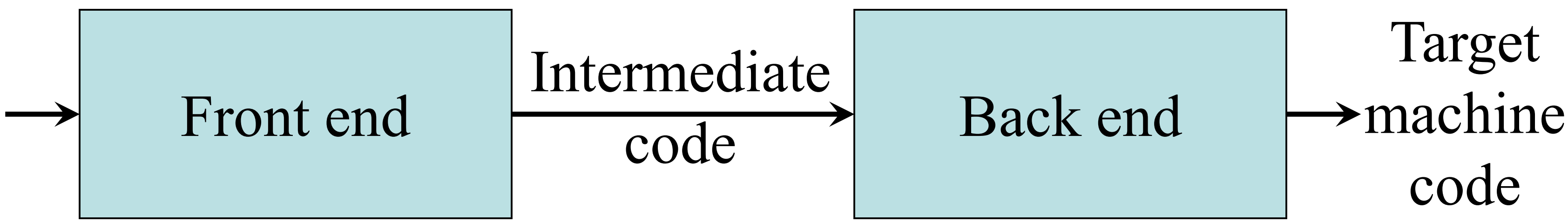
Intermediate Code Generation

Intermediate Code Generation

- Facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end



- Enables machine-independent code optimization

$T1=x*5$

$T1=x+x+x+x+x$

$T1=x+x$

$T2=x+x$

$T3=T1+T2$

$T4=X+t3$

Intermediate Representations

- *Graphical representations* (e.g. AST)
- *Postfix notation*: operations on values stored on operand stack (similar to JVM bytecode)
- *Three-address code*: (e.g. *triples* and *quads*)

  $x := y \text{ op } z$
- *Two-address code*:

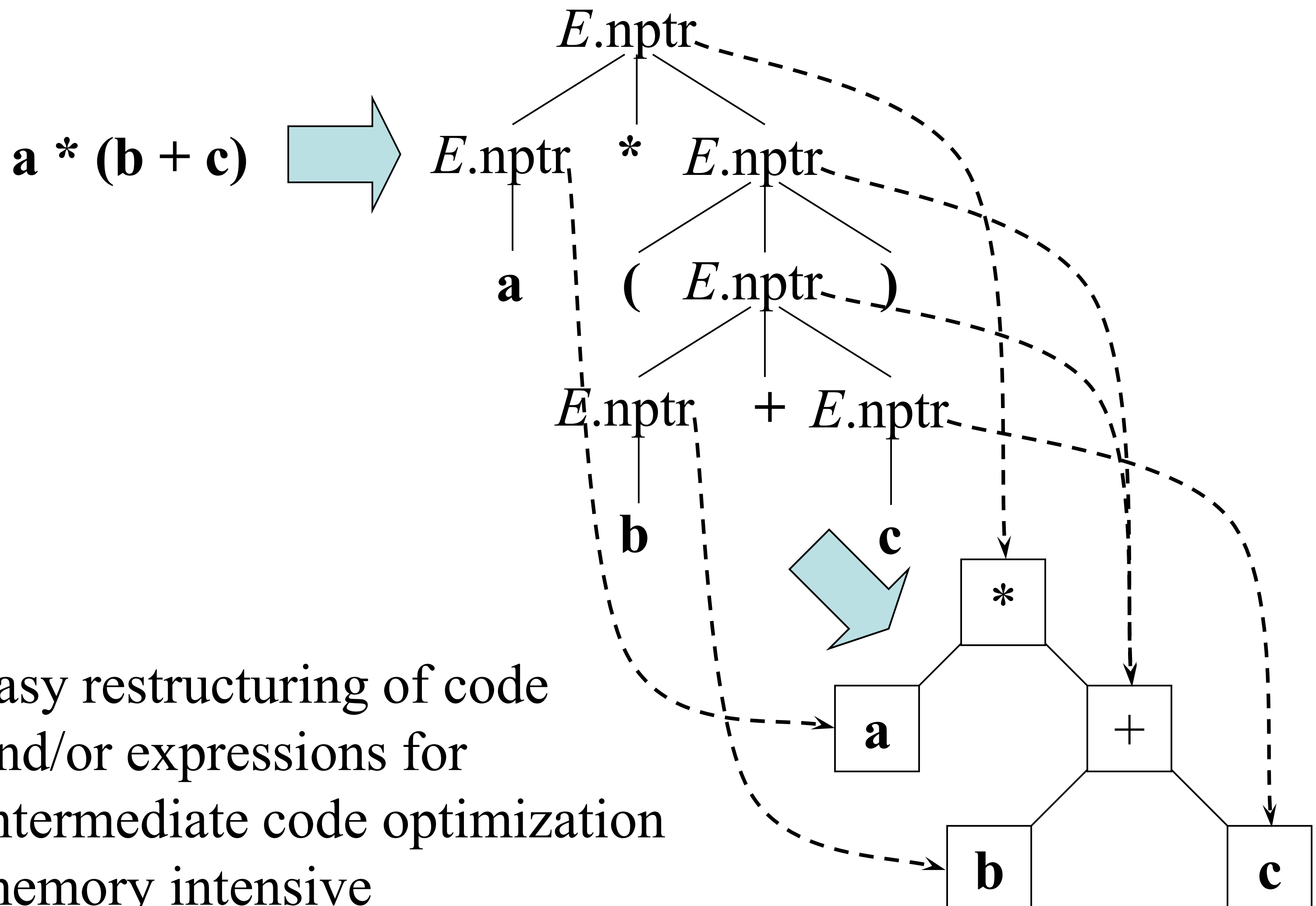
  $x := \text{op } y$

  which is the same as $x := x \text{ op } y$

Syntax-Directed Translation of Abstract Syntax Trees

Production	Semantic Rule
$S \rightarrow$ **id** **:=** E	S.nptr := $mknode$(':=' , $mkleaf$(**id**, **id**.entry), E.nptr)
$E \rightarrow E_1$ **+** E_2	E.nptr := $mknode$('+' , E_1.nptr, E_2.nptr)
$E \rightarrow E_1$ ***** E_2	E.nptr := $mknode$('*' , E_1.nptr, E_2.nptr)
$E \rightarrow$ **-** E_1	E.nptr := $mknode$('uminus' , E_1.nptr)
$E \rightarrow$ **(** E_1 **)**	E.nptr := E_1.nptr
$E \rightarrow$ **id**	E.nptr := $mkleaf$(**id**, **id**.entry)

Abstract Syntax Trees



Pro: easy restructuring of code and/or expressions for intermediate code optimization

Cons: memory intensive

Prog→if exp then st else st

Exp→E relop E|E+E|(E)|E*E

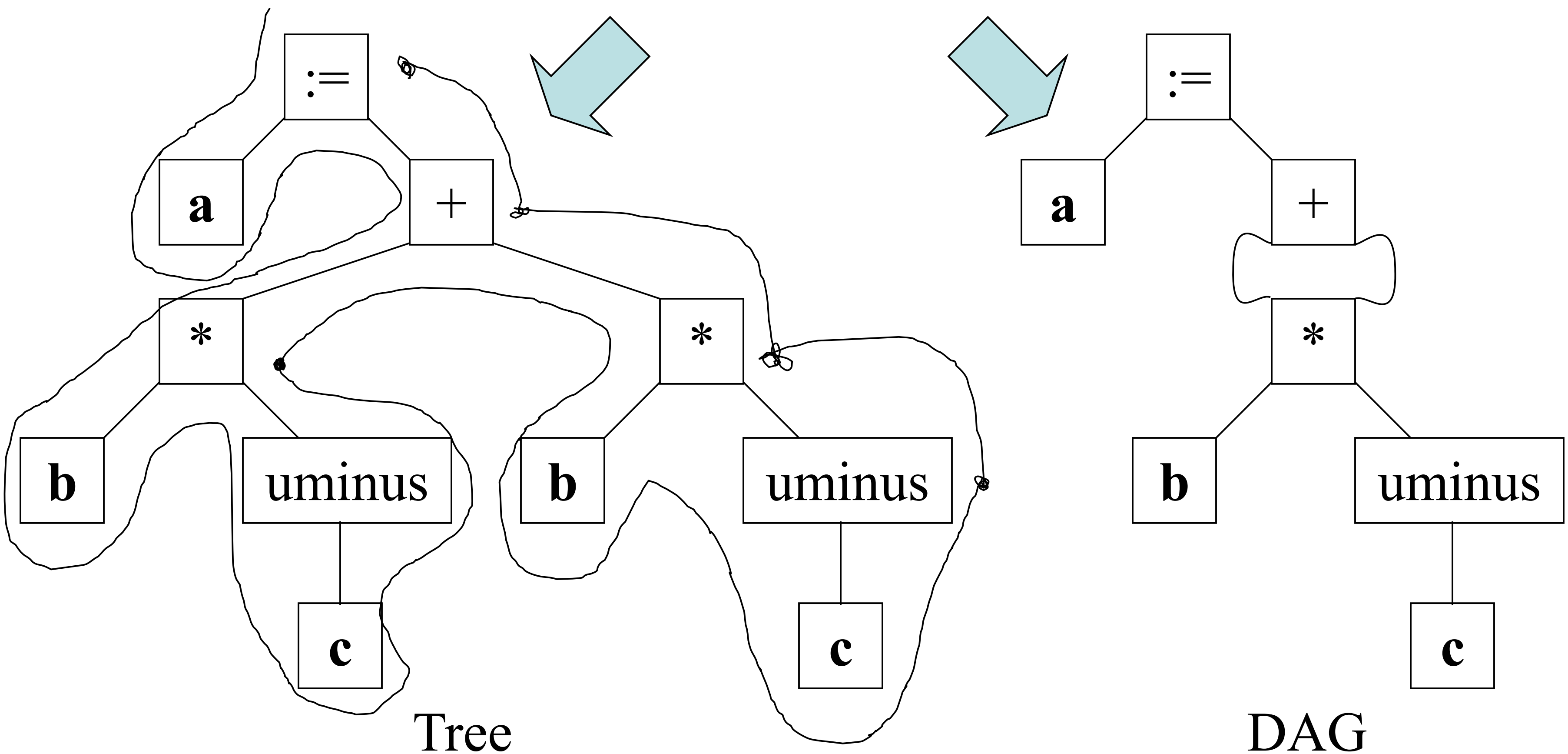
E→id|number

St→id := exp

if x>1 then x=2*(y+1) else y=y+1

Abstract Syntax Trees versus DAGs

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



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

Cons: 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
```

P-code

- Stack based virtual m/c
- Operands are always stack[top]
- Push operands onto STACK
- E1 * E2
- Code to evaluate E1 and then E2
- Mult

Three-Address Code

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

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

Linearized representation of a syntax tree

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

Linearized representation of a syntax DAG

Three-Address Statements

- Assignment statements: x **:=** y *op* z, x **:=** *op* y
- Indexed assignments: x **:=** $y[i]$, $x[i]$ **:=** y
- Pointer assignments: x **:= &**y, x **:= ***y, *****x **:=** y
- Copy statements: x **:=** y
- Unconditional jumps: **goto** *lab*
- Conditional jumps: **if** x *relop* y **goto** *lab*
- Function calls: **param** x… **call** p, n
  return y
- *A=fun(para1,para2,….) return x*

Syntax-Directed Translation into Three-Address Code

Productions

$S \rightarrow$ **id :=** E

| **while** E **do** S

$E \rightarrow E + E$

| E ***** E

| **-** E

| **(** E **)**

| **id**

| **num**

Synthesized attributes:

S.code	three-address code for S
S.begin	label to start of S or nil
S.after	label to end of S or nil
E.code	three-address code for E
E.place	a name holding the value of E

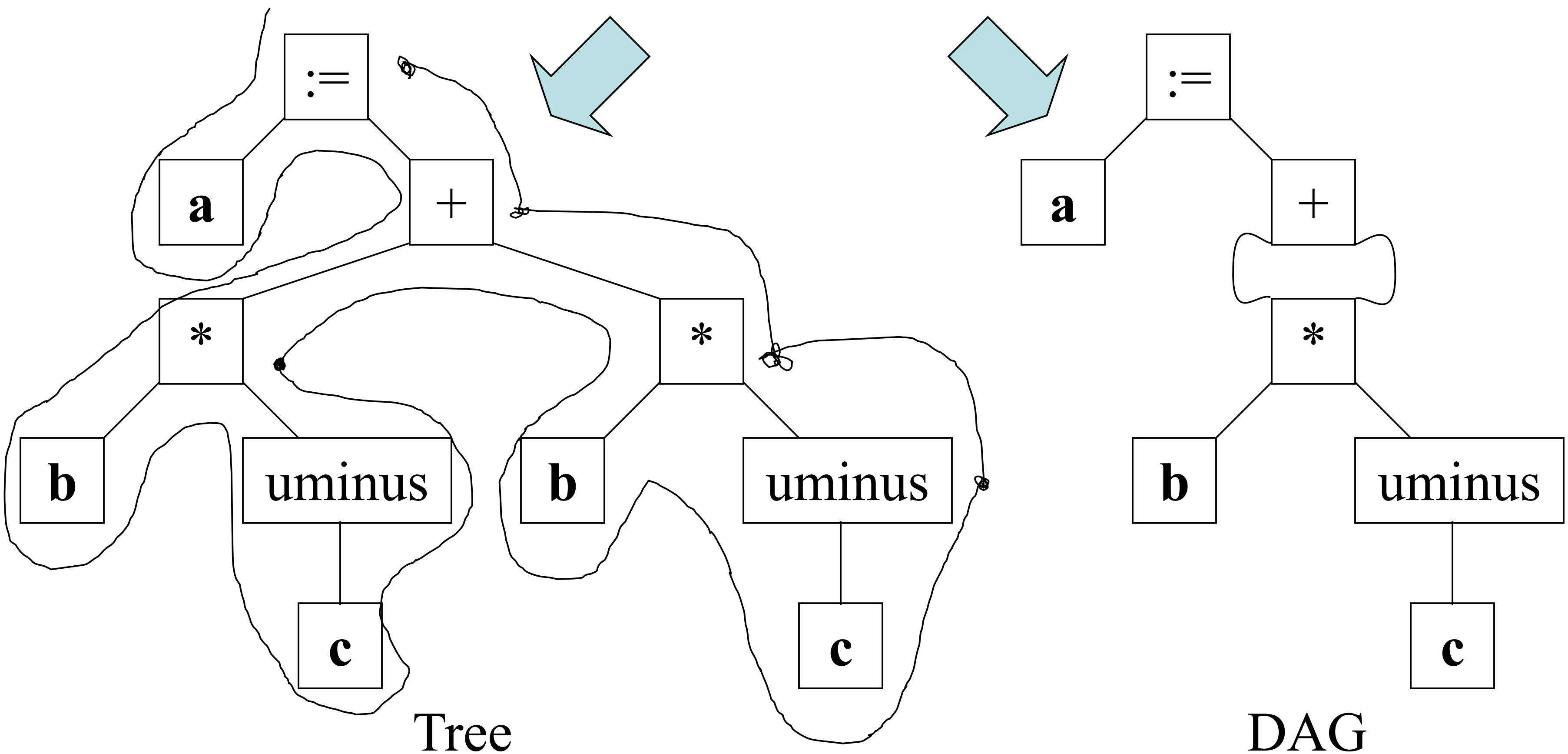
$gen(E.\text{place}\ \text{':='}\ E_1.\text{place}\ \text{'+'}\ E_2.\text{place})$

Code generation

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

Abstract Syntax Trees versus DAGs

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



Syntax-Directed Translation into Three-Address Code (cont'd)

Productions	Semantic rules
$S \to$ **id** **:=** E	S.code := E.code ‖ *gen*(**id**.place ':=' E.place); S.begin := S.after := nil
$S \to$ **while** E **do** S_1	(*see next slide)*
$E \to E_1 + E_2$	E.place := *newtemp*(); E.code := E_1.code ‖ E_2.code ‖ *gen*(E.place ':=' E_1.place '+' E_2.place)
$E \to E_1$ * E_2	E.place := *newtemp*(); E.code := E_1.code ‖ E_2.code ‖ *gen*(E.place ':=' E_1.place '*' E_2.place)
$E \to$ **-** E_1	E.place := *newtemp*(); E.code := E_1.code ‖ *gen*(E.place ':=' 'uminus' E_1.place)
$E \to$ **(** E_1 **)**	E.place := E_1.place E.code := E_1.code
$E \to$ **id**	E.place := **id**.name E.code := ''
$E \to$ **num**	E.place := *newtemp*(); E.code := *gen*(E.place ':=' **num**.value)

Syntax-Directed Translation into Three-Address Code (cont'd)

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

Semantic rule
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 ':')

S.begin:	E.code
	`if` E.place `= 0 goto` S.after
	S.code
	`goto` S.begin
S.after:	…

Example

S.begin:	*E*.code
	`if` *E*.place `= 0 goto` *S*.after
	S.code
	`goto` *S*.begin
S.after:	…

i := 2 * n + k
while i do
i := i - k
a := 10

```
    t1  := 2
    t2  := t1 * n           E.code
    t3  := t2 + k
    i   := t3
L1: if i = 0 goto L2        if code
    t4  := i - k
    i   := t4               S.code
    goto L1                  goto S.begin
L2: t5  := 10               S.after
```

Implementation of Three-Address Statements: Quads

#	*Op*	*Arg1*	*Arg2*	*Res*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

```
t1 := - c

t2 := b * t1
t3 := - c

t4 := b * t3
t5 := t2 + t4
a  := t5
```

Quads (quadruples)

Pro: easy to rearrange code for global optimization
Cons: lots of temporaries

Implementation of Three-Address Statements: Triples

#	*Op*	*Arg1*	*Arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

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

Triples

Pro: temporaries are implicit
Cons: difficult to rearrange code

Implementation of Three-Address Stmts: Indirect Triples

#	*Stmt*
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Program

#	*Op*	*Arg1*	*Arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Triple container

Pro: temporaries are implicit & easier to rearrange code

$E \rightarrow E+E|(E)|-E|id$

a+-(b+c)

Quad?

Triples?

Indirect triples?

(0)=b+c

(1)=-(0)

(2)=a+(1)

Names and Scopes

- The three-address code generated by the syntax-directed definitions shown on the previous slides is somewhat simplistic, because it assumes that the names of variables can be easily resolved by the back end in global or local variables
- We need local symbol tables to record global declarations as well as local declarations in procedures, blocks, and structs to resolve names

Symbol Tables for Scoping

```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void somefunc()
{ …
  swap(s.a, s.b);
  …
}
```

We need a symbol table for the *fields* of struct S

Need symbol table for *global* variables and functions

Need symbol table for *arguments* and *locals* for each function

Check: **`s`** is global and has fields **`a`** and **`b`**

Using symbol tables we can generate code to access **`s`** and its fields

Offset and Width for Runtime Allocation

```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void somefunc()
{ …
  swap(s.a, s.b);
  …
}
```

The fields **a** and **b** of struct S are located at *offsets* 0 and 4 from the start of S

The *width* of S is 8

a	(0)
b	(4)

Subroutine frame holds arguments **a** and **b** and local **t** at *offsets* 0, 4, and 8

Subroutine frame

fp[0]=	**a**	(0)
fp[4]=	**b**	(4)
fp[8]=	**t**	(8)

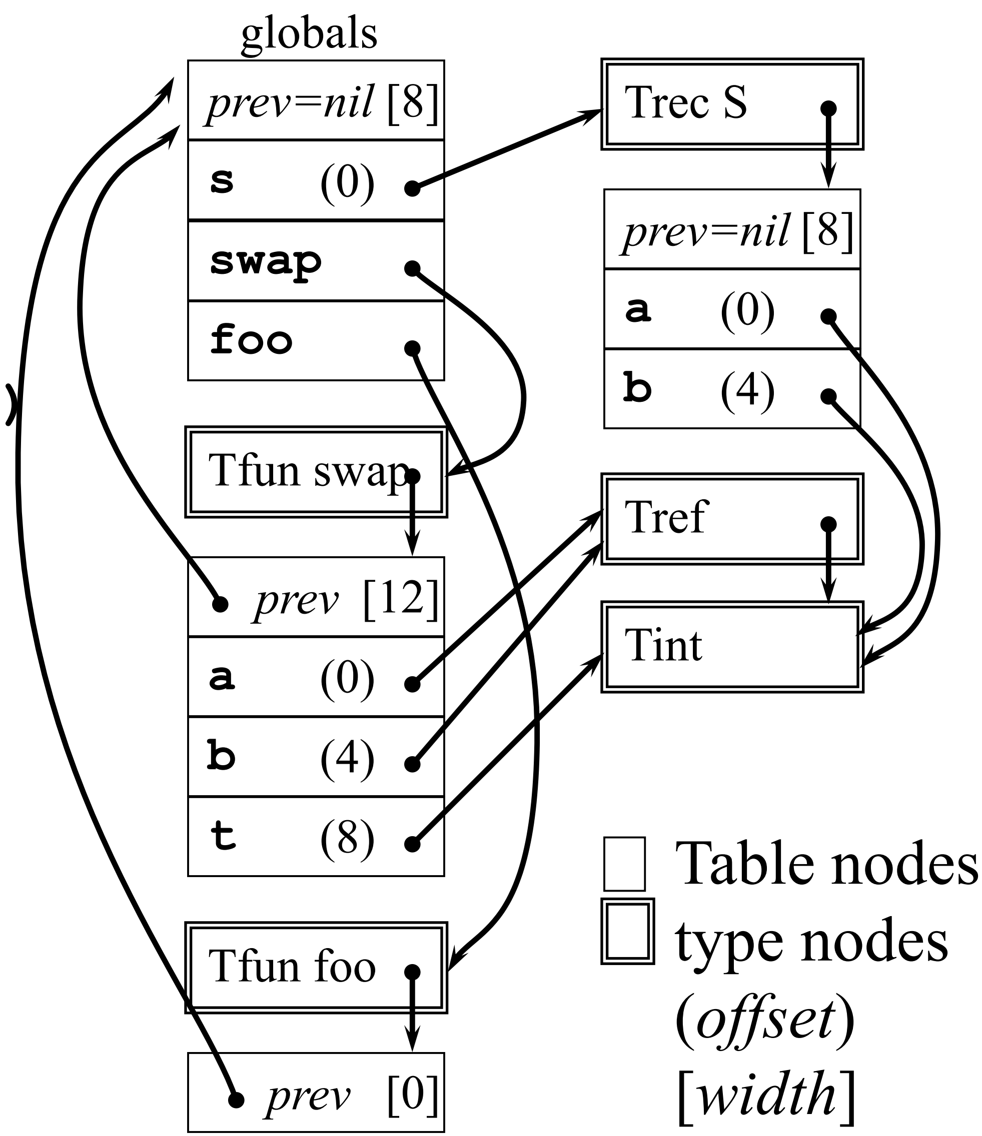
The *width* of the frame is 12

Symbol Tables for Scoping

```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void foo()
{ …
  swap(s.a, s.b);
  …
}
```



Hierarchical Symbol Table Operations

- *mktable*(*previous*) returns a pointer to a new table that is linked to a previous table in the outer scope
- *enter*(*table*, *name*, *type*, *offset*) creates a new entry in *table*
- *addwidth*(*table*, *width*) accumulates the total width of all entries in *table*
- *enterproc*(*table*, *name*, *newtable*) creates a new entry in *table* for procedure with local scope *newtable*
- *lookup*(*table*, *name*) returns a pointer to the entry in the table for *name* by following linked tables

Syntax-Directed Translation of Declarations in Scope

Productions

$P \rightarrow D$ **;** S
$D \rightarrow D$ **;** D
  | **id :** T
  | **proc id ;** D **;** S
$T \rightarrow$ **integer**
  | **real**
  | **array [num] of** T
  | **^** T
  | **record** D **end**
$S \rightarrow S$ **;** S
  | **id :=** E
  | **call id (** A **)**

Productions *(cont'd)*

$E \rightarrow E$ **+** E
  | E ***** E
  | **-** E
  | **(** E **)**
  | **id**
  | E **^**
  | **&** E
  | E **. id**
$A \rightarrow A$ **,** E
  | E

Synthesized attributes:

T.type	pointer to type
T.width	storage width of type (bytes)
E.place	name of temp holding value of *E*

Global data to implement scoping:

tblptr	stack of pointers to tables
offset	stack of offset values

Syntax-Directed Translation of Declarations in Scope (cont'd)

$P \rightarrow \quad \{\ t := mktable(\text{nil});\ push(t,\ tblptr);\ push(0,\ offset)\ \}$
$\qquad D\ \mathbf{;}\ S$

$D \rightarrow \mathbf{id}\ \mathbf{:}\ T$
$\qquad \{\ enter(top(tblptr),\ \mathbf{id}.\text{name},\ T.\text{type},\ top(offset));$
$\qquad\ \ top(offset) := top(offset) + T.\text{width}\ \}$

$D \rightarrow \mathbf{proc\ id\ ;}$
$\qquad \{\ t := mktable(top(tblptr));\ \ push(t,\ tblptr);\ push(0,\ offset)\ \}$
$\qquad D_1\ \mathbf{;}\ S$
$\qquad \{\ t := top(tblptr);\ addwidth(t,\ top(offset));$
$\qquad\ \ pop(tblptr);\ pop(offset);$
$\qquad\ \ enterproc(top(tblptr),\ \mathbf{id}.\text{name},\ t)\ \}$

$D \rightarrow D_1\ \mathbf{;}\ D_2$

Syntax-Directed Translation of Declarations in Scope (cont’d)

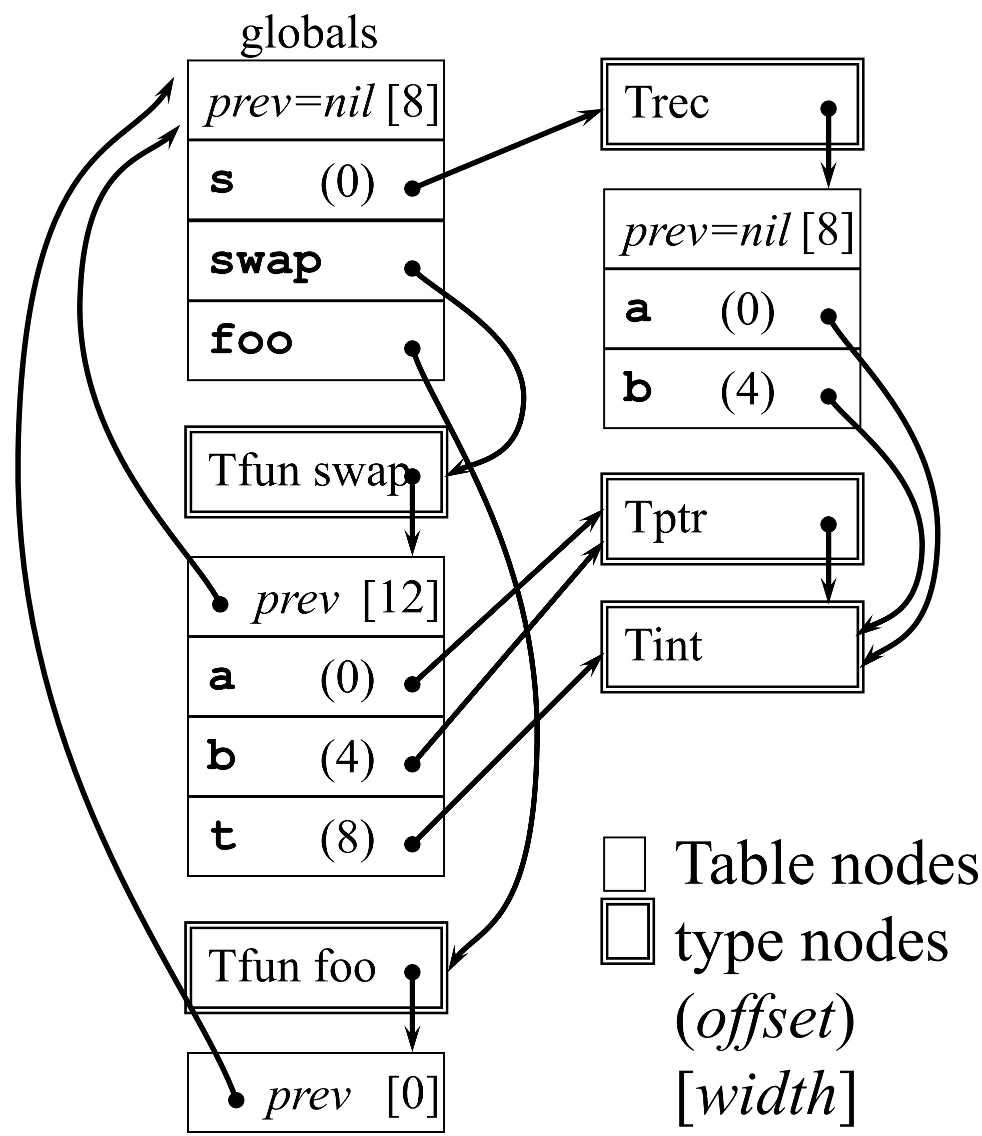
$T \rightarrow$ **integer** { T.type := ‘*integer*’; T.width := 4 }
$T \rightarrow$ **real** { T.type := ‘*real*’; T.width := 8 }
$T \rightarrow$ **array [num] of** T_1
{ T.type := *array*(**num**.val, T_1.type);
T.width := **num**.val * T_1.width }
$T \rightarrow$ ^ T_1
{ T.type := *pointer*(T_1.type); T.width := 4 }
$T \rightarrow$ **record**
{ *t* := *mktable*(nil); *push*(*t*, *tblptr*); *push*(0, *offset*) }
D **end**
{ T.type := *record*(*top*(*tblptr*)); T.width := *top*(*offset*);
addwidth(*top*(*tblptr*), *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*) }

Example

```
s: record
     a: integer;
     b: integer;
   end;

proc swap;
  a: ^integer;
  b: ^integer;
  t: integer;
  t := a^;
  a^ := b^;
  b^ := t;

proc foo;
  call swap(&s.a, &s.b);
```



Syntax-Directed Translation of Statements in Scope

$S \rightarrow S$ **;** S

$S \rightarrow$ **id :=** E

{ *p* := *lookup*(*top*(*tblptr*), **id**.name);
if *p* = nil **then**
error()
else if *p*.level = 0 **then** // *global variable*
emit(**id**.place ':=' *E*.place)
else // *local variable in subroutine frame*
emit(fp[*p*.offset] ':=' *E*.place) }

Globals

`s`	(0)
`x`	(8)
`y`	(12)

Subroutine frame

fp[0]=	`a`	(0)
fp[4]=	`b`	(4)
fp[8]=	`t`	(8)
…		

Syntax-Directed Translation of Expressions in Scope

$E \rightarrow E_1$ **+** 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*(*top*(*tblptr*), **id**.name);
if p = nil **then** *error*()
else if p.level = 0 **then** *// global variable*
E.place := **id**.place
else *// local variable in frame*
E.place := fp[p.offset] }

Syntax-Directed Translation of Expressions in Scope (cont'd)

$E \rightarrow E_1$ **^** { E.place := *newtemp*();
  emit(E.place ':=' '*' E_1.place) }

$E \rightarrow$ **&** E_1 { E.place := *newtemp*();
  emit(E.place ':=' '&' E_1.place) }

$E \rightarrow \mathbf{id}_1$ **.** $\mathbf{id}_2$ { p := *lookup*(*top*(*tblptr*), $\mathbf{id}_1$.name);
  if p = nil **or** p.type != Trec **then** *error*()
  else
    q := *lookup*(p.type.table, $\mathbf{id}_2$.name);
    if q = nil **then** error()
    else if p.level = 0 **then** // *global variable*
      E.place := $\mathbf{id}_1$.place[q.offset]
    else // *local variable in frame*
      E.place := fp[p.offset+q.offset] }

Advanced Intermediate Code Generation Techniques

- Reusing temporary names
- Addressing array elements
- Translating logical and relational expressions
- Translating short-circuit Boolean expressions and flow-of-control statements with backpatching lists
- Translating procedure calls

Reusing Temporary Names

$E_1 + E_2$ $\xrightarrow{\text{generate}}$

Evaluate E_1 into **`t1`**
Evaluate E_2 into **`t2`**
`t3 := t1 + t2`

If **`t1`** no longer used, can reuse **`t1`** instead of using new temp **`t3`**

Modify *newtemp*() to use a "stack":
Keep a counter c, initialized to 0
Decrement counter on each use of a **`$`***i* in a three-address statement
newtemp() returns temporary **`$`***c*++ (that is, *c* is post incremented)

Syntax-Directed Translation into Three-Address Code (cont'd)

Productions	Semantic rules
$S \rightarrow$ **id** := E	S.code := E.code || *gen*(**id**.place ':=' E.place); S.begin := S.after := nil
$S \rightarrow$ **while** E **do** S_1	(*see next slide)*
$E \rightarrow E_1 + E_2$	E.place := *newtemp*(); E.code := E_1.code || E_2.code || *gen*(E.place ':=' E_1.place '+' E_2.place)
$E \rightarrow E_1$ * E_2	E.place := *newtemp*(); E.code := E_1.code || E_2.code || *gen*(E.place ':=' E_1.place '*' E_2.place)
$E \rightarrow$ **-** E_1	E.place := *newtemp*(); E.code := E_1.code || *gen*(E.place ':=' 'uminus' E_1.place)
$E \rightarrow$ **(** E_1 **)**	E.place := E_1.place E.code := E_1.code
$E \rightarrow$ **id**	E.place := **id**.name E.code := ''
$E \rightarrow$ **num**	E.place := *newtemp*(); E.code := *gen*(E.place ':=' **num**.value)

Reusing Temporary Names (cont’d)

x := a * b + c * d - e * f

a := b * c + (-1 + 2) * 3

$0=b*c	**0-0-1**
$1=1	**1-1-2**
$1= - $1	**2-1-2**
$2=2	**2-2-3**
$1=$1+$2	**3-1-2**
$2=3	**2-2-3**
$1=$1*$2	**3-1-2**
$0=$0+$1	**2-0-1**
a:=$0	**1-0-0**

Statement	$c \rightarrow c_{decr} \rightarrow c_{incr}$
	0
`$0 := a * b`	$0 \rightarrow 0 \rightarrow 1$
`$1 := c * d`	$1 \rightarrow 1 \rightarrow 2$
`$0 := $0 + $1`	$2 \rightarrow 0 \rightarrow 1$
`$1 := e * f`	$1 \rightarrow 1 \rightarrow 2$
`$0 := $0 - $1`	$2 \rightarrow 0 \rightarrow 1$
`x := $0`	$1 \rightarrow 0 \rightarrow 0$

Addressing Array Elements: One-Dimensional Arrays

```
A : array [10..20] of integer;
```

`… := A[i]` $= base_{\mathtt{A}} + (i - low) * w$

$= i * w + c$

where $c = base_{\mathtt{A}} - low * w$

with $low = 10$; $w = 4$ (= type width)

`t1 := c      // c =` $base_{\mathtt{A}} - 10 * 4$

```
t2 := i * 4
t3 := t1[t2]
…  := t3
```

Addressing Array Elements: Multi-Dimensional Arrays

`A : array [1..2,1..3] of integer;`

$low_1 = 1, low_2 = 1, n_1 = 2, n_2 = 3, w = 4$

$base_{\mathtt{A}}$	`A[1,1]`
	`A[1,2]`
	`A[1,3]`
	`A[2,1]`
	`A[2,2]`
	`A[2,3]`

Row-major

$base_{\mathtt{A}}$	`A[1,1]`
	`A[2,1]`
	`A[1,2]`
	`A[2,2]`
	`A[1,3]`
	`A[2,3]`

Column-major

Addressing Array Elements: Multi-Dimensional Arrays

`A : array [1..2,1..3] of integer;` (Row-major)

base l1 n1 l2 n2 w=4

`... := A[i,j]`

$= base_{\mathtt{A}} + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$

$= ((i_1 * n_2) + i_2) * w + c$

where $c = base_{\mathtt{A}} - ((low_1 * n_2) + low_2) * w$

with $low_1 = 1; low_2 = 1; n_2 = 3; w = 4$

`t1 := i * 3`

`t1 := t1 + j`

`t2 := c` **`// c =`** $base_{\mathtt{A}} - (1 * 3 + 1) * 4$

`t3 := t1 * 4`

`t4 := t2[t3]` **`// t4 := mem[`**$((\mathtt{i} * 3) + \mathtt{j}) * 4 + c$**`]`**

`... := t4`

Addressing Array Elements: Grammar

$S \rightarrow L$ **:=** E
$E \rightarrow E + E$
  | **(** E **)**
  | L
  | **num**
$L \rightarrow Elist$ **]**
  | **id**
$Elist \rightarrow Elist$ **,** E
  | **id [** E

Synthesized attributes:

E.place	name of temp holding value of *E*
Elist.array	array name
Elist.place	name of temp holding index value
Elist.ndim	number of array dimensions
L.place	lvalue (=name of temp)
L.offset	index into array (=name of temp) **null** indicates non-array simple **id**

a:=b
a:=b[2]
a:=b[3,4]
c[2,4]:=a[3,4]

Addressing Array Elements

$$
\begin{array}{ll}
S \rightarrow L \ \mathbf{:=} \ E & \{\ \mathbf{if}\ L.\text{offset} = \mathbf{null\ then} \\
 & \quad \mathit{emit}(L.\text{place}\ \text{':='}\ E.\text{place}) \\
 & \ \mathbf{else} \\
 & \quad \mathit{emit}(L.\text{place}[L.\text{offset}]\ \text{':='}\ E.\text{place})\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.\text{place} := \mathit{newtemp}(); \\
 & \ \mathit{emit}(E.\text{place}\ \text{':='}\ E_1.\text{place}\ \text{'+'}\ E_2.\text{place})\ \} \\
E \rightarrow \mathbf{(}\ E_1\ \mathbf{)} & \{\ E.\text{place} := E_1.\text{place}\ \} \\
E \rightarrow L & \{\ \mathbf{if}\ L.\text{offset} = \mathbf{null\ then} \\
 & \quad E.\text{place} := L.\text{place} \\
 & \ \mathbf{else} \\
 & \quad E.\text{place} := \mathit{newtemp}(); \\
 & \quad \mathit{emit}(E.\text{place}\ \text{':='}\ L.\text{place}[L.\text{offset}])\ \}
\end{array}
$$

Addressing Array Elements

$L \rightarrow Elist$ **]** { L.place := $newtemp$();
L.offset := $newtemp$();
$emit$(L.place ':=' c($Elist$.array));
$emit$(L.offset ':=' $Elist$.place '*' $width$($Elist$.array)) }

$L \rightarrow$ **id** { L.place := **id**.place;
L.offset := **null** }

$Elist \rightarrow Elist_1$ **,** E
{ t := $newtemp$(); m := $Elist_1$.ndim + 1;
$emit$(t ':=' $Elist_1$.place '*' $limit$($Elist_1$.array, m));
$emit$(t ':=' t '+' E.place);
$Elist$.array := $Elist_1$.array; $Elist$.place := t;
$Elist$.ndim := m }

$Elist \rightarrow$ **id [** E { $Elist$.array := **id**.place; $Elist$.place := E.place;
$Elist$.ndim := 1 }

Translating Logical and Relational Expressions

a or b and not c

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

a < b

```
    if a < b goto L1
    t1 := 0
    goto L2
L1: t1 := 1
L2:
```

Translating Short-Circuit Expressions Using Backpatching

$E \to E$ **or** ME
$\quad | \; E$ **and** ME
$\quad |$ **not** E
$\quad |$ **(** E **)**
$\quad |$ **id relop id**
$\quad |$ **true**
$\quad |$ **false**
$M \to \varepsilon$

Synthesized attributes:

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 concatenates 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 $\}$

Backpatching with Lists: Translation Scheme (cont'd)

$E \rightarrow$ **id**$_1$ **relop id**$_2$

{
E.truelist := *makelist*(nextquad());
E.falselist := *makelist*(nextquad());
emit('`if`' **id**$_1$.place **relop**.op **id**$_2$.place '`goto _`');
emit('`goto _`') }

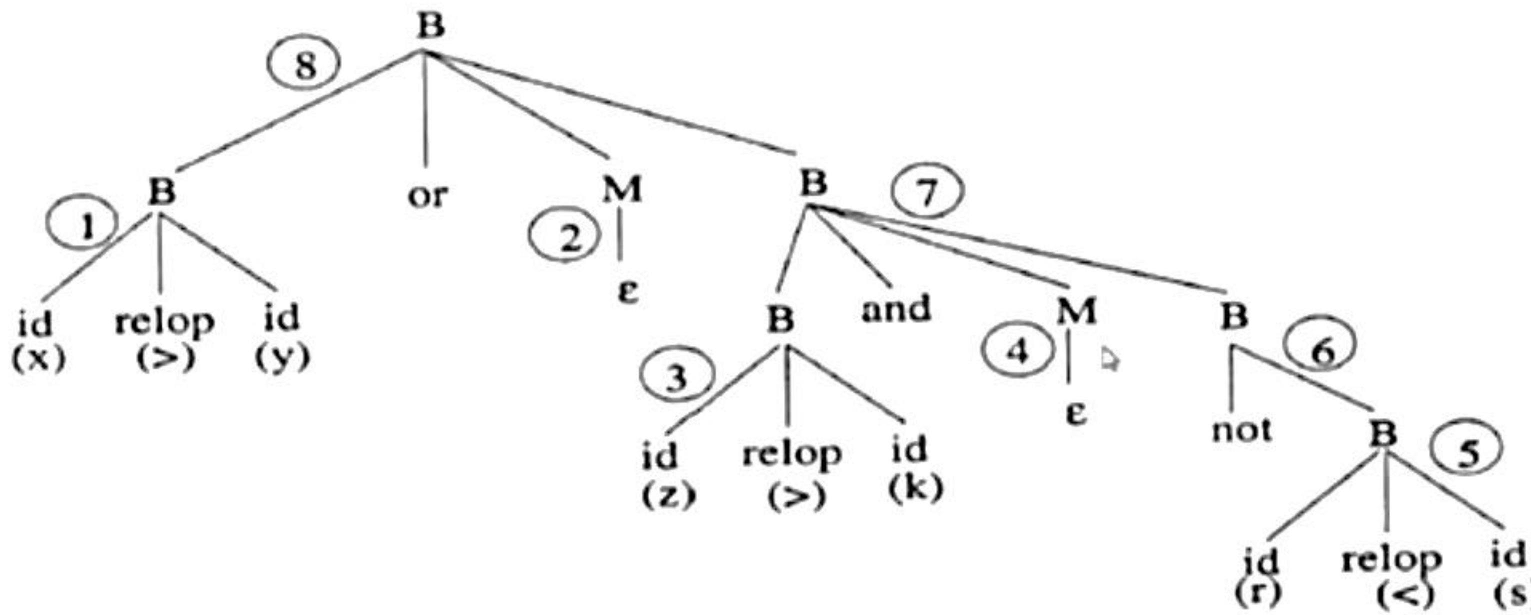
$E \rightarrow$ **true** { *E*.truelist := *makelist*(*nextquad*());
E.falselist := nil;
emit('`goto _`') }

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

x **and** y **or** z>k

T F

a and (b or c and not d)



Reduction	Action	Code generated
1	B.truelist = {1} B.falselist = {2}	1: if x > y goto ... 2: goto ...
2	M.quad = 3	
3	B.truelist = {3}, B.falselist = {4}	3: if z > k goto ... 4: goto ...
4	M.quad = 5	
5	B.truelist = {5} B.falselist = {6}	5: if r > s goto ... 6: goto ...
6	B.truelist = {6}, B.falselist = {5}	
7	Backpatches list {3} with 5 {3,6}, {5}	3: if z > k goto 5
8	Backpatches list {2} with 3 B.truelist = {1,6}, B.falselist = {4,5}	2: goto 3

x **and** M y **or** M z>k

T F

E1.truelist{1} 1: goto _3_

E1.falselist{2} 2: goto _5_

M.quad{3}

E2.truelist{3} 3: goto _TRUE_

E2.falselist{4} 4: goto _5_

Backpatch {1} with 3

E1.tl{3} E1.fl{2,4}

M.quad{5}

E2.tl{5} 5: if z > k goto _TRUE_

E2.fl{6} 6: Goto _FALSE_

Backpatch {2,4} with 5

TRUE{3,5} FALSE{6}

FALSE{4}

Flow-of-Control Statements and Backpatching: Grammar

$S \rightarrow$ **if** E **then** S
  | **if** E **then** S **else** S
  | **while** E **do** S
  | **begin** L **end**
  | A
$L \rightarrow L$ **;** S
  | S

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_5 …

Jumps out of S_1

```
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

$S \to A$ { S.nextlist := nil }

$S \to$ **begin** L **end**
{ S.nextlist := L.nextlist }

$S \to$ **if** E **then** $M\ S_1$
{ *backpatch*(E.truelist, M.quad);
S.nextlist := *merge*(E.falselist, S_1.nextlist) }

$L \to L_1$ **;** $M\ S$ { *backpatch*(L_1.nextlist, M.quad);
L.nextlist := S.nextlist; }

$L \to S$ { L.nextlist := S.nextlist; }

$M \to \varepsilon$ { M.quad := *nextquad*() }

$A \to \ldots$ *Non-compound statements, e.g. assignment, function call*

Flow-of-Control Statements and Backpatching (cont'd)

$S \rightarrow$ **if** E **then** $M_1\ S_1\ N$ **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)) }

$S \rightarrow$ **while** $M_1\ E$ **do** $M_2\ S_1$

{ *backpatch*(S_1,nextlist, M_1.quad);
backpatch(E.truelist, M_2.quad);
S.nextlist := E.falselist;
emit('`goto` M_1.quad') }

$N \rightarrow \varepsilon$ { N.nextlist := *makelist*(*nextquad*());
emit('`goto _`') }

Translating Procedure Calls

$$S \rightarrow \textbf{call id (} \; Elist \; \textbf{)}$$
$$Elist \rightarrow Elist \; \textbf{,} \; E$$
$$\mid E$$

call foo(a+1, b, 7)

```
t1 := 1
t1 := a + t1
t2 := 7
param t1
param b
param t2
call foo 3
```

Translating Procedure Calls

$S \rightarrow$ **call id (** *Elist* **)**	{ **for** each item p on *queue* **do**
	    emit('`param`' p);
	  emit('`call`' **id**.place $\lvert queue \rvert$) }

$Elist \rightarrow Elist$ **,** E	{ append E.place to the end of *queue* }
$Elist \rightarrow E$	{ initialize *queue* to contain only E.place }