Compiler course

Chapter 6

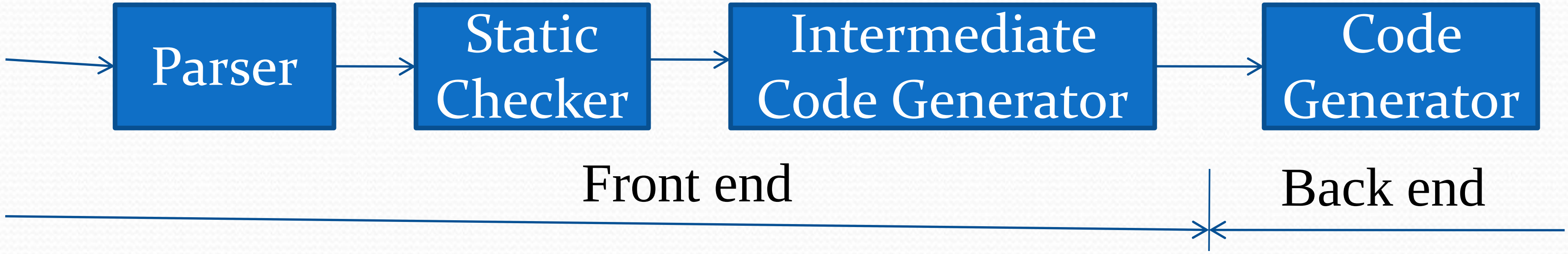
Intermediate Code Generation

Outline

- Variants of Syntax Trees
- Three-address code
- Types and declarations
- Translation of expressions

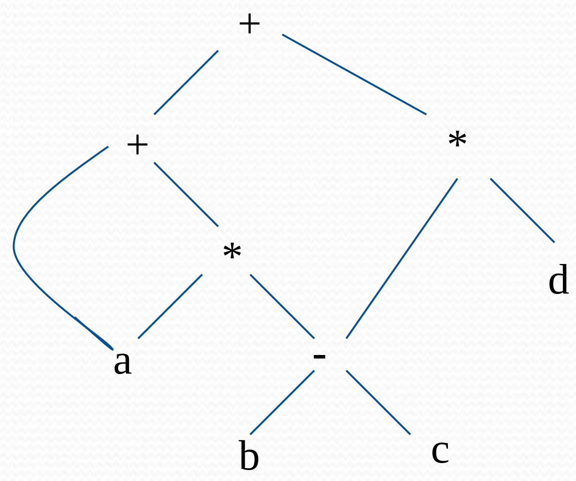
Introduction

- Intermediate code is the interface between front end and back end in a compiler
- Ideally the details of source language are confined to the front end and the details of target machines to the back end (a m*n model)
- In this chapter we study intermediate representations, static type checking and intermediate code generation



Variants of syntax trees

- It is sometimes beneficial to crate a DAG instead of tree for Expressions.
- This way we can easily show the common sub-expressions and then use that knowledge during code generation
- Example: a+a*(b-c)+(b-c)*d



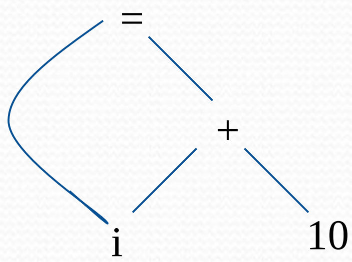
SDD for creating DAG's

Production	Semantic Rules
1) E -> E1+T	E.node= new Node('+', E1.node,T.node)
2) E -> E1-T	E.node= new Node('-', E1.node,T.node)
3) E -> T	E.node = T.node
4) T -> (E)	T.node = E.node
5) T -> id	T.node = new Leaf(id, id.entry)
6) T -> num	T.node = new Leaf(num, num.val)

Example:

1) p1=Leaf(id, entry-a)
2) P2=Leaf(id, entry-a)=p1
3) p3=Leaf(id, entry-b)
4) p4=Leaf(id, entry-c)
5) p5=Node('-',p3,p4)
6) p6=Node('*',p1,p5)
7) p7=Node('+',p1,p6)
8) p8=Leaf(id,entry-b)=p3
9) p9=Leaf(id,entry-c)=p4
10) p10=Node('-',p3,p4)=p5
11) p11=Leaf(id,entry-d)
12) p12=Node('*',p5,p11)
13) p13=Node('+',p7,p12)

Value-number method for constructing DAG's



id		
num	10	
+	1	2
3	1	3

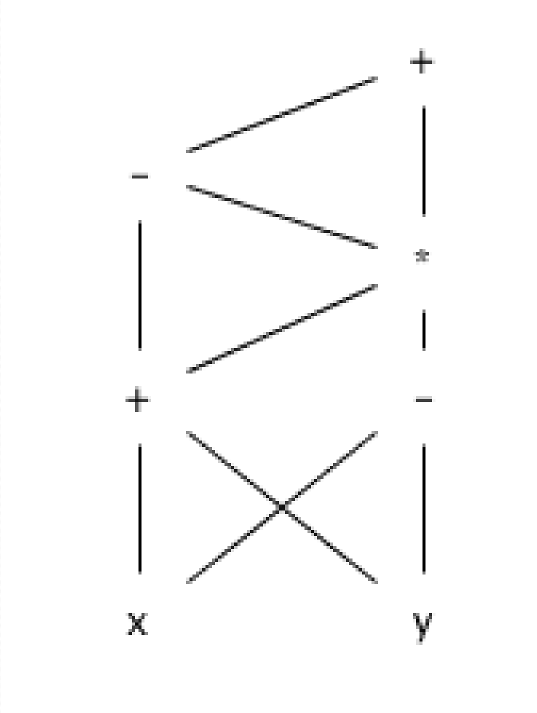
To entry for i

Nodes of a DAG for i=i+10 allocated in an array

- Algorithm
  - Search the array for a node M with label op, left child l and right child r
  - If there is such a node, return the value number M
  - If not create in the array a new node N with label op, left child l, and right child r and return its value
- We may use a hash table

Exercises

Construct the DAG for the expression
((x+y)-((x+y)*(x-y)))+((x+y)*(x-y))



Exercises

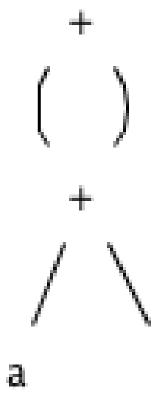
Construct the DAG and identify the value numbers for the subexpressions of the following expressions, assuming + associates from the left.

a. a+b+(a+b)

b. a+b+a+b

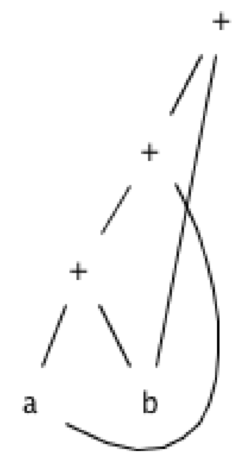
c. a+a+(a+a+a+(a+a+a+a))

a. a+b+(a+b)



1	id	a	
2	id	b	
3	+	1	2
4	+	3	3

b. a+b+a+b



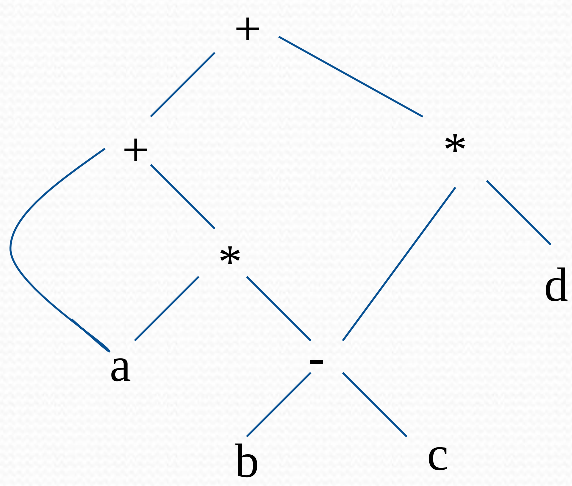
1	id	a	
2	id	b	
3	+	1	2
4	+	3	1
5	+	4	2

c. a+a+(a+a+a+(a+a+a+a))

1	id	a	
2	+	1	1
3	+	2	1
4	+	3	1
5	+	3	4
6	+	2	5

Three address code

- In a three address code there is at most one operator at the right side of an instruction
- Example:



t1 = b – c

t2 = a * t1

t3 = a + t2

t4 = t1 * d

t5 = t3 + t4

Forms of three address instructions

- x = y op z
- x = op y
- x = y
- goto L
- if x goto L and ifFalse x goto L
- if x relop y goto L
- Procedure calls using:
  - param x
  - call p,n
  - y = call p,n
- x = y[i] and x[i] = y
- x = &y and x = *y and *x =y

Example

- do i = i+1; while (a[i] < v);

```
L:      t1 = i + 1
        i = t1
        t2 = i * 8
        t3 = a[t2]
        if t3 < v goto L
```

Symbolic labels

```
100:    t1 = i + 1
101:    i = t1
102:    t2 = i * 8
103:    t3 = a[t2]
104:    if t3 < v goto 100
```

Position numbers

Data structures for three address codes

- Quadruples
  - Has four fields: op, arg1, arg2 and result
- Triples
  - Temporaries are not used and instead references to instructions are made
- Indirect triples
  - In addition to triples we use a list of pointers to triples

Example

- b * minus c + b * minus c

Three address code

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

Quadruples

op	arg1	arg2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

Triples

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

Indirect Triples

	op
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

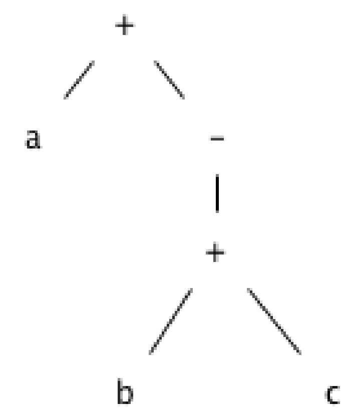
	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Exercises

Translate the arithmetic expression a + - (b + c) into

a. A Syntax tree
b. Quadruples
c. Triples
d. Indirect Triples

a.



b.

	op	arg1	arg2	result
0	+	b	c	t1
1	minus	t1		t2
2	+	a	t2	t3

c.

	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

d.

	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

	instruction
0	(0)
1	(1)
2	(2)

Exercises

a) a = b[i] + c[j]

0)	=[]	b	i	t1
1)	=[]	c	j	t2
2)	+	t1	t2	t3
3)	=	t3		a

0)	=[]	b	i
1)	=[]	c	j
2)	+	(0)	(1)
3)	=	a	(2)

0)	=[]	b	i
1)	=[]	c	j
2)	+	(0)	(1)
3)	=	a	(2)

0)
1)
2)
3)

b. a[i] = b*c - b*d

0)	*	b	c	t1
1)	*	b	d	t2
2)	-	t1	t2	t3
3)	[]=	a	i	t4
4)	=	t3		t4

0)	*	b	c
1)	*	b	d
2)	-	(0)	(1)
3)	[]=	a	i
4)	=	(3)	(2)

0)	*	b	c
1)	*	b	d
2)	-	(0)	(1)
3)	[]=	a	i
4)	=	(3)	(2)

0)
1)
2)
3)
4)

Exercises

c. x = f(y+1) + 2

0)	+	y	1	t1
1)	param	t1		
2)	call	f	1	t2
3)	+	t2	2	t3
4)	=	t3		x

0)	+	y	1
1)	param	(0)	
2)	call	f	1
3)	+	(2)	2
4)	=	x	(3)

0)			
1)			
2)			
3)			
4)			

0)	+	y	1
1)	param	(0)	
2)	call	f	1
3)	+	(2)	2
4)	=	x	(3)

Static Single-Assignment Form

- Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations.
- Two distinctive aspects distinguish SSA from three-address code.
- The first is that all assignments in SSA are to variables with distinct names; hence the term static single-assignment

p = a + b	$p_1 = a + b$
q = p - c	$q_1 = p_1 - c$
p = q * d	$p_2 = q_1 * d$
p = e - p	$p_3 = e - p_2$
q = p + q	$q_2 = p_3 + q_1$
(a) Three-address code.	(b) Static single-assignment form.

Figure : Intermediate program in three-address code and SSA

Types and Declarations

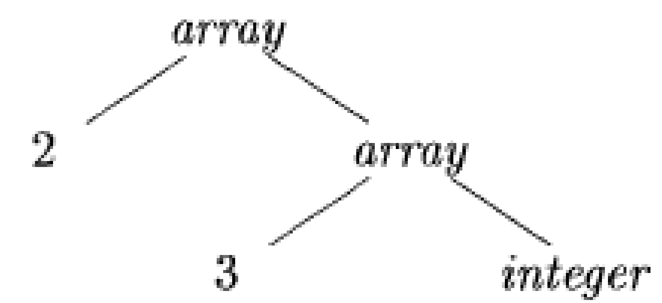
- The applications of types can be grouped under checking and translation:

  - *Type checking* uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator.
  - For example, the && operator in Java expects its two operands to be booleans; the result is also of type boolean.

  - *Translation Applications*. From the type of a name, a compiler can determine the storage that will be needed for that name at run time. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic operator, among other things.

Type Expressions

Example: int[2][3]
array(2,array(3,integer))



- A basic type is a type expression
- A type name is a type expression
- A type expression can be formed by applying the array type constructor to a number and a type expression.
- A record is a data structure with named field
- A type expression can be formed by using the type constructor → for function types
- If s and t are type expressions, then their Cartesian product s*t is a type expression
- Type expressions may contain variables whose values are type expressions

Type Equivalence

- When type expressions are represented by graphs, two types are structurally equivalent if and only if one of the following conditions is true:
  - They are the same basic type.
  - They are formed by applying the same constructor to structurally equivalent types.
  - One is a type name that denotes the other.

Declarations

$$
\begin{array}{lcl}
D & \rightarrow & T\ \textbf{id}\ ;\ D \mid \epsilon \\
T & \rightarrow & B\ C \mid \textbf{record}\ '\{'\ D\ '\}' \\
B & \rightarrow & \textbf{int} \mid \textbf{float} \\
C & \rightarrow & \epsilon \mid [\ \textbf{num}\ ]\ C
\end{array}
$$

- Nonterminal D generates a sequence of declarations. Nonterminal T generates basic, array, or record types.

- Nonterminal B generates one of the basic types int and float. Nonterminal C, for "component," generates strings of zero or more integers, each integer surrounded by brackets.

- An array type consists of a basic type specified by B, followed by array components specified by nonterminal C.

- A record type (the second production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

Storage Layout for Local Names

- Suppose that storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory. Typically, a byte is eight bits, and some number of bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of the first byte.

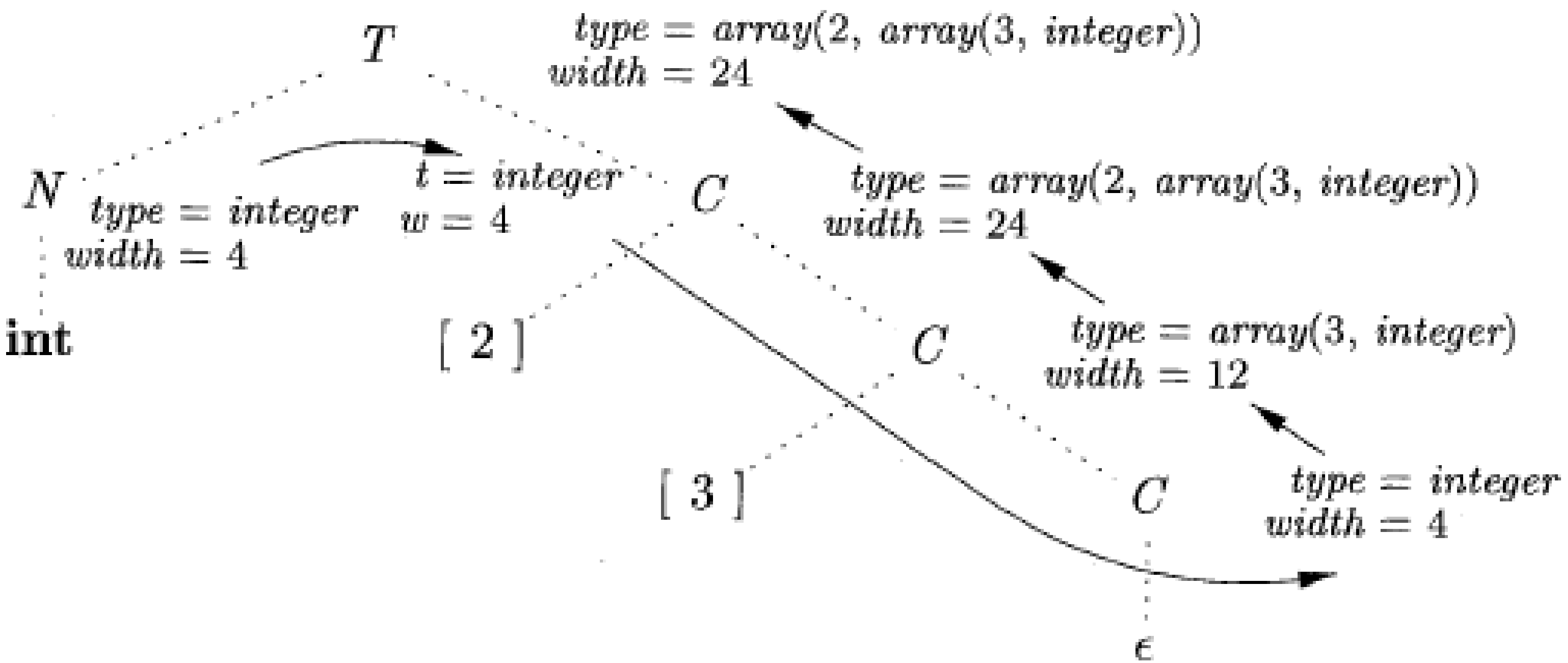
- The *width* of a type is the number of storage units needed for objects of that type. A basic type, such as a character, integer, or float, requires an integral number of bytes. For easy access, storage for aggregates such as arrays and classes is allocated in one contiguous block of bytes.

T	$\rightarrow$	B	$\{\ t = B.type;\ w = B.width;\ \}$
		C	
B	$\rightarrow$	**int**	$\{\ B.type = integer;\ B.width = 4;\ \}$
B	$\rightarrow$	**float**	$\{\ B.type = float;\ B.width = 8;\ \}$
C	$\rightarrow$	ϵ	$\{\ C.type = t;\ C.width = w;\ \}$
C	$\rightarrow$	[**num**] C_1	$\{\ array(\textbf{num}.value,\ C_1.type);$ $C.width = \textbf{num}.value \times C_1.width;\ \}$

Computing types and their widths

Storage Layout for Local Names

- Syntax-directed translation of array types



Sequences of Declarations

- $$\begin{array}{llll} P & \rightarrow & & \{\ \mathit{offset} = 0;\ \} \\ & & D & \\ D & \rightarrow & T\ \mathbf{id}\ ; & \{\ \mathit{top.put}(\mathbf{id}.\mathit{lexeme},\ T.\mathit{type},\ \mathit{offset}); \\ & & & \quad \mathit{offset}\ =\ \mathit{offset} + T.\mathit{width};\ \} \\ & & D_1 & \\ D & \rightarrow & \epsilon & \end{array}$$

- Actions at the end:
- $$\begin{array}{llll} P & \rightarrow & M\ D & \\ M & \rightarrow & \epsilon & \{\ \mathit{offset} = 0;\ \} \end{array}$$

Fields in Records and Classes

- ```
  float x;
  record { float x; float y; } p;
  record { int tag; float x; float y; } q;
  ```

- $$
\begin{array}{llll}
T & \rightarrow & \textbf{record}\ '\{' & \{\ Env.push(top);\ top = \textbf{new}\ Env();\\
 & & & \phantom{\{}\ Stack.push(offset);\ offset = 0;\ \}\\
 & & D\ '\}' & \{\ T.type = record(top);\ T.width = offset;\\
 & & & \phantom{\{}\ top = Env.pop();\ offset = Stack.pop();\ \}
\end{array}
$$
  ```

Translation of Expressions and Statements

- We discussed how to find the types and offset of variables
- We have therefore necessary preparations to discuss about translation to intermediate code
- We also discuss the type checking

Three-address code for expressions

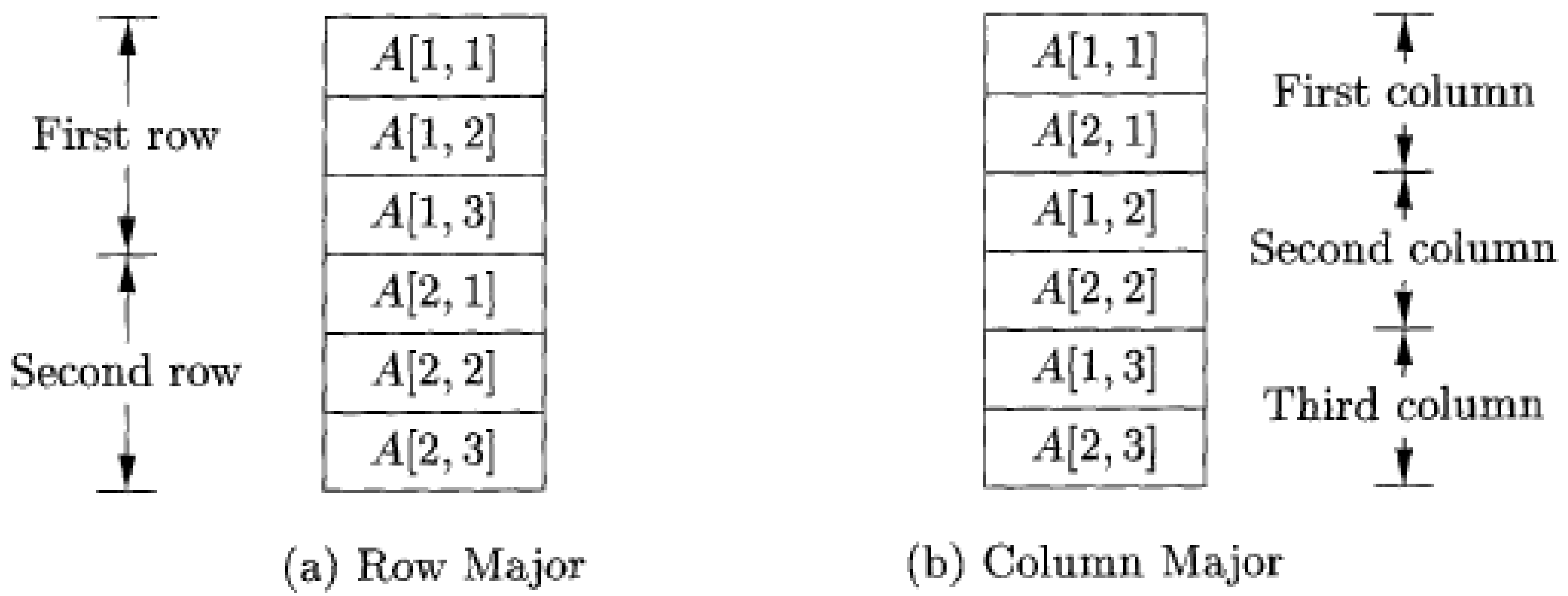
PRODUCTION	SEMANTIC RULES
$S \rightarrow \textbf{id} = E\ ;$	$S.code = E.code \ \Vert$ $gen(top.get(\textbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \ \Vert\ E_2.code \ \Vert$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ -E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \ \Vert$ $gen(E.addr\ '='\ '\textbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \textbf{id}$	$E.addr = top.get(\textbf{id}.lexeme)$ $E.code = ''$

Incremental Translation

$$
\begin{array}{lcll}
S & \rightarrow & \mathbf{id} = E \;; & \{\; gen(\; top.get(\mathbf{id}.lexeme) \;'='\; E.addr); \;\} \\
E & \rightarrow & E_1 + E_2 & \{\; E.addr = \mathbf{new}\; Temp\,(); \\
 & & & \quad gen(E.addr \;'='\; E_1.addr \;'+'\; E_2.addr); \;\} \\
 & | & -\; E_1 & \{\; E.addr = \mathbf{new}\; Temp\,(); \\
 & & & \quad gen(E.addr \;'='\; '\mathbf{minus}'\; E_1.addr); \;\} \\
 & | & (\; E_1 \;) & \{\; E.addr = E_1.addr; \;\} \\
 & | & \mathbf{id} & \{\; E.addr = top.get(\mathbf{id}.lexeme); \;\}
\end{array}
$$

Addressing Array Elements

- Layouts for a two-dimensional array:



Semantic actions for array reference

$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} = E\ ; & \{\ gen(\ top.get(\textbf{id}.lexeme)\ '='\ E.addr);\ \} \\[1em]
 & | & L = E\ ; & \{\ gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} \\[1em]
E & \rightarrow & E_1 + E_2 & \{\ E.addr = \textbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\[1em]
 & | & \textbf{id} & \{\ E.addr = top.get(\textbf{id}.lexeme);\ \} \\[1em]
 & | & L & \{\ E.addr = \textbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \} \\[1em]
L & \rightarrow & \textbf{id}\ [\ E\ ] & \{\ L.array = top.get(\textbf{id}.lexeme); \\
 & & & \quad L.type = L.array.type.elem; \\
 & & & \quad L.addr = \textbf{new}\ Temp\,(); \\
 & & & \quad gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\[1em]
 & | & L_1\ [\ E\ ] & \{\ L.array = L_1.array; \\
 & & & \quad L.type = L_1.type.elem; \\
 & & & \quad t = \textbf{new}\ Temp\,(); \\
 & & & \quad L.addr = \textbf{new}\ Temp\,(); \\
 & & & \quad gen(t\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
 & & & \quad gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

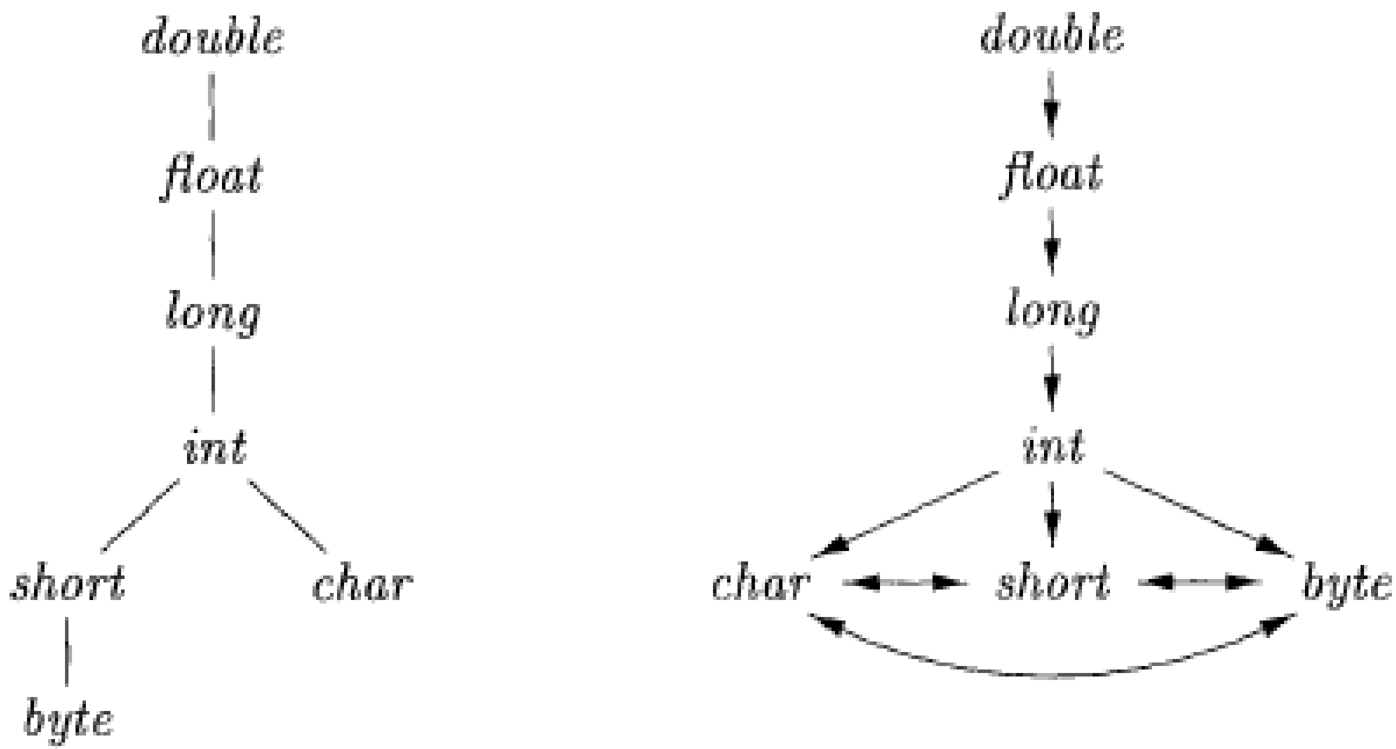
Translation of Array References

Nonterminal *L* has three synthesized attributes:

- *L.addr*
- L.array
- L.type

END

Conversions between primitive types in Java



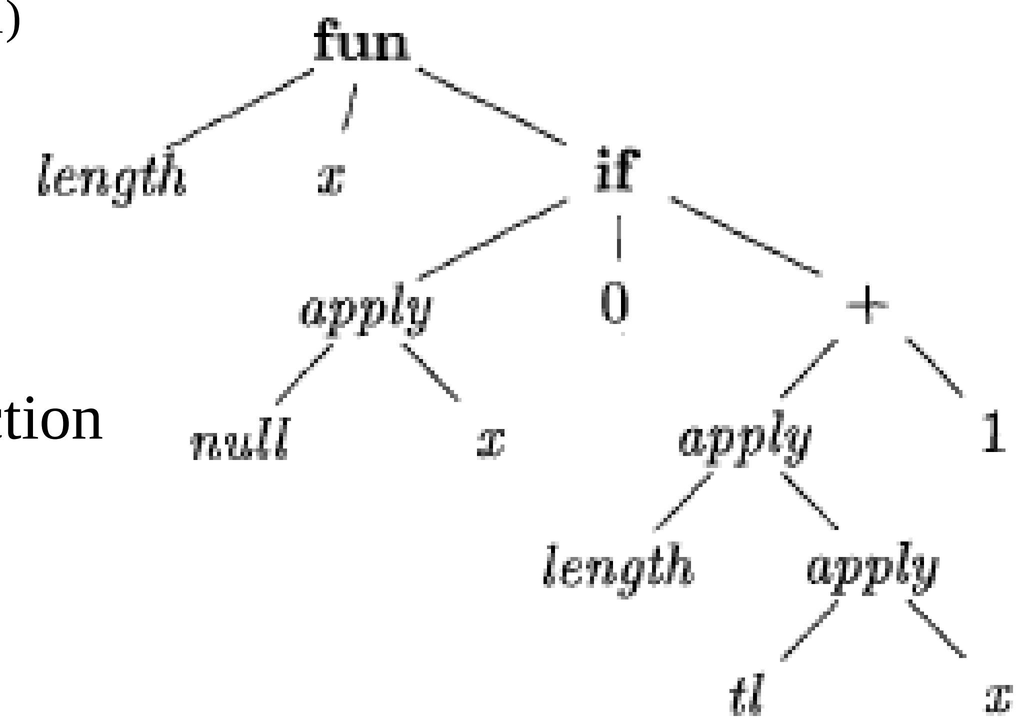
(a) Widening conversions (b) Narrowing conversions

Introducing type conversions into expression evaluation

$$
\begin{array}{lcll}
E & \to & E_1 + E_2 & \{\ E.type\ =\ max(E_1.type, E_2.type); \\
 & & & \quad a_1\ =\ widen(E_1.addr, E_1.type, E.type); \\
 & & & \quad a_2\ =\ widen(E_2.addr, E_2.type, E.type); \\
 & & & \quad E.addr = \textbf{new}\ Temp\,(); \\
 & & & \quad gen(E.addr\ '='\ a_1\ '+'\ a_2);\ \}
\end{array}
$$

Abstract syntax tree for the function definition

fun length(x) =
  if null(x) then 0 else length(tl(x)+1)



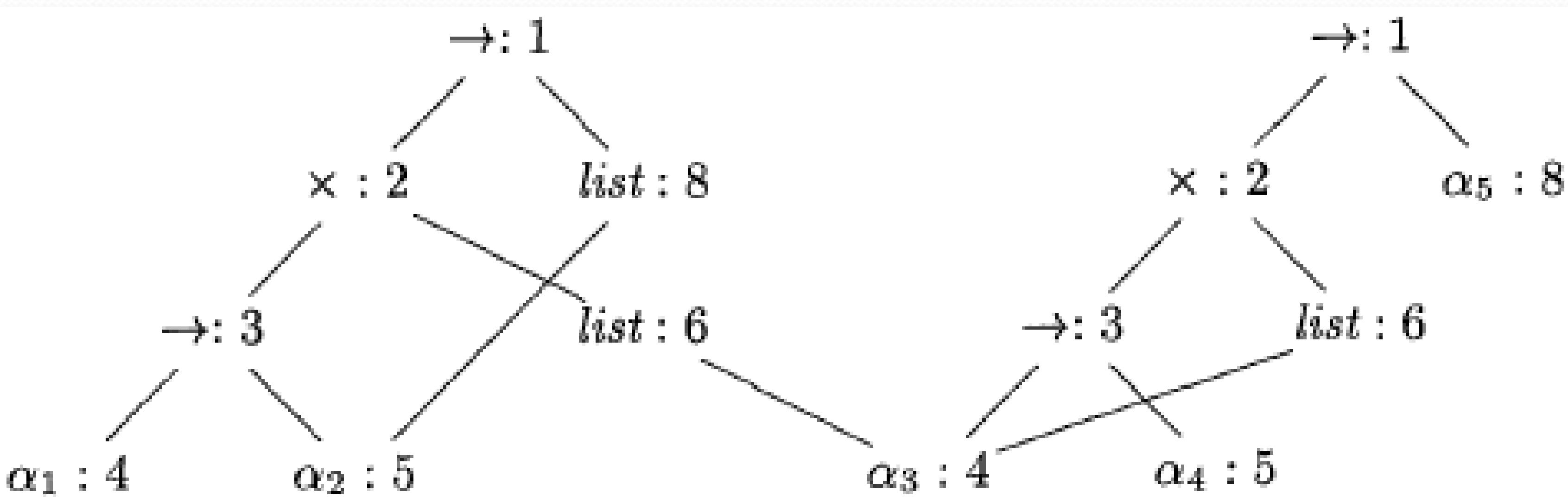
This is a polymorphic function in ML language

Inferring a type for the function *length*

Line	Expression : Type		Unify
1)	$length$:	$\beta \to \gamma$	
2)	x :	β	
3)	**if** :	$boolean \times \alpha_i \times \alpha_i \to \alpha_i$	
4)	$null$:	$list(\alpha_n) \to boolean$	
5)	$null(x)$:	$boolean$	$list(\alpha_n) = \beta$
6)	0 :	$integer$	$\alpha_i = integer$
7)	$+$:	$integer \times integer \to integer$	
8)	tl :	$list(\alpha_t) \to list(\alpha_t)$	
9)	$tl(x)$:	$list(\alpha_t)$	$list(\alpha_t) = list(\alpha_n)$
10)	$length(tl(x))$:	γ	$\gamma = integer$
11)	1 :	$integer$	
12)	$length(tl(x)) + 1$:	$integer$	
13)	**if**$(\cdots)$:	$integer$	

Algorithm for Unification

$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$$
$$((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5$$



Unification algorithm

```
boolean unify (Node m, Node n) {
    s = find(m); t = find(n);
    if ( s = t ) return true;
    else if ( nodes s and t represent the same basic type ) return true;
    else if (s is an op-node with children s1 and s2 and
            t is an op-node with children t1 and t2) {
            union(s , t) ;
            return unify(s1, t1) and unify(s2, t2);
    }
    else if s or t represents a variable {
            union(s, t) ;
            return true;
    }
    else return false;
}
```

Control Flow

boolean expressions are often used to:

- *Alter the flow of control.*
- *Compute logical values.*

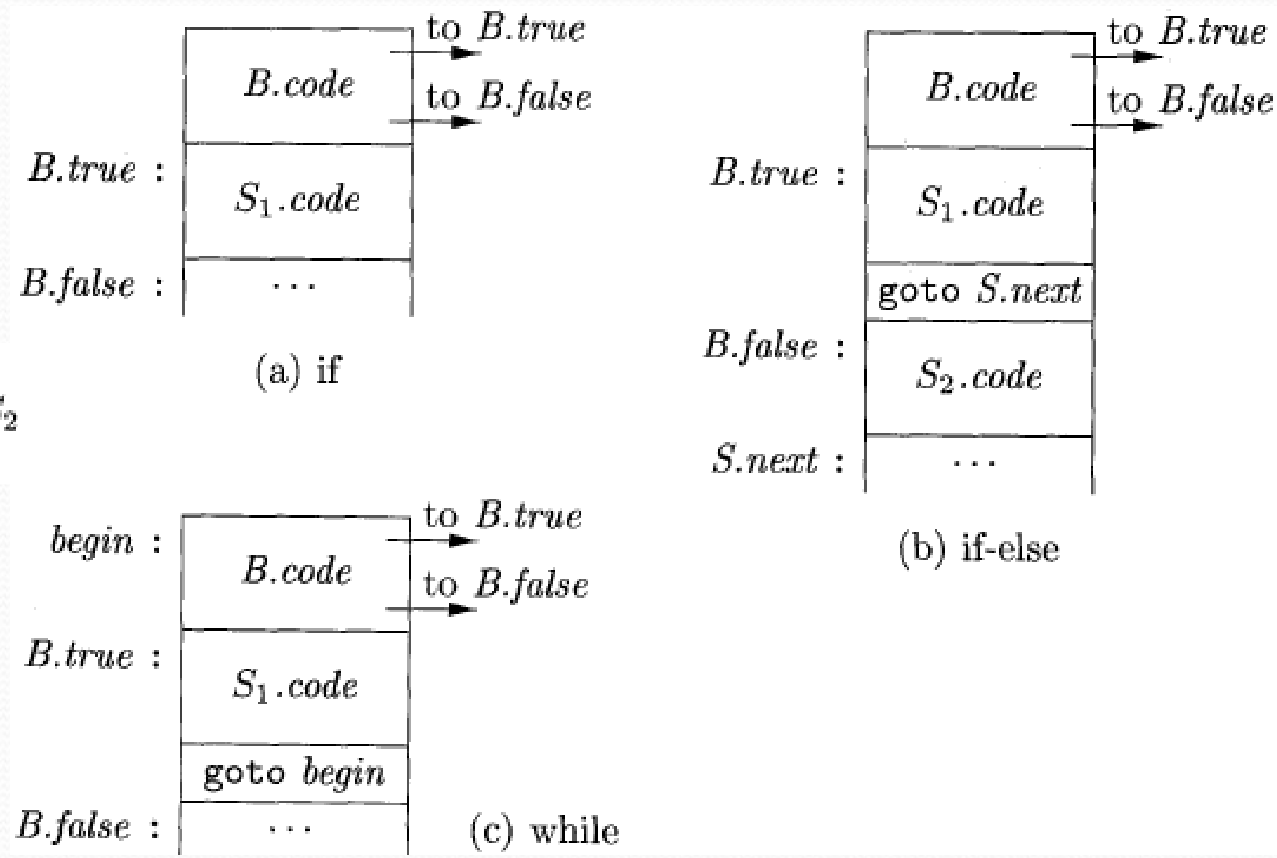
Short-Circuit Code

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

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

Flow-of-Control Statements

$$
\begin{aligned}
S &\rightarrow \textbf{if}\ (\ B\ )\ S_1 \\
S &\rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2 \\
S &\rightarrow \textbf{while}\ (\ B\ )\ S_1
\end{aligned}
$$



(a) if

(b) if-else

(c) while

Syntax-directed definition

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

Generating three-address code for booleans

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow \,!\, B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow \textbf{false}$	$B.code = gen('\texttt{goto}'\ B.false)$

translation of a simple if-statement

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

- ```
        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:
  ```

Backpatching

- Previous codes for Boolean expressions insert symbolic labels for jumps
- It therefore needs a separate pass to set them to appropriate addresses
- We can use a technique named backpatching to avoid this
- We assume we save instructions into an array and labels will be indices in the array
- For nonterminal B we use two attributes B.truelist and B.falselist together with following functions:
  - makelist(i): create a new list containing only I, an index into the array of instructions
  - Merge(p1,p2): concatenates the lists pointed by p1 and p2 and returns a pointer to the concatenated list
  - Backpatch(p,i): inserts i as the target label for each of the instruction on the list pointed to by p

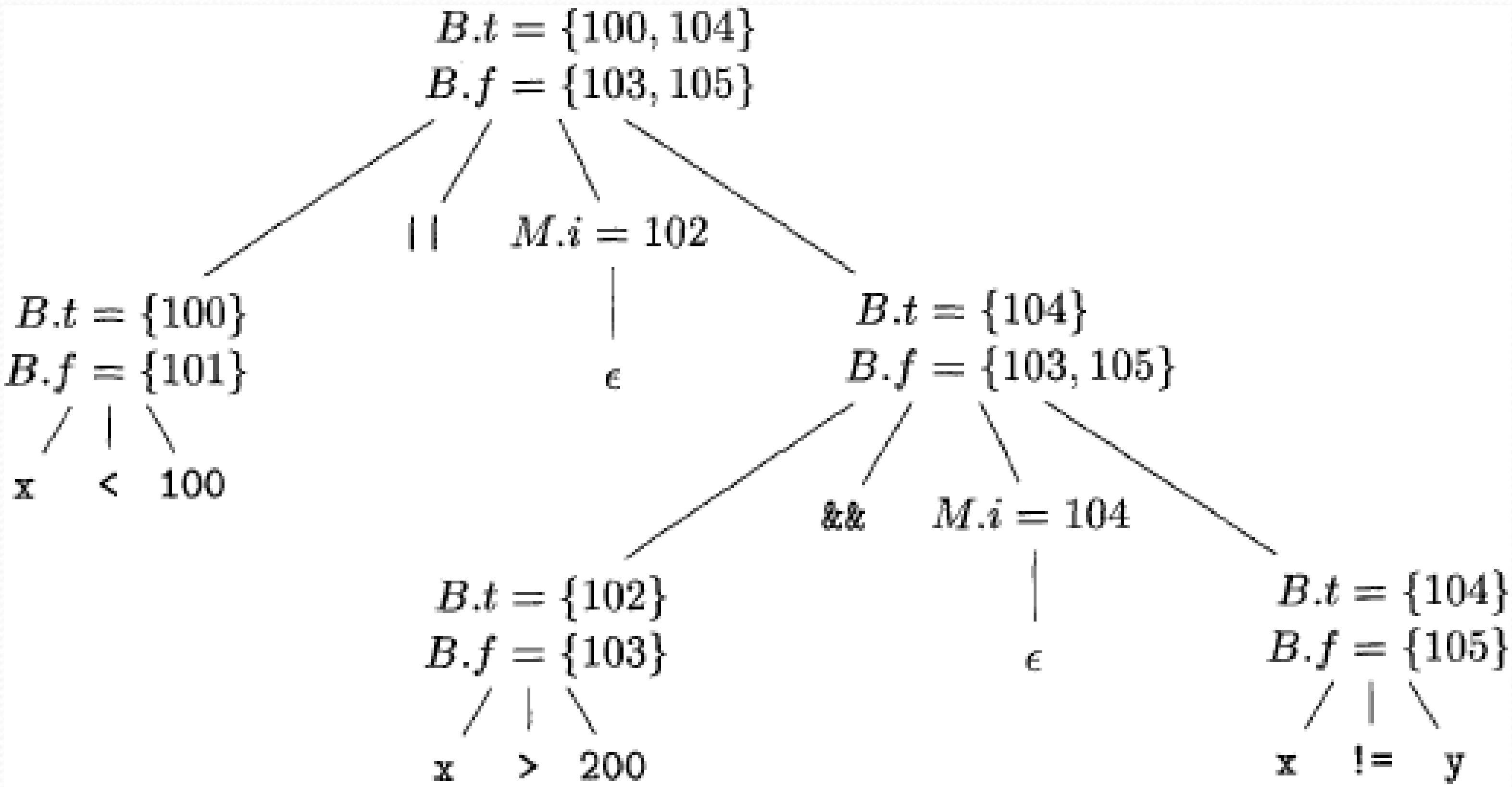
Backpatching for Boolean Expressions

- $B \rightarrow B_1 \; || \; M \; B_2 \mid B_1 \; \&\& \; M \; B_2 \mid \; ! \; B_1 \mid (\; B_1 \;) \mid E_1 \; \textbf{rel} \; E_2 \mid \textbf{true} \mid \textbf{false}$
  $M \rightarrow \epsilon$
-

1)	$B \rightarrow B_1 \; || \; M \; B_2$	$\{$ $backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist; \}$
2)	$B \rightarrow B_1 \; \&\& \; M \; B_2$	$\{$ $backpatch(B_1.truelist, M.instr);$ $B.truelist = B_2.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist); \}$
3)	$B \rightarrow \; ! \; B_1$	$\{$ $B.truelist = B_1.falselist;$ $B.falselist = B_1.truelist; \}$
4)	$B \rightarrow (\; B_1 \;)$	$\{$ $B.truelist = B_1.truelist;$ $B.falselist = B_1.falselist; \}$
5)	$B \rightarrow E_1 \; \textbf{rel} \; E_2$	$\{$ $B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $emit('\texttt{if}' \; E_1.addr \; \textbf{rel}.op \; E_2.addr \; '\texttt{goto \_}');$ $emit('\texttt{goto \_}'); \}$
6)	$B \rightarrow \textbf{true}$	$\{$ $B.truelist = makelist(nextinstr);$ $emit('\texttt{goto \_}'); \}$
7)	$B \rightarrow \textbf{false}$	$\{$ $B.falselist = makelist(nextinstr);$ $emit('\texttt{goto \_}'); \}$
8)	$M \rightarrow \epsilon$	$\{$ $M.instr = nextinstr; \}$

Backpatching for Boolean Expressions

- Annotated parse tree for x < 100 || x > 200 && x ! = y



Flow-of-Control Statements

$S \rightarrow \textbf{while}\ M_1\ (\ B\ )\ M_2\ S_1$

1) $S \rightarrow \textbf{if}\ (\ B\ )\ M\ S_1$ $\{$ $backpatch(B.truelist,\ M.instr);$
$S.nextlist\ =\ merge(B.falselist,\ S_1.nextlist);$ $\}$

2) $S \rightarrow \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$
$\{$ $backpatch(B.truelist,\ M_1.instr);$
$backpatch(B.falselist,\ M_2.instr);$
$temp\ =\ merge(S_1.nextlist,\ N.nextlist);$
$S.nextlist\ =\ merge(temp,\ S_2.nextlist);$ $\}$

3) $S \rightarrow \textbf{while}\ M_1\ (\ B\ )\ M_2\ S_1$
$\{$ $backpatch(S_1.nextlist,\ M_1.instr);$
$backpatch(B.truelist,\ M_2.instr);$
$S.nextlist\ =\ B.falselist;$
$emit(\texttt{'goto'}\ M_1.instr);$ $\}$

4) $S \rightarrow \{\ L\ \}$ $\{$ $S.nextlist\ =\ L.nextlist;$ $\}$

5) $S \rightarrow A\ ;$ $\{$ $S.nextlist\ =\ \textbf{null};$ $\}$

6) $M \rightarrow \epsilon$ $\{$ $M.instr\ =\ nextinstr;$ $\}$

7) $N \rightarrow \epsilon$ $\{$ $N.nextlist\ =\ makelist(nextinstr);$
$emit(\texttt{'goto \_'});$ $\}$

8) $L \rightarrow L_1\ M\ S$ $\{$ $backpatch(L_1.nextlist,\ M.instr);$
$L.nextlist\ =\ S.nextlist;$ $\}$

9) $L \rightarrow S$ $\{$ $L.nextlist\ =\ S.nextlist;$ $\}$

Translation of a switch-statement

switch (E) {
    case V_1: S_1
    case V_2: S_2
    ...
    case V_{n-1}: S_{n-1}
    default: S_n
}

	code to evaluate E into `t`
	`goto test`
L_1:	code for S_1
	`goto next`
L_2:	code for S_2
	`goto next`
	...
L_{n-1}:	code for S_{n-1}
	`goto next`
L_n:	code for S_n
	`goto next`
`test:`	`if t =` V_1 `goto` L_1
	`if t =` V_2 `goto` L_2
	...
	`if t =` V_{n-1} `goto` L_{n-1}
	`goto` L_n
`next:`	

	code to evaluate E into `t`
	`if t !=` V_1 `goto` L_1
	code for S_1
	`goto next`
L_1:	`if t !=` V_2 `goto` L_2
	code for S_2
	`goto next`
L_2:	
	...
L_{n-2}:	`if t !=` V_{n-1} `goto` L_{n-1}
	code for S_{n-1}
	`goto next`
L_{n-1}:	code for S_n
`next:`	

Readings

- Chapter 6 of the book