Principles of Programming Languages

http://www.di.unipi.it/~andrea/Didattica/PLP-16/

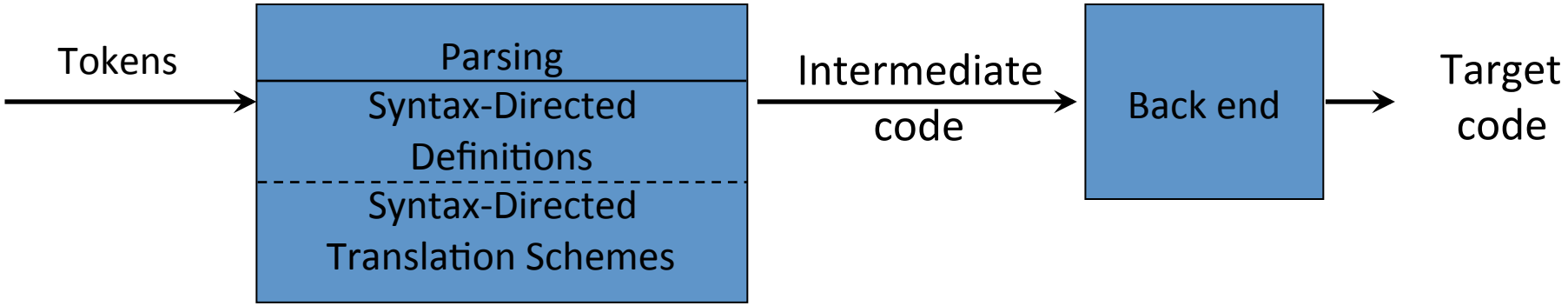
Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 11

- Intermediate-Code Generation
  - Intermediate representations
  - Syntax-directed translation to three address code

Intermediate Code Generation (II)

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



Summary

- Intermediate representations
- Three address statements and their implementations
- Syntax-directed translation to three address statements
  - Expressions and statements
  - Logical and relational expressions
  - Translating short-circuit Boolean expressions and flow-of-control statements with backpatching lists

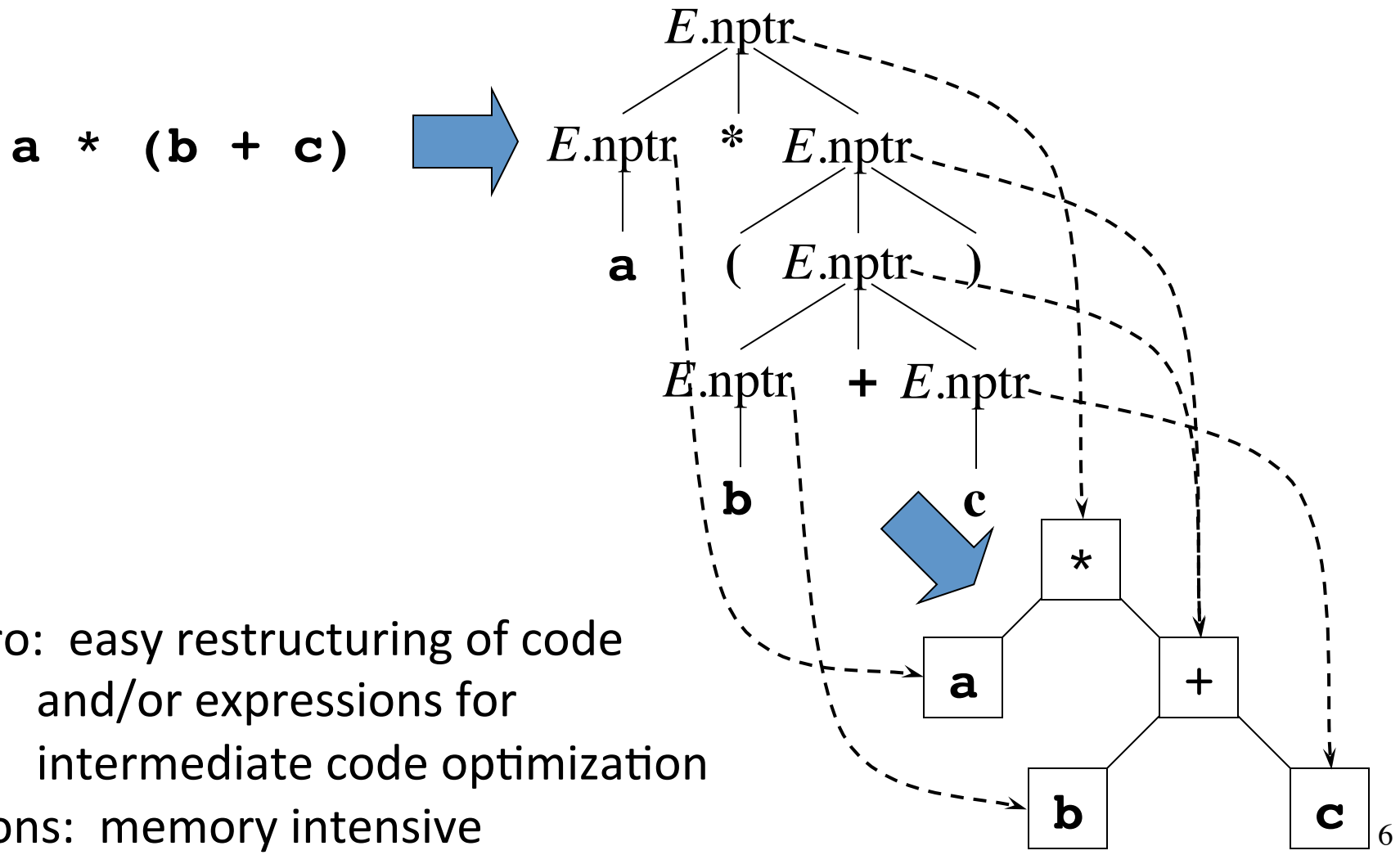
Intermediate Representations

- *Graphical representations* (e.g. AST and DAGs)
- *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)

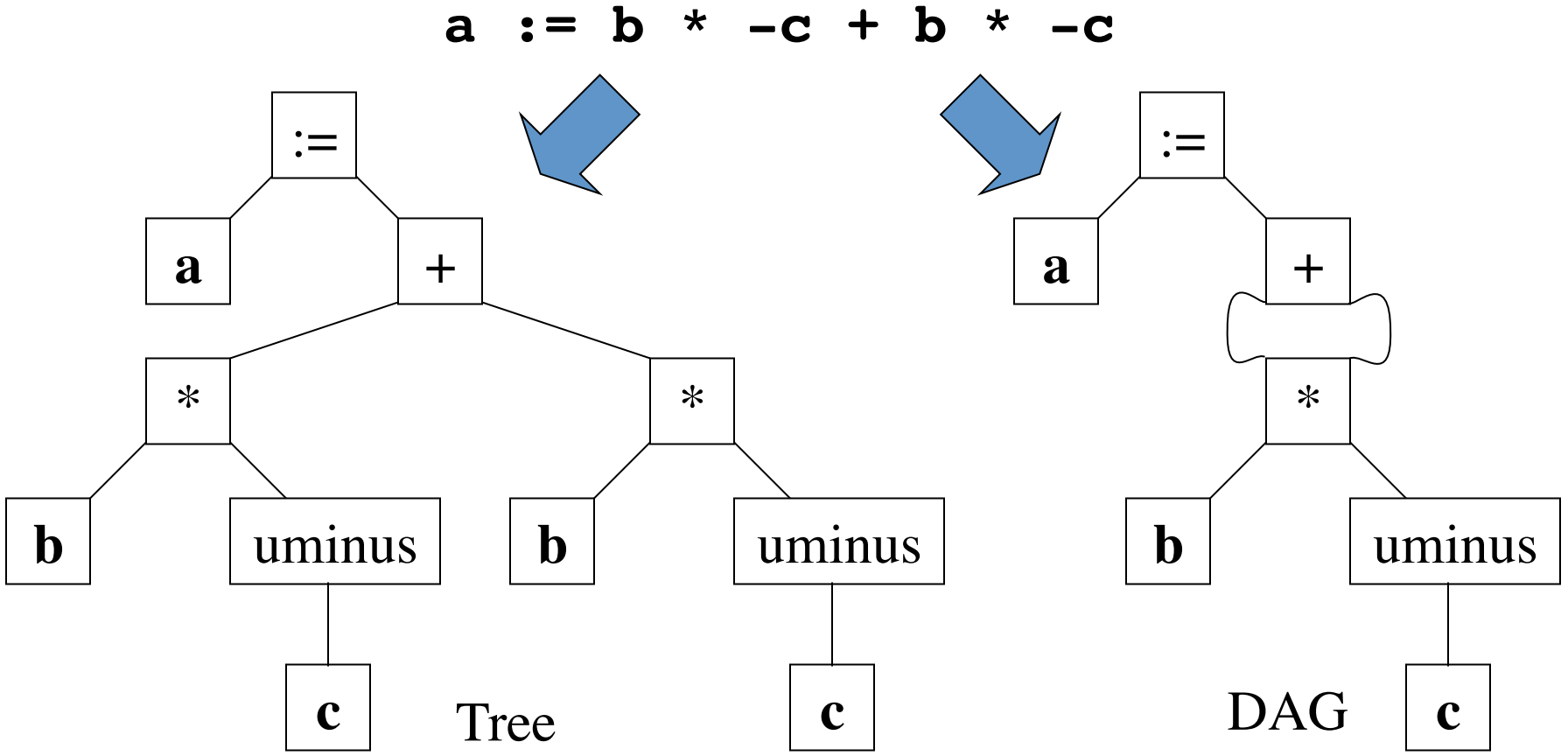
Abstract Syntax Trees



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

Cons: memory intensive

Abstract Syntax Trees versus DAGs



- Repeated subtrees are shared
- Implementation: *mkleaf* and *makenode are* redefined. They do not create a new node if it exists already.

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)

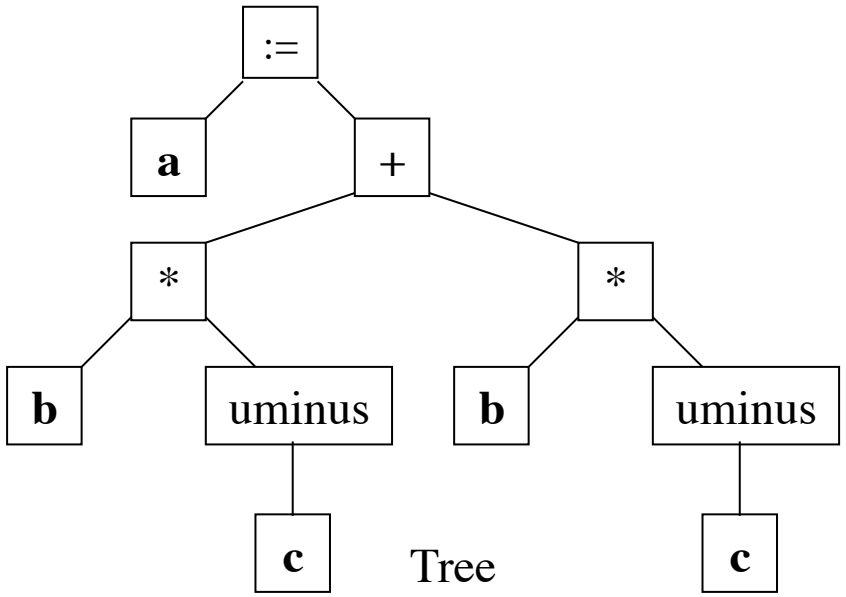
```
iload 2    // push b
iload 3    // push c
ineg       // uminus
imul       // *
iload 2    // push b
iload 3    // push c
ineg       // uminus
imul       // *
iadd       // +
istore 1   // store a
```

Three-Address Code (1)

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

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

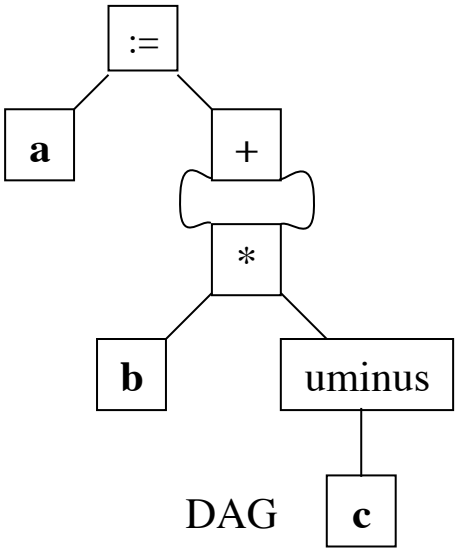
Linearized representation of a syntax tree



Tree

Three-Address Code (2)

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



DAG

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

Linearized representation of a syntax DAG

Three-Address Statements

"Addresses" are *names*, *constants* or *temporaries*

- 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
  (or y = **`call`** p, n) ; **`return`** y

Implementation of Three-Address Statements: Quads

Sample expression

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

Three-address code

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

#	*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

Quads (quadruples)

Pro: easy to rearrange code for global optimization

Cons: lots of temporaries

Implementation of Three-Address Statements: Triples

Sample expression

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

Three-address code

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

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

Triples

Pro: temporaries are implicit

Cons: difficult to rearrange code

Implementation of Three-Address Statements: **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

Syntax-Directed Translation into Three-Address Code

Productions

$S \rightarrow$ **id :=** E
| **while** E **do** S
| S ; 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.place ‘:=’ E_1.place ‘+’ E_2.place)

Code generation → `t3 := t1 + t2`

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)
$S \rightarrow S_1 ; S_2$	S.code := S_1.code ‖ S_2.code S.begin := S_1. begin S.after := S_2.after

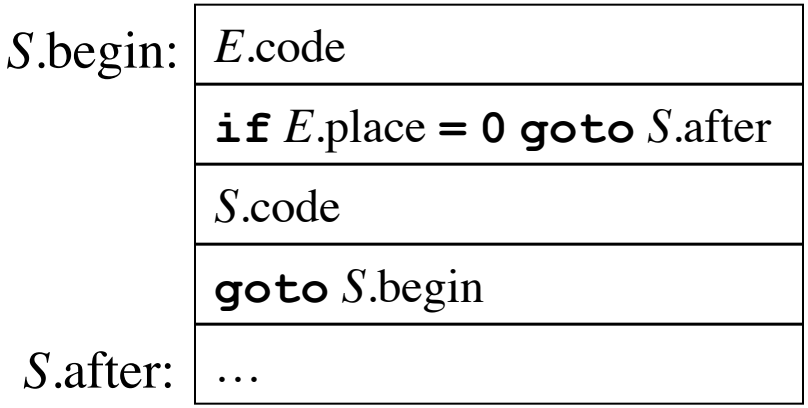
Syntax-Directed Translation into Three-Address Code (cont'd)

Production

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

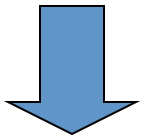
Semantic rule

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 ':')



Example (check it at home!)

```
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:
```

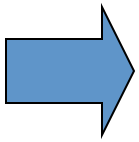
How does the code access names?

- 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
- We will see all this later in the course

Translating Logical and Relational Expressions

Boolean expressions intended to represent values:

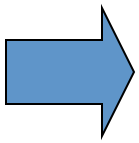
a or b and not c



```
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;
```

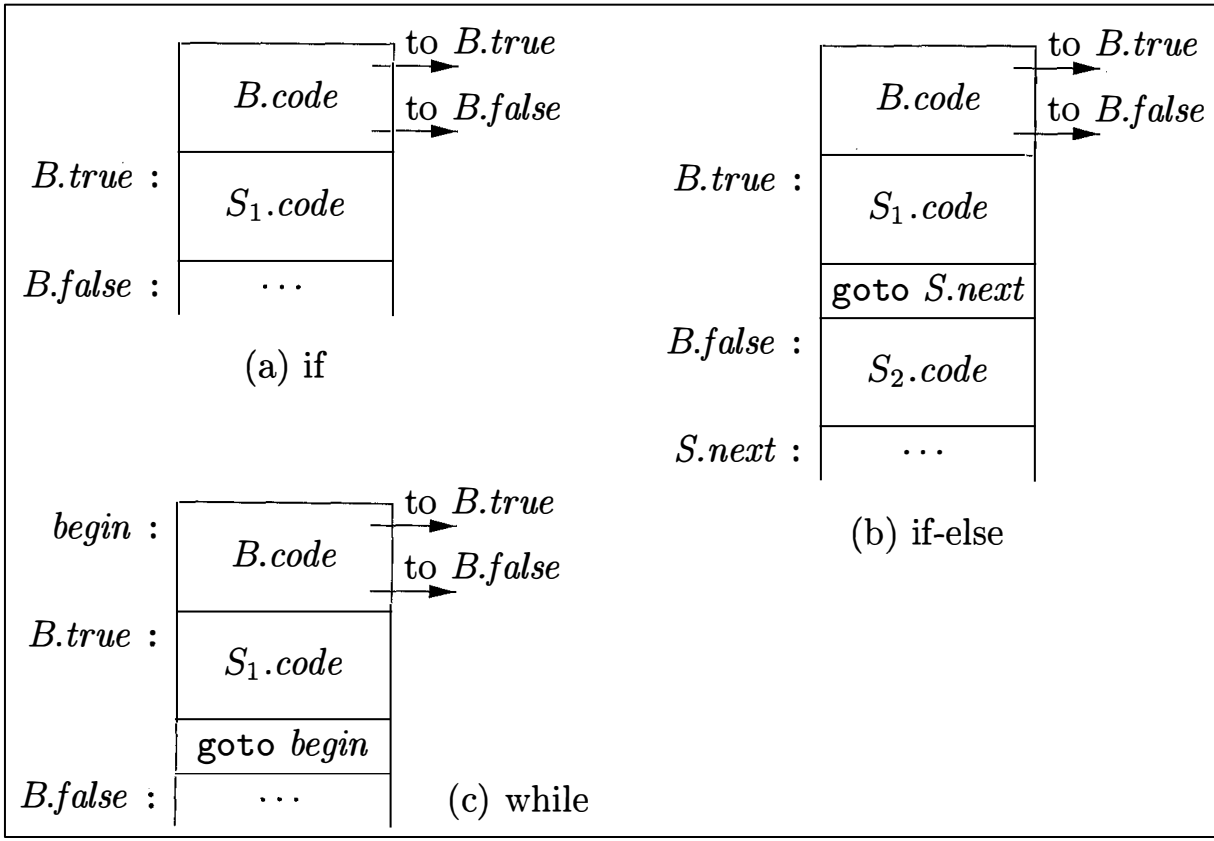
may be translated into:

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

Translating Flow-of-control Statements

$S \rightarrow$ **if** (B) S_1
$S \rightarrow$ **if** (B) S_1 **else** S_2
$S \rightarrow$ **while** (B) S_1

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



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

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	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow$ **true**	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow$ **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:

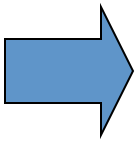
***E*.truelist**	backpatch list for jumps on true
***E*.falselist**	backpatch list for jumps on false
***M*.quad**	location of current three-address quad

Backpatch Operations with Lists

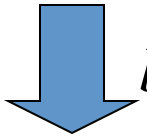
- ***makelist*(*i*)** creates a new list containing three-address location i, returns a pointer to the list
- ***merge*(p_1, p_2)** concatenates lists pointed to by p_1 and p_2, returns a pointer to the concatenated list
- ***backpatch*(*p*, *i*)** inserts i as the target label for each of the statements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

$M \rightarrow \varepsilon$ { 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

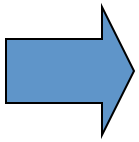
$L \rightarrow L$ **;** S
| S

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 _`') }

Summarizing…

- Translation from source code to intermediate representation can be performed in a syntax-directed way during parsing
- We considered translation of expressions and statements to three address code, with a simplified handling of names (for example: no definitions)
- More effective translation of boolean expressions for flow control require inherited attributes
- They can be avoided using **backpatching**