Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-14/

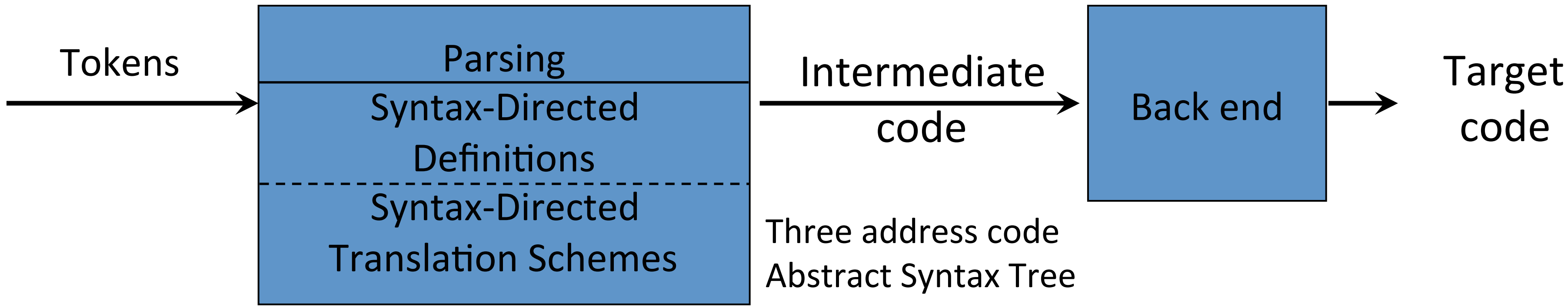
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 13

- Intermediate-Code Generation Techniques
  - Array elements
  - Booleans and logical conditions
  - Function/procedure calls

Intermediate Code Generation (II)

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



Recap (last lecture)

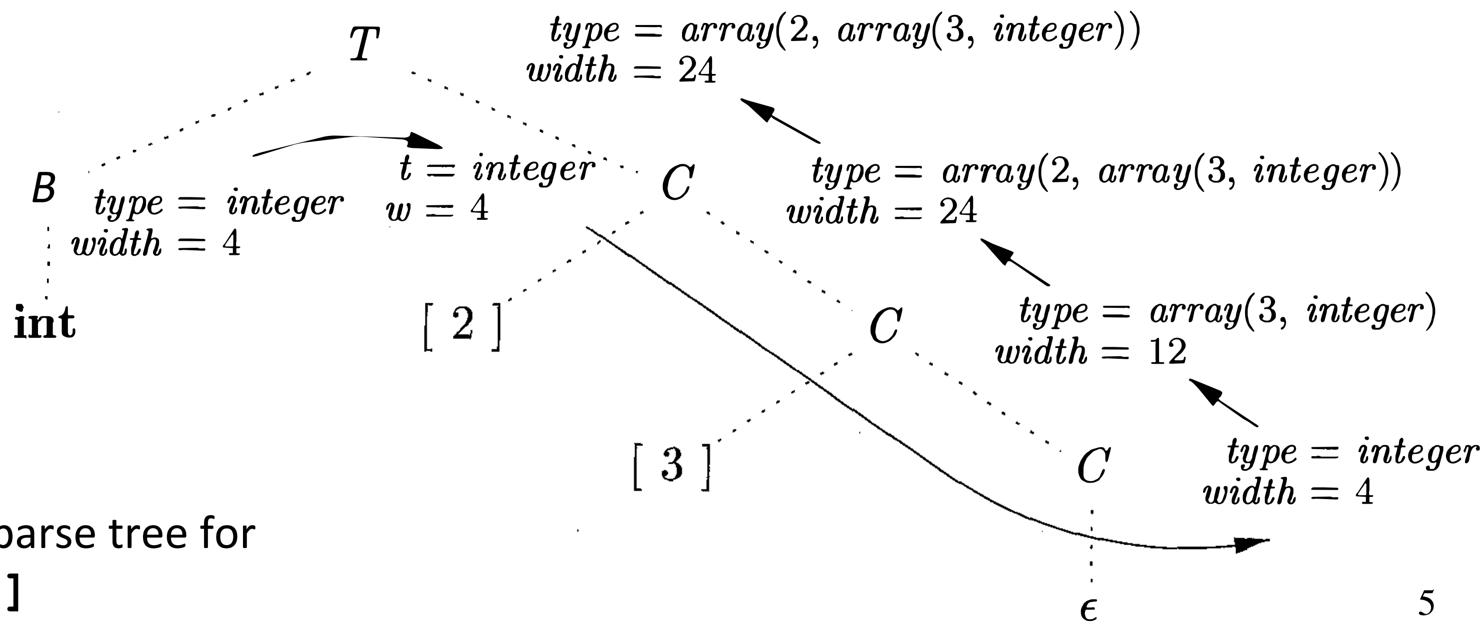
- Intermediate representations
- Three address statements and their implementations
- Syntax-directed translation to three address statements
  - Expressions and statements
- Handling local names and scopes with symbol tables
- Syntax-directed translation of
  - Declarations in scope
  - Expressions in scope
  - Statements in scope

Summary

- Multi-dimensional arrays
  - Translation scheme for computing type and width
  - Generation of three address statement for addressing array elements
- Translating logical and relational expressions
- Translating short-circuit Boolean expressions and flow-of-control statements with backpatching lists
- Translating procedure calls

Decl. of Multidim. Arrays: SDTS for type/width

T →	*B*	{ *t* = *B.type*; *w* = *B.width*; }
	C	{ *T.type* = *C.type*; *T.width* = *C.width* }
B →	**int**	{ *B.type* = 'integer'; *B.width* = 4; }
B →	**float**	{ *B.type* = 'float'; *B.width* = 8; }
C →	ε	{ *C.type* = *t*; *C.width* = *w*; }
C →	[**num**] C_1	{ *C.type* = *array*(**num**.*value*, C_1.*type*); *C.width* = **num**.*value* * C_1.*width*; }



Annotated parse tree for
`int[2][3]`

Addressing Array Elements: One-Dimensional Arrays

- Assuming that elements are stored in adjacent cells:

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

low (10) *high* (20) *Type's size* (w)

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

- If *base, low* and *w* are known at compile time:

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

Example with $low = 10$; $w = 4$

…

t1 := c //c = $base_{\mathtt{A}}$ - 10 * 4, can be stored in the symbol table

t2 := i * 4

t3 := t1[t2]

… := t3

Addressing Array Elements: Multi-Dimensional Arrays

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

$$low_1 = 1, low_2 = 1,$$
$$n_1 = high_1 - low_1 + 1 = 2, \quad n_2 = 3,$$
$$w = 4 \text{ (element type size)}$$

Row-major (as in C)
$base_{\mathtt{A}}$ → `A[1][1]`
`A[1][2]`
`A[1][3]`
`A[2][1]`
`A[2][2]`
`A[2][3]`

Column-major (as in Fortran)
$base_{\mathtt{A}}$ → `A[1][1]`
`A[2][1]`
`A[1][2]`
`A[2][2]`
`A[1][3]`
`A[2][3]`

(as in C) Row-major Column-major (as in Fortran)

Addressing Array Elements: Multi-Dimensional Arrays

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

… **`:= A[i][j]`** $= base_{\mathbf{A}} + ((i - low_1) * n_2 + j - low_2) * w$

$= ((i * n_2) + j) * w + c$

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

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

```
t1  := i * 3
t1  := t1 + j
t2  := c           // c = baseA - (1 * 3 + 1) * 4
t3  := t1 * 4
t4  := t2[t3]      // base t2, offset t3
…   := t4
```

Addressing Array Elements: Grammar

Grammar:

$S \rightarrow$ **id** **=** E ;
  | L **=** E ;
$E \rightarrow E$ **+** E
  | **id**
  | L
$L \rightarrow$ **id** [E]
  | L [E]

Synthesized attributes:

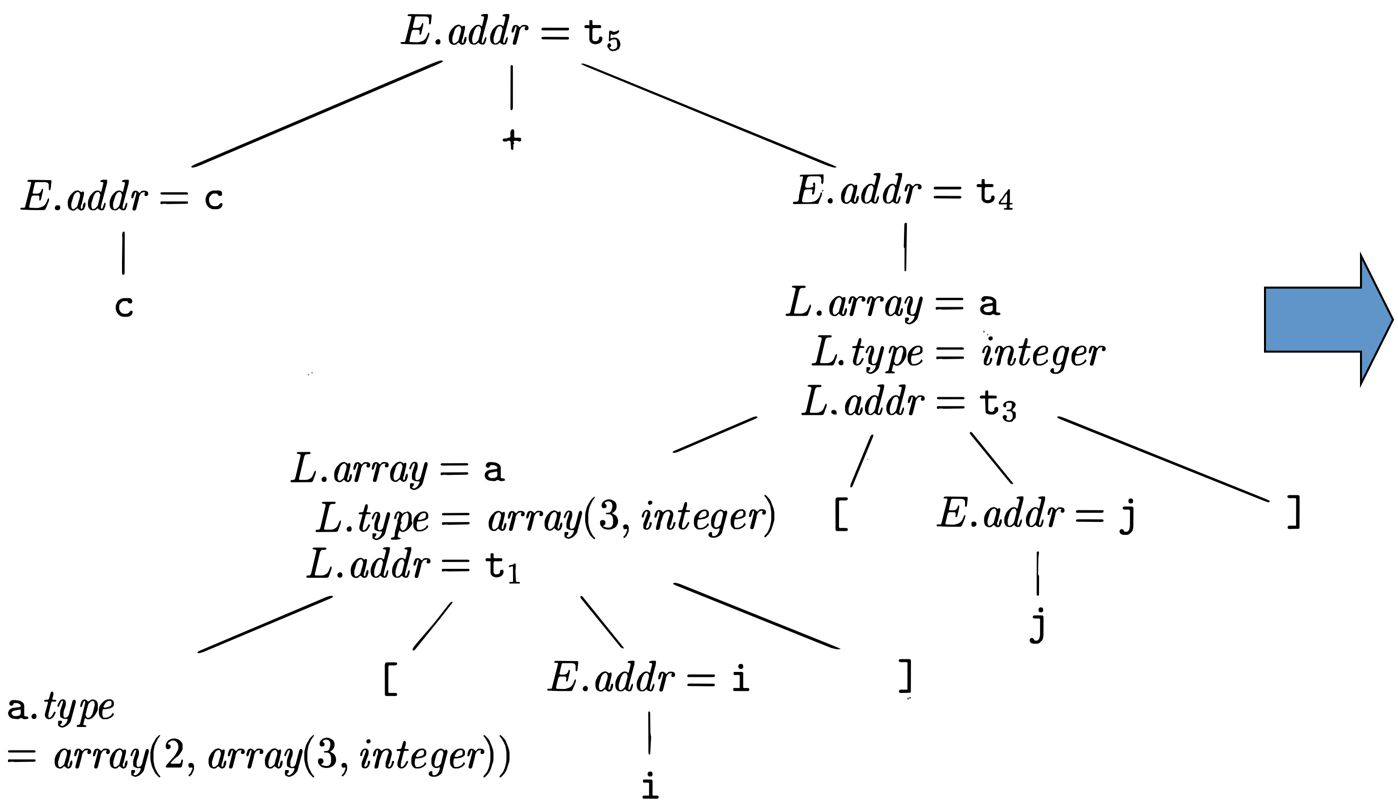
E.addr	name of temp holding value of *E*
L.addr	temporary to compute offset
L.array	pointer to symbol table entry for the array name
L.array.base	base address
L.array.type	type of the array, eg. *array(2, array(3,int))*
L.array.type.elem	type of array elements, eg. *array(3,int)*
L.type	type of the subarray generated by *L*
L.type.width	memory allocated for data of type *L.type*

- Nonterminal *L* generates an array name followed by a sequence of indexes, like
  `a[i][j][k]`
- *L* can appear both as left- and right-value

Addressing array elements: generating three address statements

$S \rightarrow$ **id** **=** *E* ;	{ *gen(top.get(***id***.lexeme)* '=' *E.addr);* }	// no array
| *L* **=** *E* ;	{ *gen(L.array.base* '[' *L.addr* ']' '=' *E.addr);* }	// address = base + offset
$E \rightarrow E_1$ **+** E_2	{ *E.addr* = **new** *Temp()*; *gen(E.addr* '=' E_1*.addr* '+' E_2*.addr);* }	// similarly for *, -, …
| **id**	{ *E.addr = top.get(***id***.lexeme);* }	
| *L*	{ *E.addr* = **new** *Temp();* *gen(E.addr* '=' *L.array.base* '[' *L.addr* ']'*);* }	// address = base + offset
$L \rightarrow$ **id** [*E*]	{ *L.array = top.get(***id***.lexeme);* *L.type = L.array.type.elem;* *L.addr* = **new** *Temp();* *gen(L.addr* '=' *E.addr* '*' *L.type.width);* }	
| L_1 [*E*]	{ *L.array=*L_1*.array;* *L.type =* L_1*.type.elem;* *t* = **new** *Temp();* *L.addr=* **new** *Temp();* *gen(t* '=' *E.addr* '*' *L.type.width);* *gen(L.addr* '=' L_1*.addr* '+' *t);* }	

Example - generating intermediate code for access to array: `c + a[i][j]`



t_1 = i * 12
t_2 = j * 4
t_3 = t_1 + t_2
t_4 = a [t_3]
t_5 = c + t_4

Translating Logical and Relational Expressions

Boolean expressions intended to represent values:

a or b and not c

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

Boolean expressions used to alter the control flow:

a < b

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

Short-Circuit Code

- The boolean operators &&, || and ! are translated into jumps.
- Example:

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

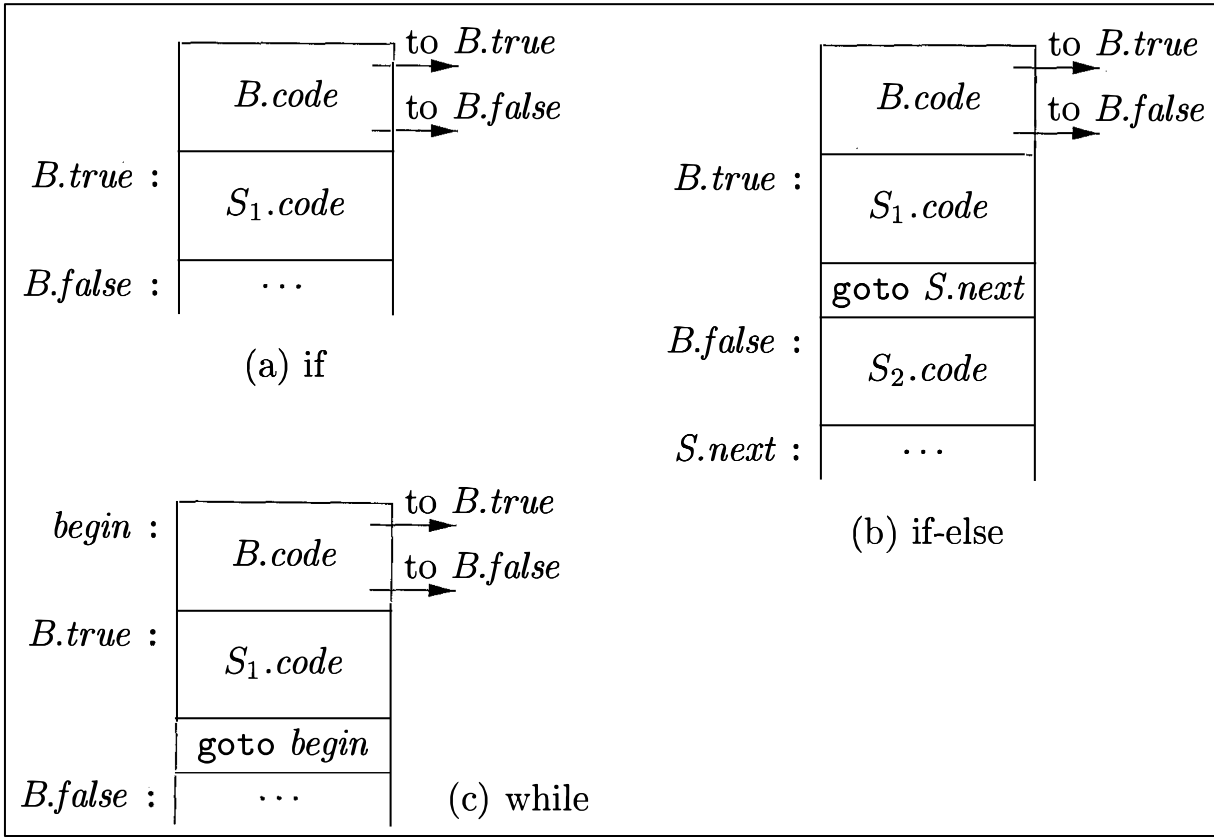
may be translated into:

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

Translating Flow-of-control Statements

$$S \rightarrow \textbf{if} (B) S_1$$
$$S \rightarrow \textbf{if} (B) S_1 \textbf{ else } S_2$$
$$S \rightarrow \textbf{while} (B) S_1$$

Synthesized Attributes:
S.code, B.Code
Inherited Attributes:
labels for jumps:
B.true, B.false, S.next



PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\|\; label(B.true) \;\|\|\; S_1.code$
$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\|\| \; label(B.true) \;\|\|\; S_1.code$ $\|\| \; gen(\texttt{'goto'}\ S.next)$ $\|\| \; label(B.false) \;\|\|\; S_2.code$
$S \rightarrow \textbf{while}\ (\ B\ )\ S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \;\|\|\; B.code$ $\|\| \; label(B.true) \;\|\|\; S_1.code$ $\|\| \; gen(\texttt{'goto'}\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \;\|\|\; label(S_1.next) \;\|\|\; S_2.code$

Not relevant for control flow

Inherited Attributes

Translation of Boolean Expressions

Production	Semantic Rules
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \ ! \ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \ \textbf{rel} \ E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}' \ E_1.addr \ \textbf{rel}.op \ E_2.addr \ '\texttt{goto}' \ B.true)$ $\mid\mid gen('\texttt{goto}' \ B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen('\texttt{goto}' \ B.true)$
$B \rightarrow \textbf{false}$	$B.code = gen('\texttt{goto}' \ B.false)$

Inherited Attributes

Example

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

is translated into:

```
    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:
```

By removing several redundant jumps we can obtain the equivalent:

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

Translating Short-Circuit Expressions Using Backpatching

Idea: avoid using inherited attributes by generating partial code. Addresses for jumps will be inserted when known.

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

M : marker nonterminal

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 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 }

Backpatching with Lists: Translation Scheme (cont'd)

$E \rightarrow$ **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 _`**') }

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

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

Flow-of-Control Statements and Backpatching: Grammar

S → **if** *E* **then** *S*
  | **if** *E* **then** *S* **else** *S*
  | **while** *E* **do** *S*
  | **begin** *L* **end**
  | *A*
L → *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

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

$S \rightarrow$ **begin** L **end**

{ S.nextlist := L.nextlist }

$S \rightarrow$ **if** E **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*() }

$A \rightarrow$ … *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 _`**') }