Intermediate Code Generation

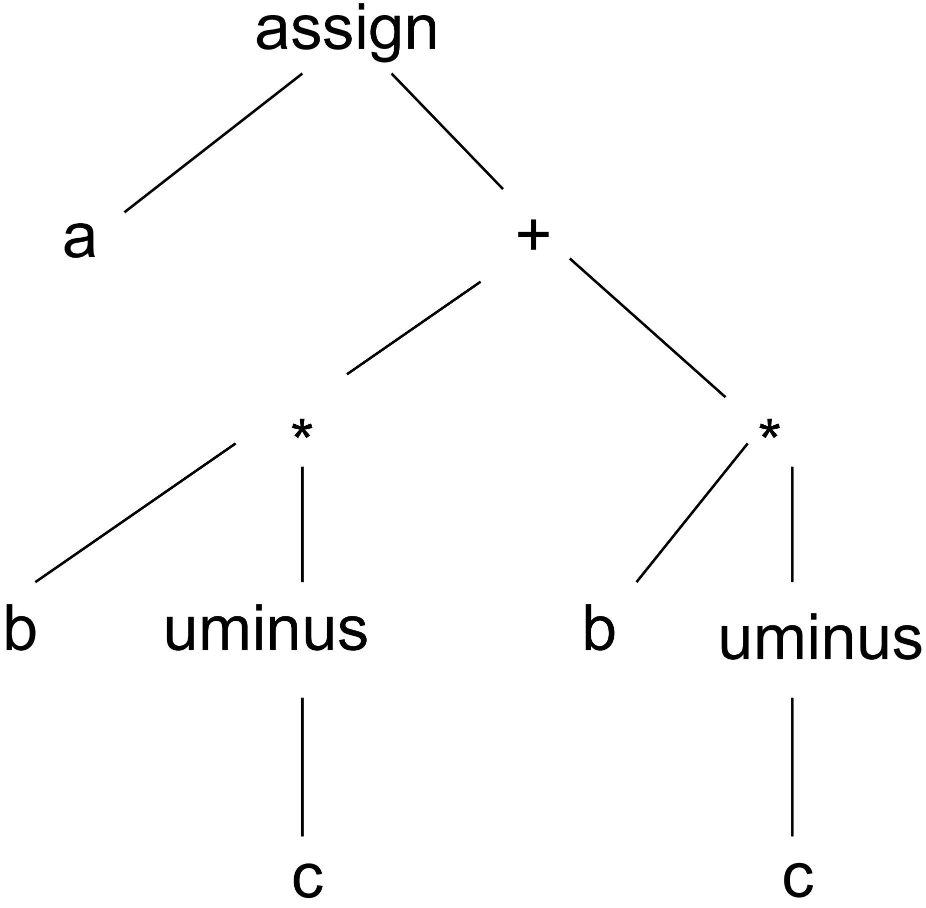
Intermediate Code Generation

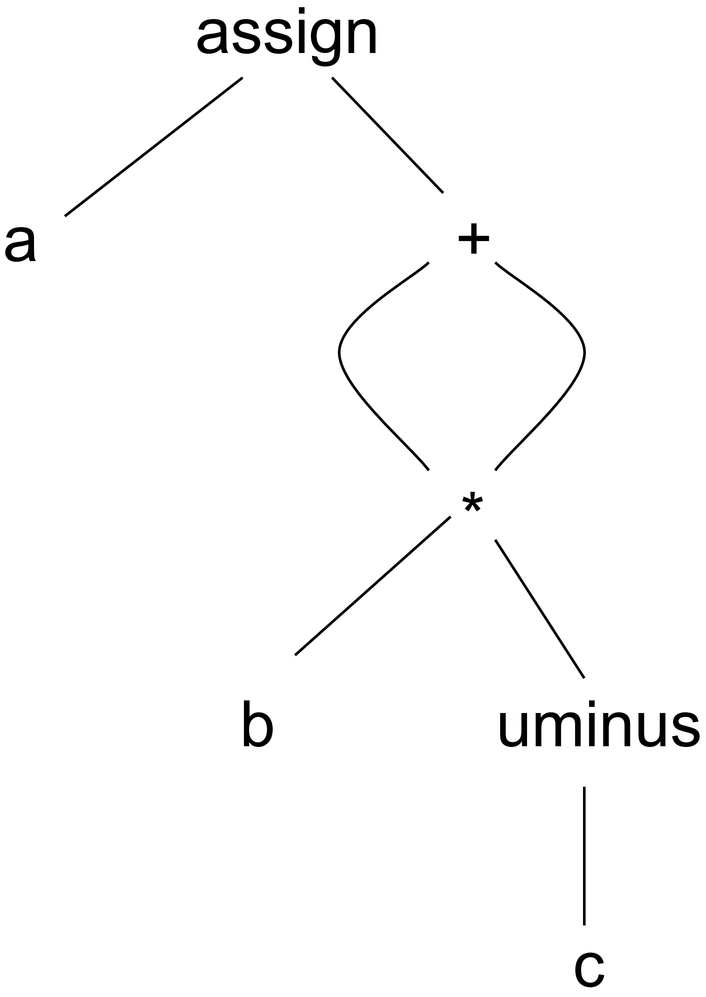
- Translating source program into an “intermediate language"
  - Simple
  - CPU Independent,
  - … yet, close in spirit to machine language
- Benefits
  - Retargeting is facilitated
  - Machine independent code optimization can be applied

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

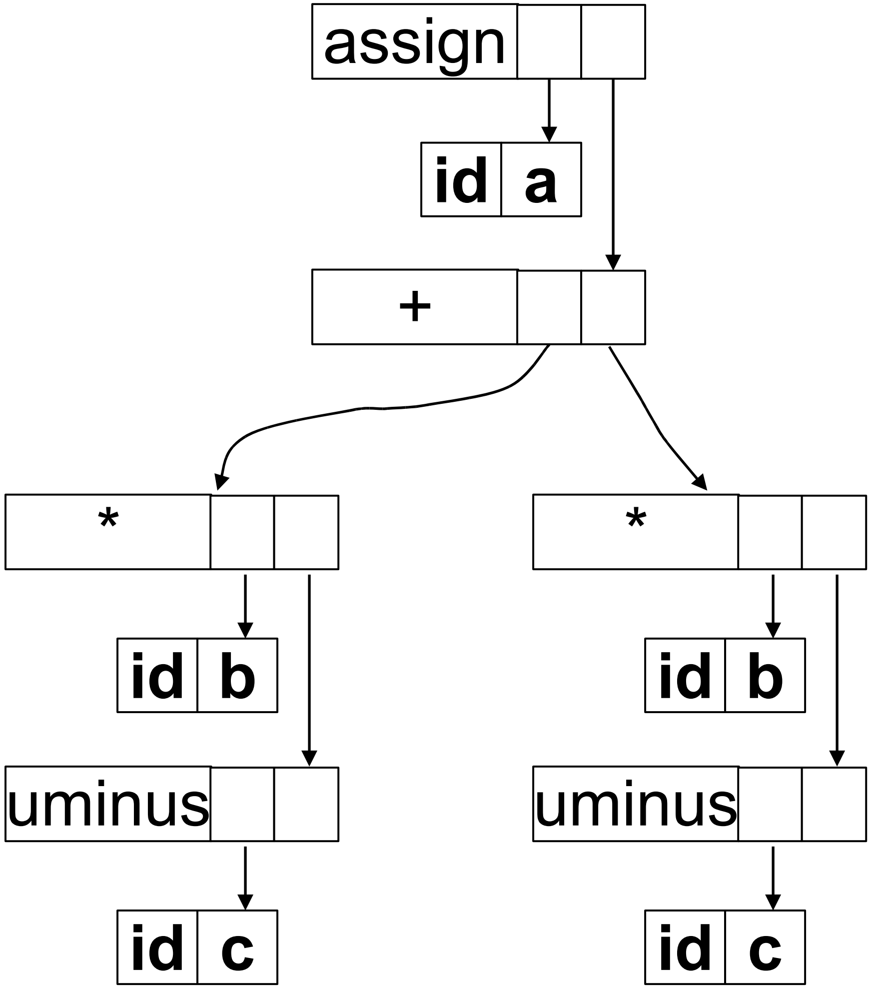
Postfix notation represents operations on a stack

Pro: easy to generate
Con: stack operations are more difficult to optimize

Bytecode (for example)

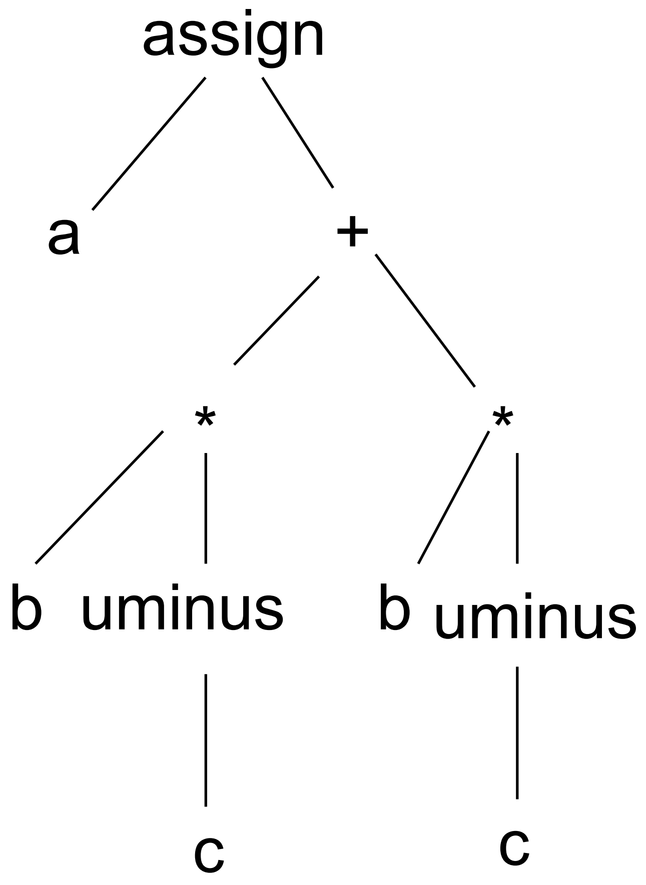
```
iload 2       // push b
iload 3       // push c
ineg          // uminus
imul          // *
iload 2       // push b
iload 3       // push c
ineg          // uminus
imul          // *
iadd          // +
istore 1      // store a
```

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

t_1:=-c

t_2:=b*t_1

t_3:=-c

t_4:=b*t_3

t_5:=t_2+t_4

a:=t_5

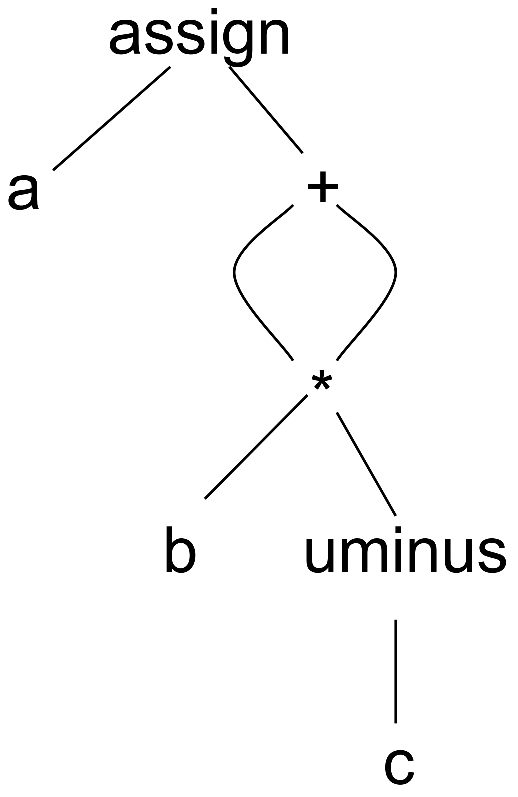
Dag

t_1:=-c

t_2:=b*t_1

t_5:=t_2+t_2

a:=t_5



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

Types of Three-Address Code Instructions

- Binary operations:
  $x := y \; op \; z$
- Unary operations:
  $x := op \; y$
- Copy instructions:
  $x := y$
- Conditional jumps:
  if x *relop* y *goto* L
- Procedure calls:
  param x_1
  param x_2
  …
  param x_n
  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

$$\mathit{gen}(E.\text{place}\ \text{‘:=’}\ E_1.\text{place}\ \text{‘+’}\ E_2.\text{place})$$

Code generation

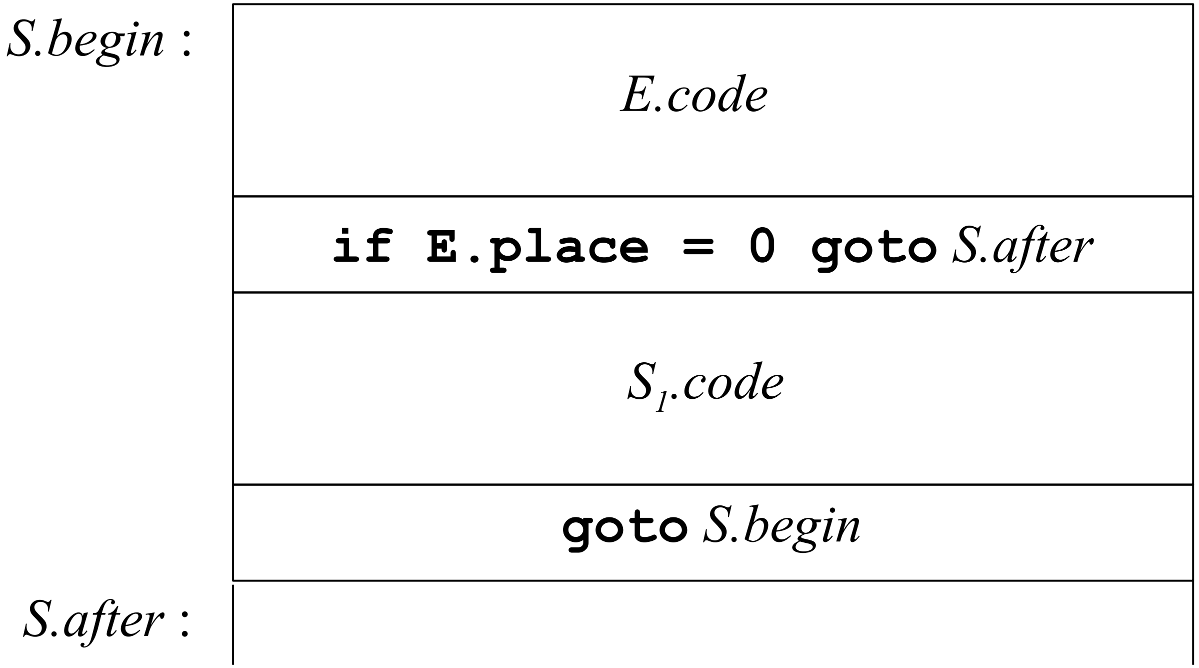
```
t3 := t1 + t2
```

Code Generation for Expressions

<table>
<tr><th>Production</th><th>Semantic Rules</th></tr>
<tr><td>$S \rightarrow$ id:=E</td><td>S.code:=E.code||gen(id.place ':=' E.place)</td></tr>
<tr><td>$E \rightarrow E_1 + E_2$</td><td>E.place:=newtemp;
E.code:=E_1.code||E_2.code||gen(E.place ':=' E_1.place '+' E_2.place)</td></tr>
<tr><td>$E \rightarrow E_1 * E_2$</td><td>E.place:=newtemp;
E.code:=E_1.code||E_2.code||gen(E.place ':=' E_1.place '*' E_2.place)</td></tr>
<tr><td>$E \rightarrow -E_1$</td><td>E.place:=newtemp;
E.code:=E_1.code||gen(E.place ':=' 'uminus' E_1.place)</td></tr>
<tr><td>$E \rightarrow (E_1)$</td><td>E.place:=E_1.place;
E.code:=E_1.code</td></tr>
<tr><td>$E \rightarrow$ id</td><td>E.place := id.place
E.code:=''</td></tr>
</table>

Code Generation for *while* Loops

Production	**Semantic Rules**
S→**while** E **do** S_1	S.begin:=newlabel; S.after:=newlabel; S.code:=gen(S.begin ':') || E.code|| gen('if' E.place '=' '0' 'goto' S.after) || S_1.code || gen('goto' S.begin)|| gen(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:
```

Quadruple Representation of Three-Address Statements

	op	arg1	arg2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

t_1:=- c
t_2:=b * t_1
t_3:=- c
t_4:=b * t_3
t_5:=t_2 + t_4
a:=t_5

Temporary names must be entered into the symbol table as they are created

Triple Representation of Three-Address Statements

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

$t_1 := - c$
$t_2 := b * t_1$
$t_3 := - c$
$t_4 := b * t_3$
$t_5 := t_2 + t_4$
$a := t_5$

Temporary names are not entered into the symbol table

Other Types of 3-address Statements

- Ternary operations like
  - *x[i]:=y* *x:=y[i]*
- require two or more entries:

	op	arg1	arg2
(0)	[]=	x	i
(1)	assign	(0)	y

x[i] := y

	op	arg1	arg2
(0)	=[]	y	i
(1)	assign	x	(0)

x := y[i]

Indirect Triples Representation of Three-Address Statements

	Instruction
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg1	arg2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Declarations

$P \to$ { *offset* := 0 } D

$D \to$ D ; D

$D \to$ **id** : T { *enter*(**id**.*name*, *T*.*type*, *offset*);
offset := *offset* + *T*.*width* }

$T \to$ **char** { *T*.*type* := *char*; *T*.*width* := 1 }

$T \to$ **integer** { *T*.*type* := *integer*; *T*.*width* := 4 }

$T \to$ **real** { *T*.*type* := *real*; *T*.*width* := 8 }

$T \to$ ^T_1 { *T*.*type* := *pointer*(T_1.*type*); *T*.*width* := 4 }

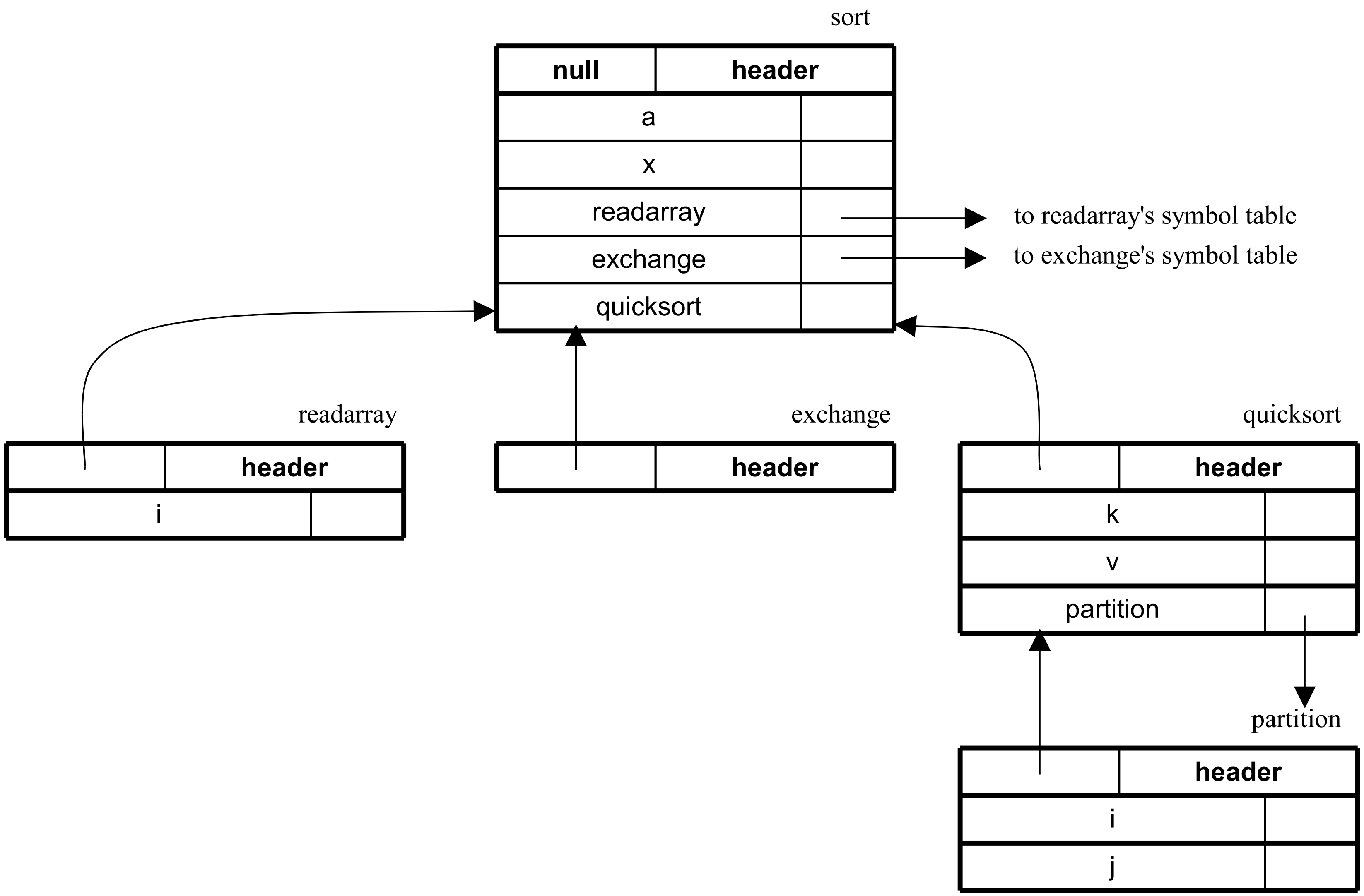
$T \to$ **array [num] of** T_1
{ *T*.*type* := *array*(1..**num**.*val*, T_1.*type*) ;
T.*width* := **num**.*val* * T_1.*width* }

name	**type**	**offset**
a	*char*	0
b	*pointer*(*real*)	1
c	*real*	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
  - *enter(symtable,name,type,offset)* – create a new 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



sort

null	header
a	
x	
readarray	
exchange	
quicksort	

to readarray's symbol table

to exchange's symbol table

readarray

	header
i	

exchange

	header

quicksort

	header
k	
v	
partition	

partition

	header
i	
j	

Nested Procedure Declarations

Production	Semantic Rules
$P \rightarrow MD$	addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset)
$M \rightarrow \in$	t:=mktable(nil); push(t, tblptr); push(0,offset)
$D \rightarrow D_1 ; D_2$	

D→**proc id**; ND_1; S	t:=top(tblptr);
	addwidth(t,top(offset));
	pop(tblptr); pop(offset);
	enterproc(top(tblptr), **id**.name, t)

D→**id**: T	enter(top(tblptr), **id**.name, T.type, top(offset)); top(offset) := top(offset)+T.width
$N \rightarrow \in$	t:=mktable(top(tblptr)); push(t,tblptr); push(0,offset)

Code Generation for Expressions

Production	**Semantic Rules**

$S \rightarrow$ **id**:=E	p:=lookup(**id**.name);
	if p<> nil **then**
	emit(p ':=' E.place)
	else error

$E \rightarrow E_1 + E_2$	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

$E \rightarrow$ id	p:= lookup(**id**.name);
	if p<> nil **then**
	E.place:=p
	else error

Addresing of Array Elements

$A[low..high]$ $A[i]$ $base + (i - low) * w$

$i * w + (base - low*w)$

$A[low_1..high_1, low_2..high_2]$ $A[i_1,i_2]$ $n_2 = high_2 - low_2 + 1$

$base + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$

$((i_1 * n_2 + i_2) * w + (base - ((low_1 * n_2) + low_2) * w)$

Conversion of Types

```
$E \rightarrow E_1 + E_2$  { $E.place$ := $newtemp$;
    if $E_1.type$ = $integer$ and $E_2.type$ = $integer$ then begin
        emit ($E.place$ ':=' $E_1.place$ 'int+' $E_2.place$);
        $E.type$ := $integer$;
    end
    else if $E_1.type$ = $real$ and $E_2.type$ = $real$ then begin
        emit ($E.place$ ':=' $E_1.place$ 'real+' $E_2.place$);
        $E.type$ := $real$;
    end
    else if $E_1.type$ = $integer$ and $E_2.type$ = $real$ then begin
        $u$ := $newtemp$;
        emit ($u$ ':=' 'inttoreal' $E_1.place$);
        emit ($E.place$ ':=' $u$ 'real+' $E_2.place$ );
        $E.type$ := $real$;
    end
    else if $E_1.type$ = $real$ and $E_2.type$ = $integer$ then begin
        $u$ := $newtemp$;
        emit ($u$ ':=' 'inttoreal' $E_2.place$);
        emit ($E.place$ ':=' $E_1.place$ 'real+' $u$);
        $E.type$ := $real$;
    end
    else
        $E.type$ := $type\_error$

}
```

```
x := y + i * j

$t_1$ := i int* j
$t_3$ := inttoreal $t_1$
$t_2$ := y real+ $t_3$
x := $t_2$
```

Boolean Expressions

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

<u>**`a or b and not c`**</u>

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

<u>**`a < b ⇒ if a < b then 1 else 0`**</u>

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

Boolean Expressions

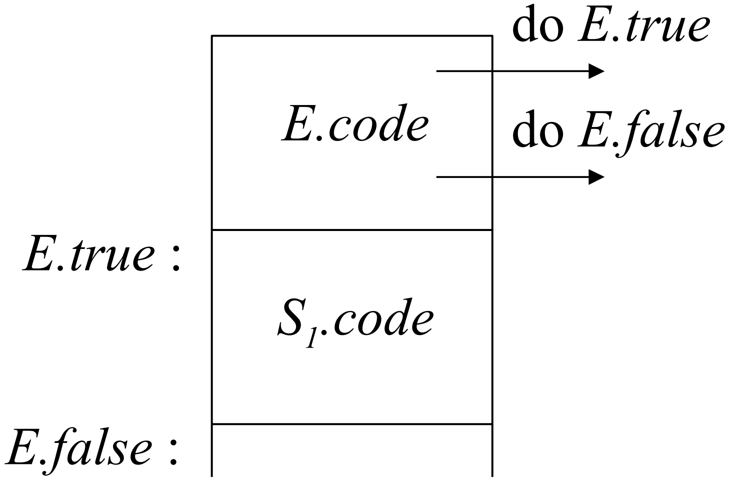
$E \rightarrow E_1$ **or** E_2	{ *E.place* := *newtemp; emit* (*E.place* ':=' E_1*.place* '`or`' E_2*.place*) }
$E \rightarrow E_1$ **and** E_2	{ *E.place* := *newtemp; emit* (*E.place* ':=' E_1*.place* '`and`' E_2*.place*) }
$E \rightarrow$ **not** E_1	{ *E.place* := *newtemp; emit* (*E.place* ':=' '`not`' E_1*.place*) }
$E \rightarrow (E_1)$	{ *E.place* := E_1*.place* }
$E \rightarrow$ **id**$_1$ **relop id**$_2$	{ *E.place* := *newtemp;* *emit* ('`if`' **id**$_1$*.place* **relop***.op* **id**$_2$*.place* '`goto`' *nextstat* + 3); *emit* (*E.place* ':=' '`0`') *emit* ('`goto`' *nextstat* + 2) *emit* (*E.place* ':=' '`1`') }
$E \rightarrow$ **true**	{ *E.place* := *newtemp; emit* (*E.place* ':=' '`1`') }
$E \rightarrow$ **false**	{ *E.place* := *newtemp; emit* (*E.place* ':=' '`0`') }

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
```

Control Instructions

$S \rightarrow$ **if** E **then** S_1



do $E.true$

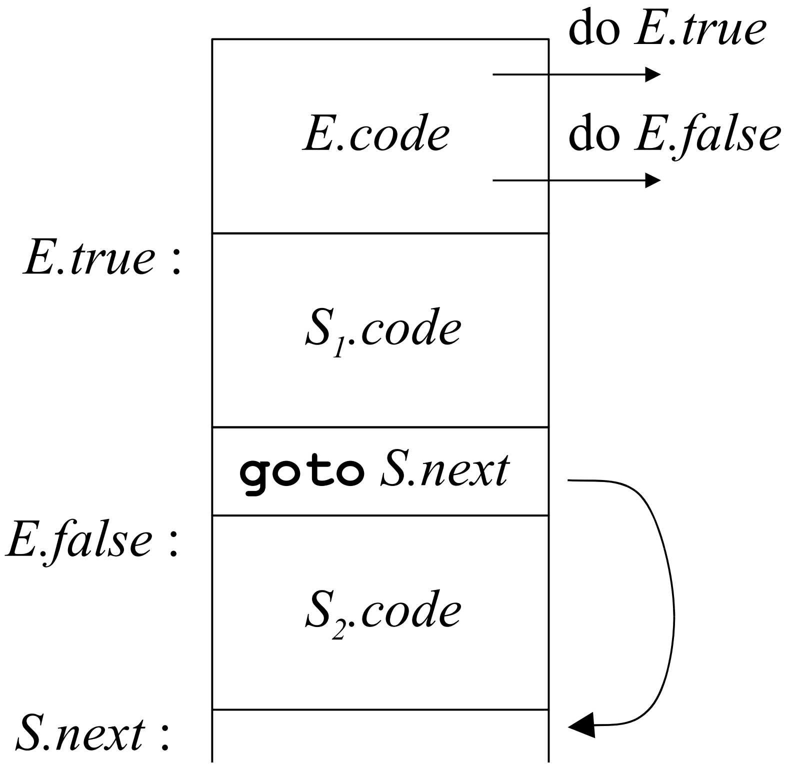
do $E.false$

$E.code$

$E.true$: $S_1.code$

$E.false$:

$S \rightarrow$ **if** E **then** S_1 **else** S_2



do $E.true$

do $E.false$

$E.code$

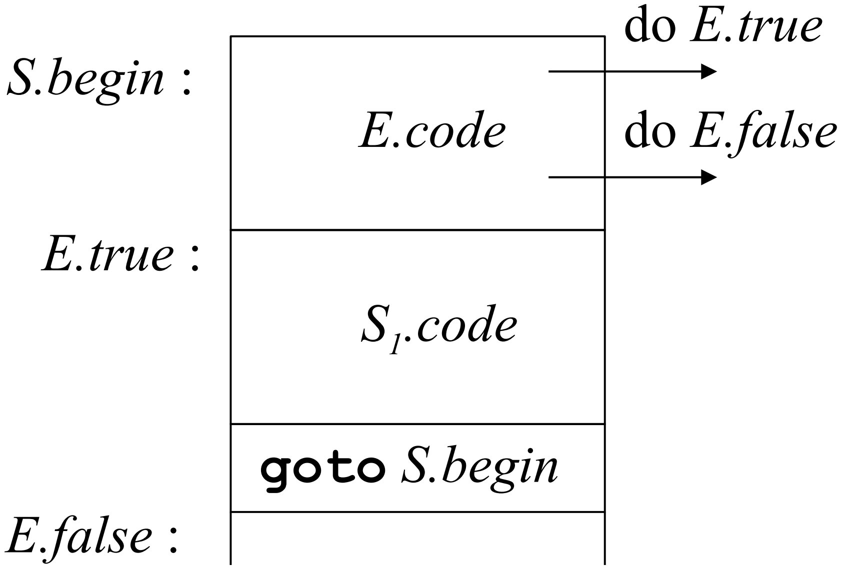
$E.true$: $S_1.code$

`goto` $S.next$

$E.false$: $S_2.code$

$S.next$:

$S \rightarrow$ **while** E **do** S_1



do $E.true$

do $E.false$

$S.begin$: $E.code$

$E.true$: $S_1.code$

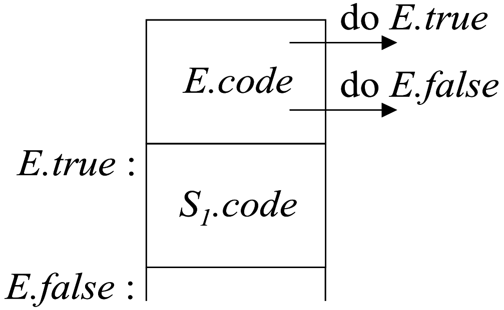
`goto` $S.begin$

$E.false$:

Dyntax-Directed Definition for Flow-of-Control Statements

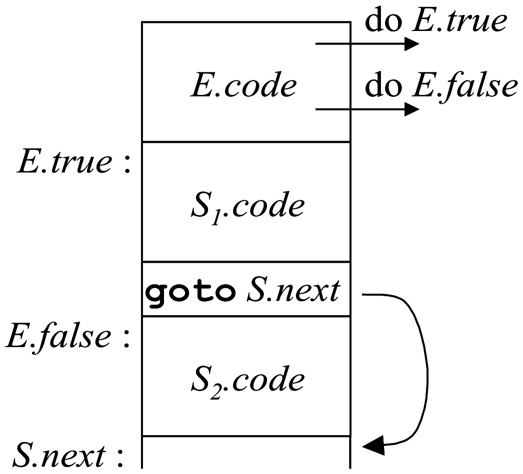
$S \rightarrow \textbf{if}\ E\ \textbf{then}\ S_1$

$E.true := newlabel;$
$E.false := S.next;$
$S_1.next := S.next;$
$S.code := E.code \parallel gen(E.true\ ':') \parallel S_1.code$


do $E.true$
$E.code$
do $E.false$
$E.true$:
$S_1.code$
$E.false$:

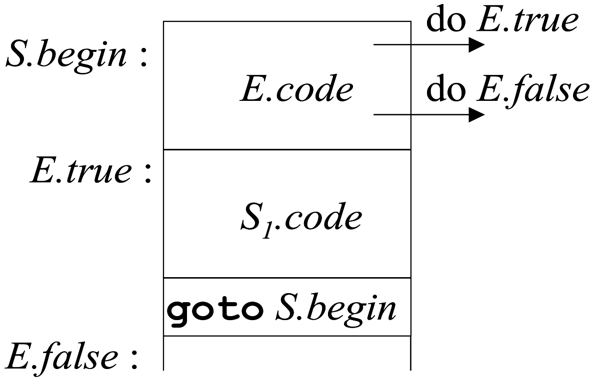
$S \rightarrow \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2$

$E.true := newlabel;$
$E.false := newlabel;$
$S_1.next := S.next;$
$S_2.next := S.next;$
$S.code := E.code \parallel gen(E.true\ ':') \parallel S_1.code \parallel$
$\quad gen('\texttt{goto}'\ S.next) \parallel gen(E.false\ ':') \parallel S_2.code$


do $E.true$
$E.code$
do $E.false$
$E.true$:
$S_1.code$
`goto` $S.next$
$E.false$:
$S_2.code$
$S.next$:

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

$S.begin := newlabel;$
$E.true := newlabel;$
$E.false := S.next$
$S_1.next := S.begin;$
$S.code := gen(S.begin\ ':') \parallel E.code \parallel$
$\quad gen(E.true\ ':') \parallel S_1.code \parallel$
$\quad gen('\texttt{goto}'\ S.begin)$


$S.begin$:
do $E.true$
$E.code$
do $E.false$
$E.true$:
$S_1.code$
`goto` $S.begin$
$E.false$:

Boolean Expression - "Short-Circuit" Translation

Code for E= a<b:

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

`goto` *E*.*false*

$E \rightarrow E_1$ **or** E_2	E_1.*true* := *E*.*true*; E_1.*false* := *newlabel*; E_2.*true* := *E*.*true*; E_2.*false* := *E*.*false*; *E*.*code* := E_1.*code* ‖ *gen*(E_1.*false* ':') ‖ E_2.*code*
$E \rightarrow E_1$ **and** E_2	E_1.*true* := *newlabel*; E_1.*false* := *E*.*false*; E_2.*true* := *E*.*true*; E_2.*false* := *E*.*false*; *E*.*code* := E_1.*code* ‖ *gen*(E_1.*true* ':') ‖ E_2.*code*
$E \rightarrow$ **not** E_1	E_1.*true* := *E*.*false*; E_1.*false* := *E*.*true*; *E*.*code* := E_1.*code*;
$E \rightarrow$ **(** E_1 **)**	E_1.*true* := *E*.*true*; E_1.*false* := *E*.*false*; *E*.*code* := E_1.*code*;
$E \rightarrow$ **id**$_1$**relop id**$_2$	*E*.*code* := *gen*('**`if`**' **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* '**`goto`**' *E*.*true*) ‖ *gen*('**`goto`**' *E*.*false*)
$E \rightarrow$ **true**	*E*.*code* := *gen*('**`goto`**' *E*.*true*)
$E \rightarrow$ **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)
```

$$E \rightarrow \quad E + E \mid E \textbf{ and } E \mid E \textbf{ relop } E \mid \textbf{id}$$

$E \rightarrow E_1 + E_2$

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

```
$E_2.true$  :  $E.place$ :=  $E_1.place$ + 1
               goto $nextstat$ + 1
$E_2.false$ :  $E.place$ :=  $E_1.place$
```

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** $M\ E$
$\mid E$ **and** $M\ E$
$\mid$ **not** E
$\mid$ **(** E **)**
$\mid$ **id relop id**
$\mid$ **true**
$\mid$ **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 _
```

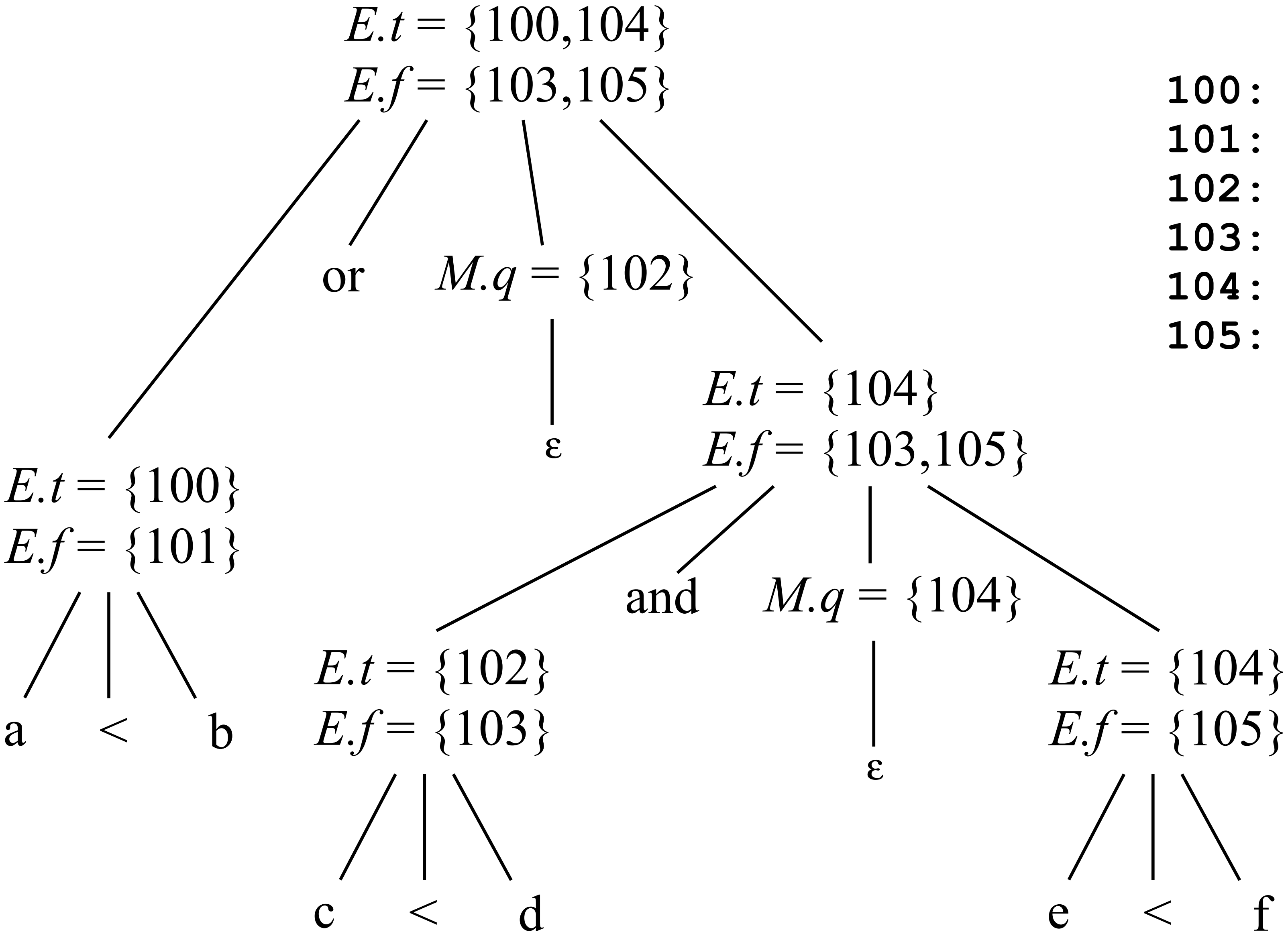
backpatch

```
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

$M \to \varepsilon$	{ M.quad := *nextquad* }
$E \to E_1$ **or** M E_2	{ *backpatch*(E_1.falselist, M.quad); E.truelist := *merge*(E_1.truelist, E_2.truelist); E.falselist := E_2.falselist }
$E \to E_1$ **and** M E_2	{ *backpatch*(E_1.truelist, M.quad); E.truelist := E_2.truelist; E.falselist := *merge*(E_1.falselist, E_2.falselist); }
$E \to$ **not** E_1	{ E.truelist := E_1.falselist; E.falselist := E_1.truelist }
$E \to$ **(** E_1 **)**	{ E.truelist := E_1.truelist; E.falselist := E_1.falselist }
$E \to$ **id**$_1$ **relop id**$_2$	{ E.truelist := *makelist*(*nextquad*); E.falselist := *makelist*(*nextquad* + 1); *emit*(‘`if`’ **id**$_1$.place **relop**.op **id**$_2$.place ‘`goto _`’); *emit*(‘`goto _`’) }
$E \to$ **true**	{ E.truelist := *makelist*(*nextquad*); E.falselist := nil; *emit*(‘`goto _`’) }
$E \to$ **false**	{ E.falselist := *makelist*(*nextquad*); E.truelist := nil; *emit*(‘`goto _`’) }

Backpatching Example



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

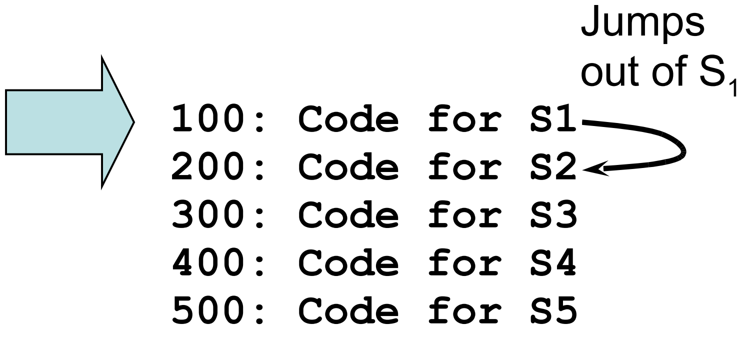
Flow-of-Control Statements and Backpatching: Grammar

$$
\begin{aligned}
S &\rightarrow \textbf{if } E \textbf{ then } S \\
&\mid \textbf{if } E \textbf{ then } S \textbf{ else } S \\
&\mid \textbf{while } E \textbf{ do } S \\
&\mid \textbf{begin } L \textbf{ end} \\
&\mid A \\
L &\rightarrow L \textbf{ ; } S \\
&\mid S
\end{aligned}
$$

Synthesized attributes:

S.nextlist	backpatch list for jumps to the next statement after *S* (or nil)
L.nextlist	backpatch list for jumps to the next statement after *L* (or nil)

S_1 ; S_2 ; S_3 ; S_4 ; S_4 …



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

backpatch(S_1.nextlist, 200)
backpatch(S_2.nextlist, 300)
backpatch(S_3.nextlist, 400)
backpatch(S_4.nextlist, 500)

Flow-of-Control Statements and Backpatching

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