Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-15/

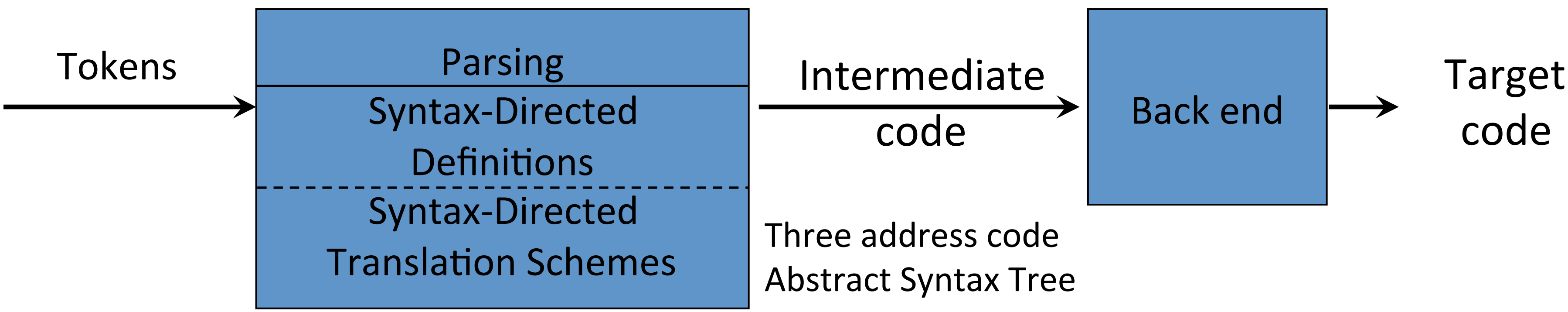
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 20

- Intermediate-Code Generation Techniques
  - Translation in Scope
  - Booleans and logical conditions
  - Backpatching

Intermediate Code Generation (II)

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



Summary

- Handling local names and scopes with symbol tables
- Syntax-directed translation of
  - Declarations in scope
  - Expressions in scope
  - Statements in scope
- Translating logical and relational expressions
- Translating short-circuit Boolean expressions and flow-of-control statements with backpatching lists

Names and Scopes

- The three-address code generated by the syntax-directed definitions shown is simplistic
- 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 records (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)

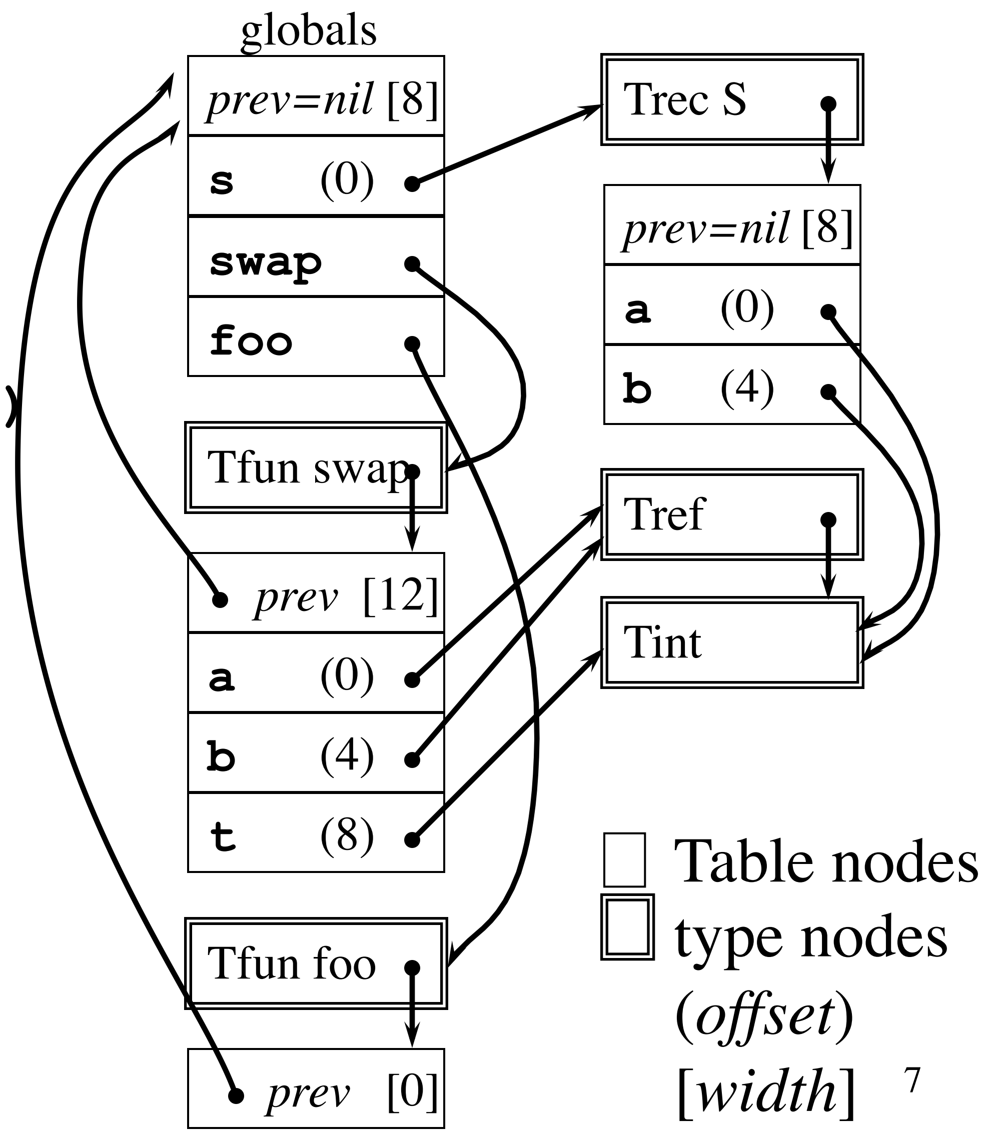
The *width* of the frame is 12

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 (empty) 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: Grammar and Attributes

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 (ex.: '*integer*', *array*(2, '*real*'), *pointer*(*record*(*Table*)), …)
***T*.width** storage width of type (bytes)
***E*.place** name of temp holding value of *E*

Global data to implement scoping:

LeBlanc&Cook stack of tables

tblptr stack of pointers to tables
offset stack of offset values

Syntax-Directed Translation of Declarations in Scope

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

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

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

$D \rightarrow D_1$ **;** D_2

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

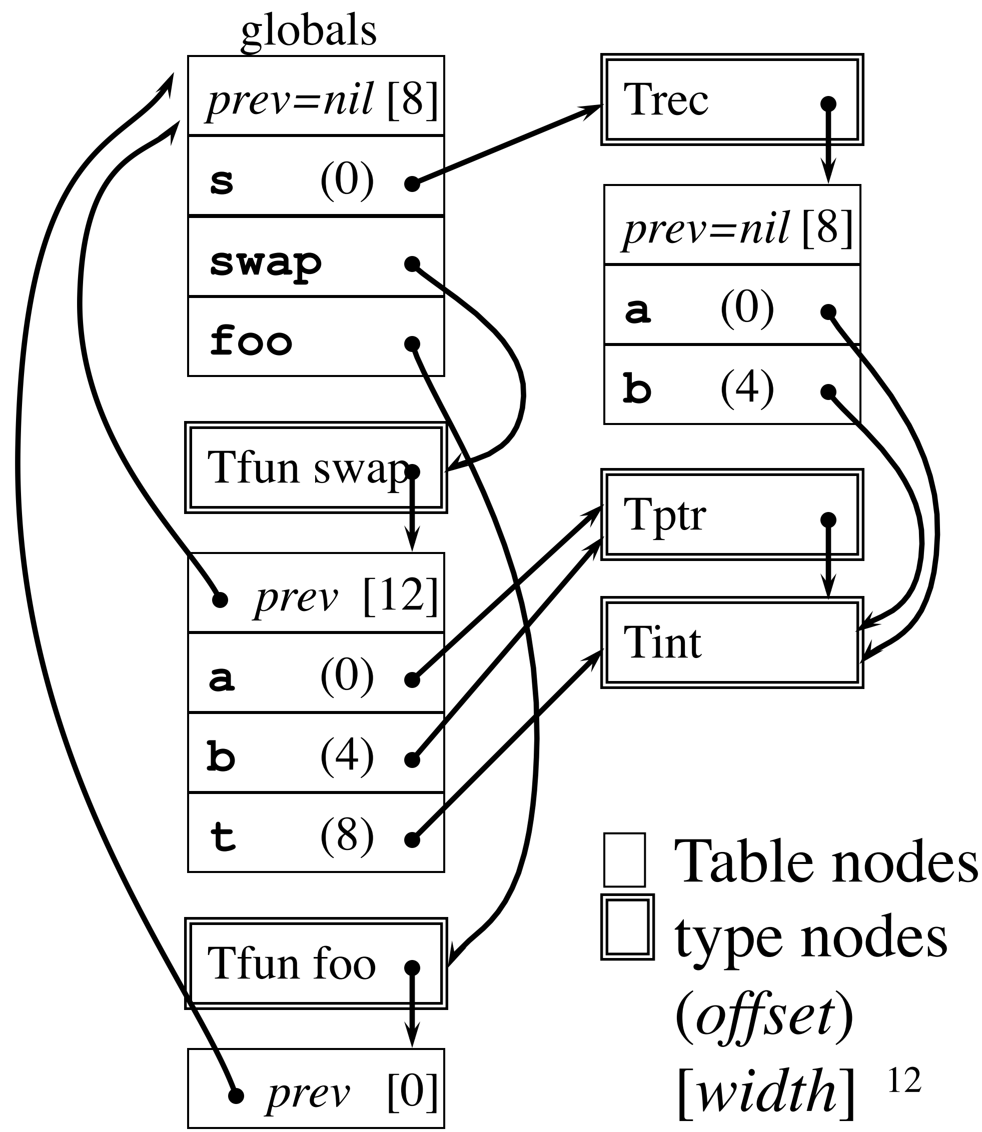
$T \rightarrow$ **integer** { T.type := '*integer*'; T.width := 4 }
$T \rightarrow$ **real** { T.type := '*real*'; T.width := 8 }
$T \rightarrow$ **array [num] of** T_1
{ T.type := *array*(**num**.val, T_1.type);
T.width := **num**.val * T_1.width }
$T \rightarrow$ **^** T_1
{ T.type := *pointer*(T_1.type); 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

```
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);
  …
}
```



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

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:

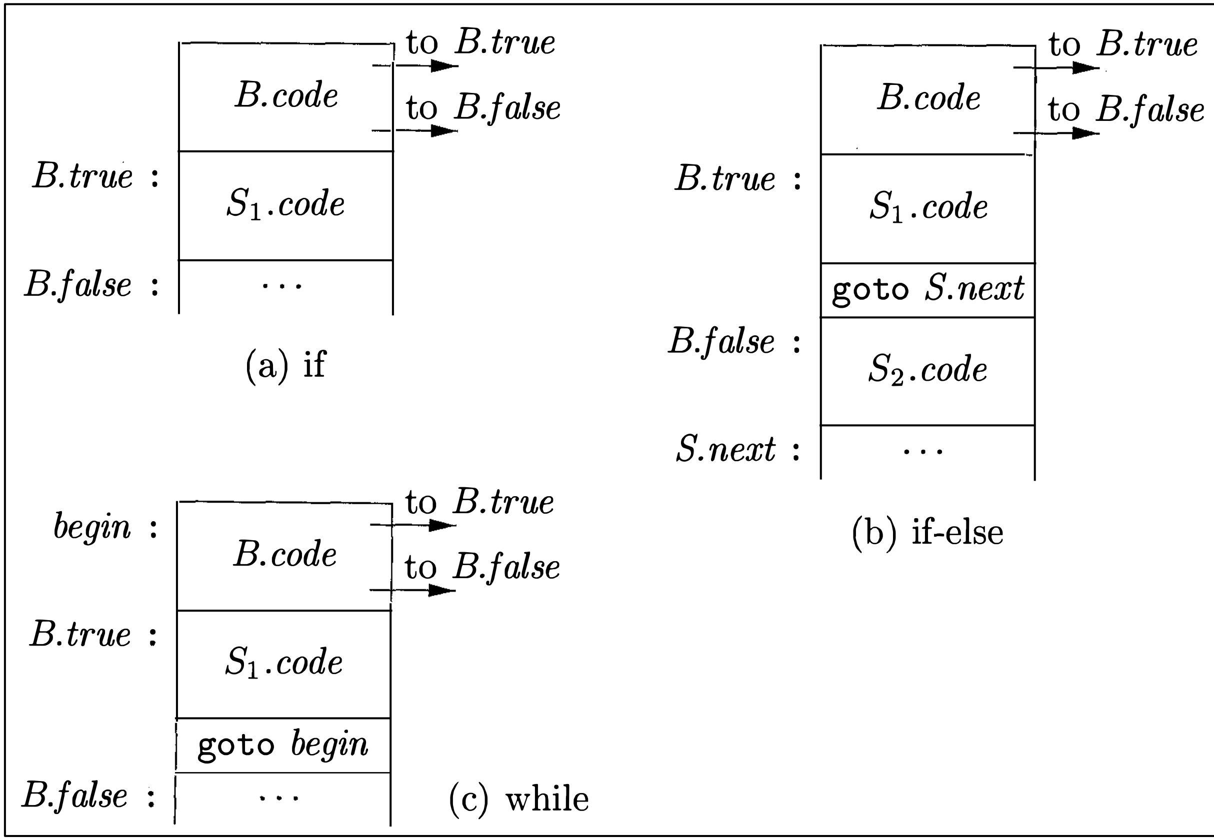
```
if ( x < 100 || x > 200 && x != y ) x = 0;
```

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	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow\ !\ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1$ **rel** E_2	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen($'if' $E_1.addr$ **rel**$.op$ $E_2.addr$ 'goto' $B.true)$ $\mid\mid gen($'goto' $B.false)$
$B \rightarrow$ **true**	$B.code = gen($'goto' $B.true)$
$B \rightarrow$ **false**	$B.code = gen($'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

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

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

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

$E \rightarrow$ **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 }

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

$S \rightarrow$ **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 …

```
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

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

The grammar is modified adding suitble marking non-terminals

$S \rightarrow A$ { 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*() }
$A \rightarrow \ldots$ *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 _`') }