 * n_2 + i_2) * w + (\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w))$$

Conversion of Types

```
E → E1 + E2 { E.place := newtemp;  
  if E1.type = integer and E2.type = integer then begin  
    emit (E.place := E1.place 'int+' E2.place);  
    E.type := integer;  
  end  
  else if E1.type = real and E2.type = real then begin  
    emit (E.place := E1.place 'real+' E2.place);  
    E.type := real;  
  end  
  else if E1.type = integer and E2.type = real then begin  
    u := newtemp;  
    emit (u := 'inttoreal' E1.place);  
    emit (E.place := u 'real+' E2.place);  
    E.type := real;  
  end  
  else if E1.type = real and E2.type = integer then begin  
    u := newtemp;  
    emit (u := 'inttoreal' E2.place);  
    emit (E.place := E1.place 'real+' u);  
    E.type := real;  
  end  
  else  
    E.type := type_error  
}
```

```
x := y + i * j
```

```
t1 := i int* j
```

```
t3 := inttoreal t1
```

```
t2 := y real+ t3
```

```
x := t2
```

Boolean Expressions

$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id relop id} \mid \text{true} \mid \text{false}$

a or b and not c

$t_1 := \text{not } c$

$t_2 := b \text{ and } t_1$

$t_3 := a \text{ or } t_2$

a < b \Rightarrow if a < b then 1 else 0

100: if a < b goto 103

101: t := 0

102: goto 104

103: t := 1

104:

Boolean Expressions

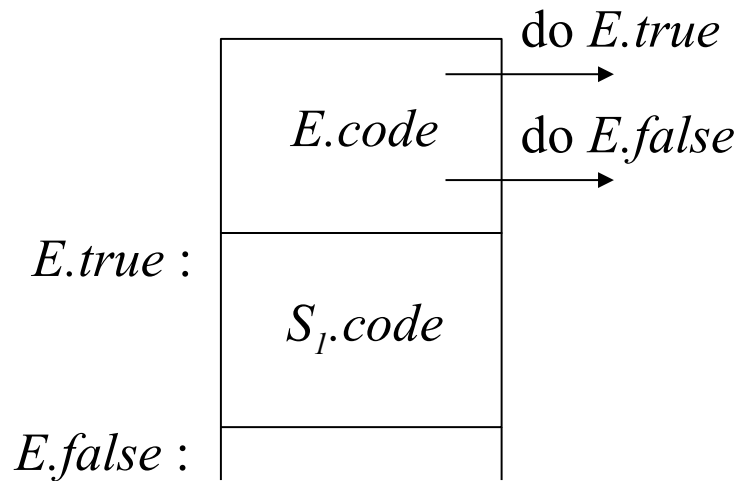
$E \rightarrow E_1 \text{ or } E_2$ $\{ E.place := newtemp; emit (E.place := E_1.place \text{ or } E_2.place) \}$
 $E \rightarrow E_1 \text{ and } E_2$ $\{ E.place := newtemp; emit (E.place := E_1.place \text{ and } E_2.place) \}$
 $E \rightarrow \text{not } E_1$ $\{ E.place := newtemp; emit (E.place := \text{not } E_1.place) \}$
 $E \rightarrow (E_1)$ $\{ E.place := E_1.place \}$
 $E \rightarrow id_1 \text{ relop } id_2$ $\{ E.place := newtemp;$
 $emit ('if' id_1.place \text{ relop.op } id_2.place \text{ goto } nextstat + 3);$
 $emit (E.place := '0')$
 $emit ('goto' nextstat + 2)$
 $emit (E.place := '1')$
 $\}$
 $E \rightarrow \text{true}$ $\{ E.place := newtemp; emit (E.place := '1') \}$
 $E \rightarrow \text{false}$ $\{ E.place := newtemp; emit (E.place := '0') \}$

$a < b \text{ or } c < d \text{ and } e < f$

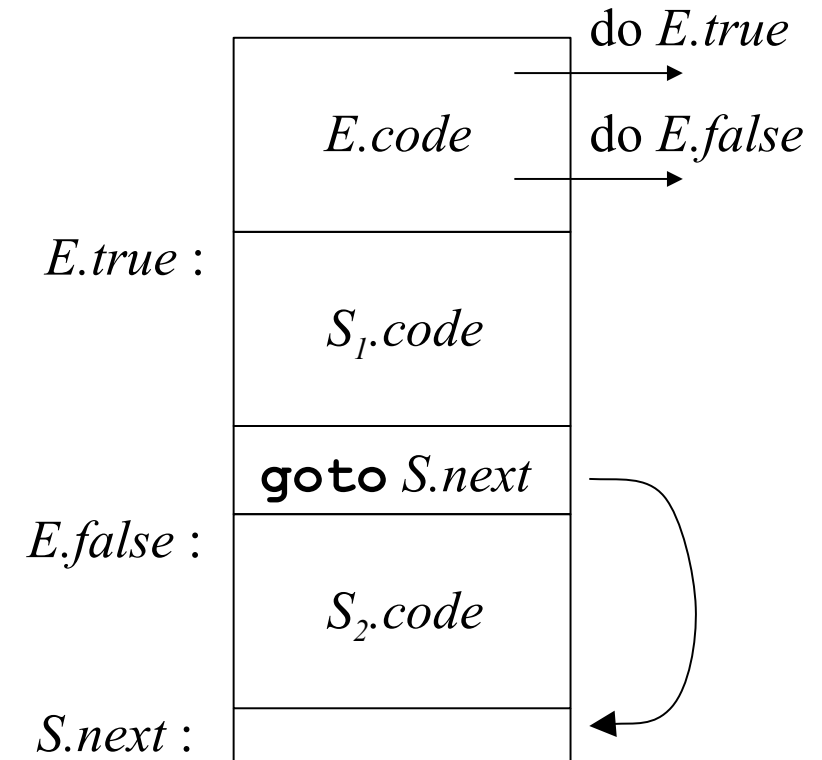
```
100: if a < b goto 103          107: t2 := 1
101: t1 := 0                  108: if e < f goto 111
102: goto 104                 109: t3 := 0
103: t1 := 1                  110: goto 112
104: if c < d goto 107        111: t3 := 1
105: t2 := 0                  112: t4 := t2 and t3
106: goto 108    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

Control Instructions

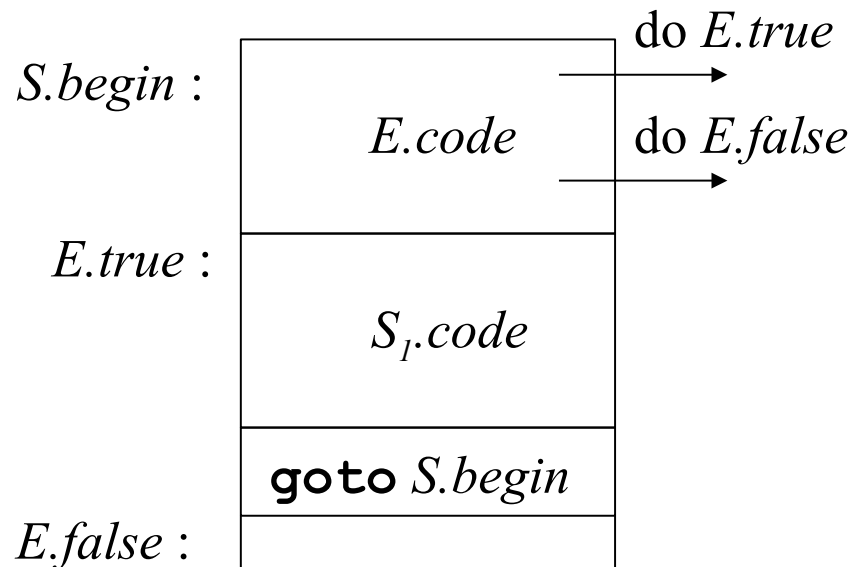
$S \rightarrow \text{if } E \text{ then } S_1$



$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$



$S \rightarrow \text{while } E \text{ do } S_1$



Dyntax-Directed Definition for Flow-of-Control Statements

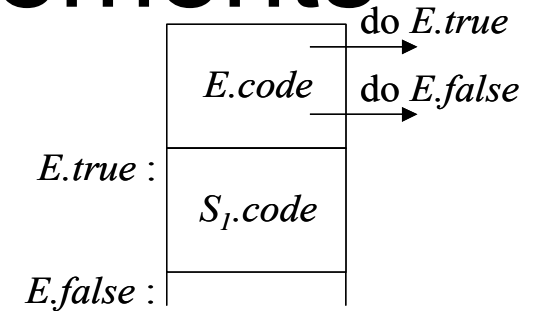
$S \rightarrow \text{if } E \text{ then } S_1$

 $E.true := \text{newlabel};$

 $E.false := S.next;$

 $S_1.next := S.next;$

 $S.code := E.code \parallel \text{gen}(E.true ':') \parallel S_1.code$



$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

 $E.true := \text{newlabel};$

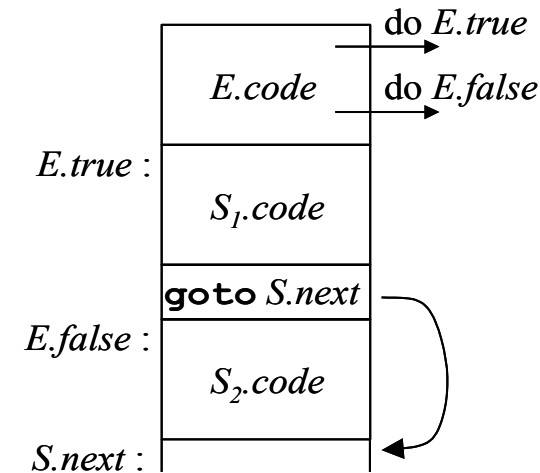
 $E.false := \text{newlabel};$

 $S_1.next := S.next;$

 $S_2.next := S.next;$

 $S.code := E.code \parallel \text{gen}(E.true ':') \parallel S_1.code \parallel$

 $\text{gen}(\text{goto } S.next) \parallel \text{gen}(E.false ':') \parallel S_2.code$



$S \rightarrow \text{while } E \text{ do } S_1$

 $S.begin := \text{newlabel};$

 $E.true := \text{newlabel};$

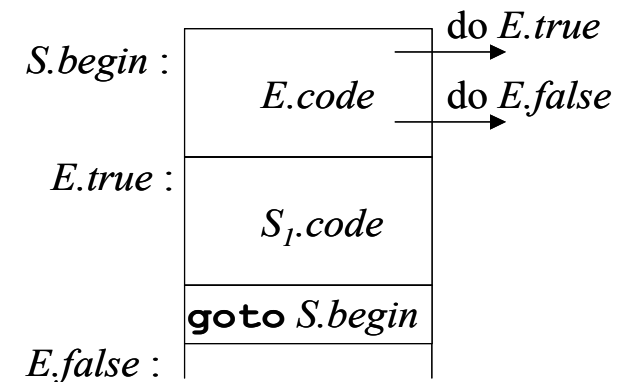
 $E.false := S.next$

 $S_1.next := S.begin;$

 $S.code := \text{gen}(S.begin ':') \parallel E.code \parallel$

 $\text{gen}(E.true ':') \parallel S_1.code \parallel$

 $\text{gen}(\text{goto } S.begin)$



Boolean Expression - "Short-Circuit" Translation

$E \rightarrow E_1 \text{ or } E_2$

```
E1.true := E.true;  
E1.false := newlabel;  
E2.true := E.true;  
E2.false := E.false;  
E.code := E1.code || gen(E1.false ':') || E2.code
```

Code for $E = a < b$:

```
if a < b goto E.true  
goto E.false
```

$E \rightarrow E_1 \text{ and } E_2$

```
E1.true := newlabel;  
E1.false := E.false;  
E2.true := E.true;  
E2.false := E.false;  
E.code := E1.code || gen(E1.true ':') || E2.code
```

$E \rightarrow \text{not } E_1$

```
E1.true := E.false;  
E1.false := E.true;  
E.code := E1.code;
```

$E \rightarrow (E_1)$

```
E1.true := E.true;  
E1.false := E.false;  
E.code := E1.code;
```

$E \rightarrow \text{id}_1 \text{ relop } \text{id}_2$

```
E.code := gen( 'if' id1.place relop.op id2.place 'goto' E.true ) ||  
gen( 'goto' E.false )
```

$E \rightarrow \text{true}$

```
E.code := gen( 'goto' E.true )
```

$E \rightarrow \text{false}$

```
E.code := gen( 'goto' E.false )
```

Boolean Expression - "Short-Circuit" Translation

a < b or c < d and e < f

```
    if a < b goto Ltrue
    goto L1
L1:  if c < d goto L2
    goto Lfalse
L2:  if e < f goto Ltrue
    goto Lfalse
```

```
while a < b do
    if c < d then
        x := y + z
    else
        x := y - z
```

```
L1:  if a < b goto L2
    goto Lnext
L2:  if c < d goto L3
    goto L4
L3:  t1 := y + z
    x := t1
    goto L1
L4:  t2 := y - z
    x := t2
    goto L1
Lnext:
```

Boolean Expressions - Mixed-Mode

$(a + b) < c$
 $(a < b) + (b < a)$

$E \rightarrow E + E \mid E \text{ and } E \mid E \text{ relop } E \mid \text{id}$

$E \rightarrow E_1 + E_2$ *E.type := arith;*
 if *E₁.type = arith* **and** *E₂.type = arith* **then begin**
 /* arithmetic addition */
 E.place := newtemp;
 E.code := E₁.code || E₂.code || gen(E.place := E₁.place '+' E₂.place)
 end
 else if *E₁.type = arith* **and** *E₂.type = bool* **then begin**
 E.place := newtemp;
 E₂.true := newlabel;
 E₂.false := newlabel;
 E.code := E₁.code || E₂.code || gen(E₂.true := E.place := E₁.place '+' 1) ||
 gen('goto' nextstat + 1) ||
 gen(E₂.false := E.place := E₁.place)
 else if ...

E₂.true : E.place := E₁.place + 1
 goto nextstat + 1
E₂.false : E.place := E₁.place

Translating Short-Circuit Expressions Using Backpatching

- For the examples of the previous lectures for implementing syntax-directed definitions, the easiest way is to use two passes. First syntax tree is constructed and is then traversed in depth-first order to compute the translations given in the definition
- The main problem in generating three address codes in a single pass for Boolean expressions and flow of control statements is that we may not know the labels that control must go to at the time jump statements are generated

Translating Short-Circuit Expressions Using Backpatching

- This problem is solved by generating a series of branch statements with the targets of the jumps temporarily left unspecified.
- Each such statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined.
- This subsequent filling of addresses for the determined labels is called **BACKPATCHING**.

Syntax-Directed Definition for Backpatching

$E \rightarrow E$ **or** $M E$
| E **and** $M E$
| **not** E
| (E)
| **id relop id**
| **true**
| **false**

$M \rightarrow \varepsilon$

Synthesized attributes:

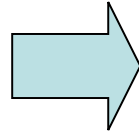
$E.code$	three-address code
$E.truelist$	backpatch list for jumps on true
$E.falselist$	backpatch list for jumps on false
$M.quad$	location of current three-address quad

Backpatch Operations with Lists

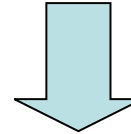
- *makelist(i)* creates a new list containing three-address location i , returns a pointer to the list
- *merge(p_1, p_2)* concatenates lists pointed to by p_1 and p_2 , returns a pointer to the concatenated list
- *backpatch(p, i)* inserts i as the target label for each of the statements in the list pointed to by p

Backpatching with Lists: Example

a < b or c < d and e < f



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



backpatch

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


Backpatching with Lists: Translation Scheme

$M \rightarrow \varepsilon$ { $M.\text{quad} := \text{nextquad}$ }

$E \rightarrow E_1 \text{ or } M E_2$ { $\text{backpatch}(E_1.\text{falselist}, M.\text{quad});$
 $E.\text{truelist} := \text{merge}(E_1.\text{truelist}, E_2.\text{truelist});$
 $E.\text{falselist} := E_2.\text{falselist}$ }

$E \rightarrow E_1 \text{ and } M E_2$ { $\text{backpatch}(E_1.\text{truelist}, M.\text{quad});$
 $E.\text{truelist} := E_2.\text{truelist};$
 $E.\text{falselist} := \text{merge}(E_1.\text{falselist}, E_2.\text{falselist});$ }

$E \rightarrow \text{not } E_1$ { $E.\text{truelist} := E_1.\text{falselist};$ $E.\text{falselist} := E_1.\text{truelist}$ }

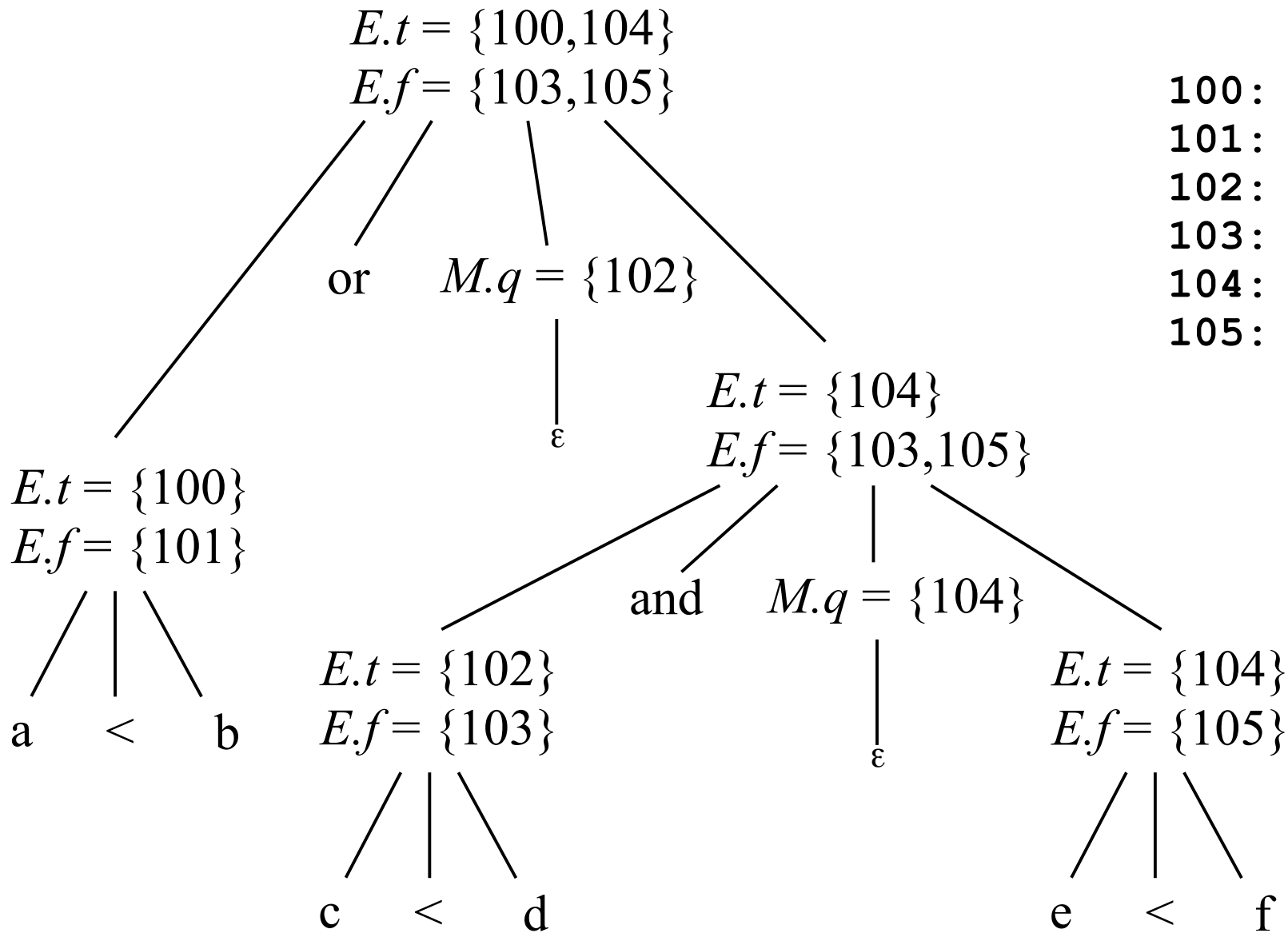
$E \rightarrow (E_1)$ { $E.\text{truelist} := E_1.\text{truelist};$ $E.\text{falselist} := E_1.\text{falselist}$ }

$E \rightarrow \text{id}_1 \text{ relop } \text{id}_2$ { $E.\text{truelist} := \text{makelist}(\text{nextquad});$
 $E.\text{falselist} := \text{makelist}(\text{nextquad} + 1);$
 $\text{emit}(\text{'if' id}_1.\text{place relop.op id}_2.\text{place 'goto _'});$
 $\text{emit}(\text{'goto _'})$ }

$E \rightarrow \text{true}$ { $E.\text{truelist} := \text{makelist}(\text{nextquad});$
 $E.\text{falselist} := \text{nil};$
 $\text{emit}(\text{'goto _'})$ }

$E \rightarrow \text{false}$ { $E.\text{falselist} := \text{makelist}(\text{nextquad});$
 $E.\text{truelist} := \text{nil};$
 $\text{emit}(\text{'goto _'})$ }

Backpatching Example



```

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

```

Flow-of-Control Statements and Backpatching: Grammar

$S \rightarrow \text{if } E \text{ then } S$

| $\text{if } E \text{ then } S \text{ else } S$

| $\text{while } E \text{ do } S$

| $\text{begin } L \text{ end}$

| A

$L \rightarrow L ; S$

| S

Synthesized attributes:

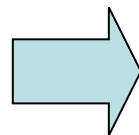
$S.\text{nextlist}$

backpatch list for jumps to the next statement after S (or nil)

$L.\text{nextlist}$

backpatch list for jumps to the next statement after L (or nil)

$S_1 ; S_2 ; S_3 ; S_4 ; S_4 \dots$



100: Code for S1
 200: Code for S2
 300: Code for S3
 400: Code for S4
 500: Code for S5

Jumps
out of S_1



$\text{backpatch}(S_1.\text{nextlist}, 200)$

$\text{backpatch}(S_2.\text{nextlist}, 300)$

$\text{backpatch}(S_3.\text{nextlist}, 400)$

$\text{backpatch}(S_4.\text{nextlist}, 500)$

Flow-of-Control Statements and Backpatching

$S \rightarrow A$	{ $S.nextlist := nil$ }
$S \rightarrow \mathbf{begin} L \mathbf{end}$	{ $S.nextlist := L.nextlist$ }
$S \rightarrow \mathbf{if} E \mathbf{then} M S_1$	{ $backpatch(E.truelist, M.quad);$ $S.nextlist := merge(E.falselist, S_1.nextlist)$ }
$L \rightarrow L_1 ; M S$	{ $backpatch(L_1.nextlist, M.quad); L.nextlist := S.nextlist;$ }
$L \rightarrow S$	{ $L.nextlist := S.nextlist;$ }
$M \rightarrow \varepsilon$	{ $M.quad := nextquad$ }
$S \rightarrow \mathbf{if} E \mathbf{then} M_1 S_1 N \mathbf{else} M_2 S_2$	{ $backpatch(E.truelist, M_1.quad);$ $backpatch(E.falselist, M_2.quad);$ $S.nextlist := merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist))$ }
$S \rightarrow \mathbf{while} M_1 E \mathbf{do} M_2 S_1$	{ $backpatch(S_1.nextlist, M_1.quad);$ $backpatch(E.truelist, M_2.quad);$ $S.nextlist := E.falselist;$ $emit(\mathbf{'goto M_1.quad'})$ }
$N \rightarrow \varepsilon$	{ $N.nextlist := makelist(nextquad); emit(\mathbf{'goto _'})$ }