UNIT-IV

Intermediate Code Generation

Contents

What is intermediate code?

During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as **intermediate code**

Intermediate code

The following are commonly used intermediate code representation :

- Syntax tree
- Postfix Notation
- Three-Address Code

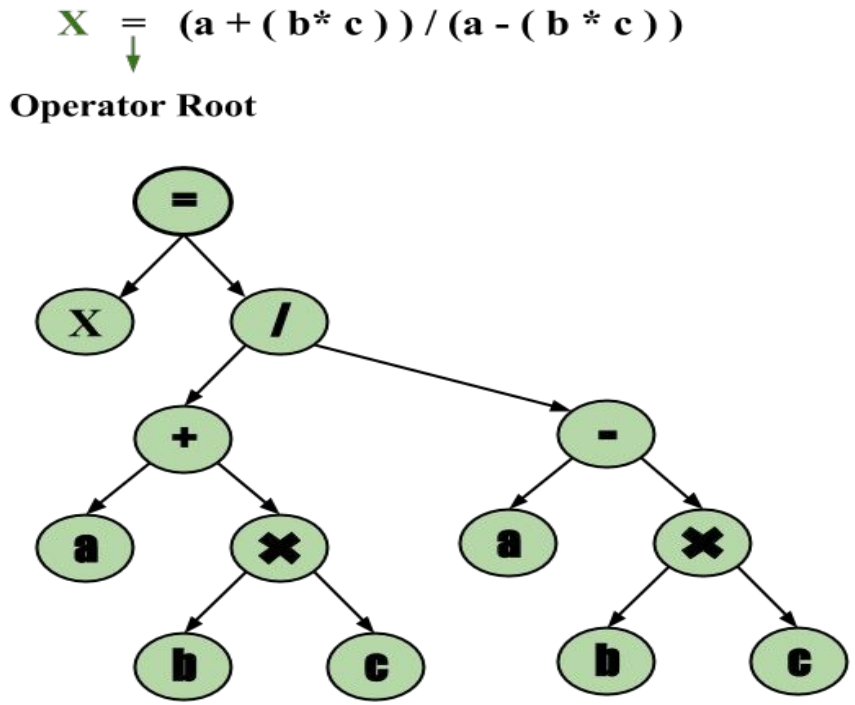
Syntax tree

- Syntax tree is more than condensed form of a parse tree.
- The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree.
- The internal nodes are operators and child nodes are operands.
- To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Syntax tree

- **Example –**

x = (a + b * c) / (a – b * c)



Postfix Notation

- The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b
- The postfix notation for the same expression places the operator at the right end as ab +.
- In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +.
- No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression.
- In postfix notation the operator follows the operand.

Postfix Notation

- **Example –** The postfix representation of the expression (a – b) * (c + d) + (a – b) is

  ab – cd + ab -+*.

Three-Address Code

- A statement involving no more than three references(two for operands and one for result) is known as three address statement.
- A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location).
- Sometimes a statement might contain less than three references but it is still called three address statement.

- For Example :a = b + c * d;
  - The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;

r2 = b + r1;

a = r2
```

Three-Address Code

- A three-address code has at most three address locations to calculate the expression. A three- address code can be represented in three forms :
  - Quadruples
  - Triples
  - Indirect Triples

Quadruples

- Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The example is represented below in quadruples format:

```
r1 = c * d;
r2 = b + r1;
a = r2
```

OP	arg1	arg2	result
*	c	d	r1
+	b	r1	r2
=	r2		a

Triples

•Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

	OP	arg1	arg2
(0)	*	c	d
(1)	+	b	(0)
(2)	=	(1)	

Indirect Triples

•This representation is an enhancement over triples representation.

•It uses pointers instead of position to store results.

• This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

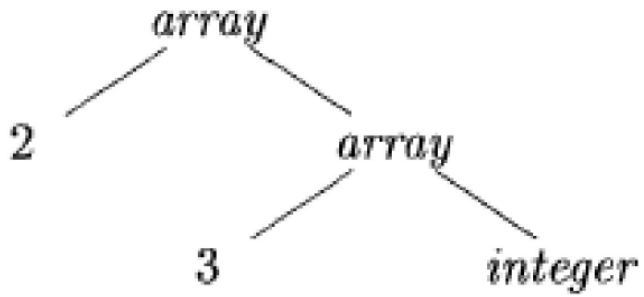
```
r1 = c * d;
r2 = b + r1;
a = r2
```

		OP	arg1	arg2
35	(0)	*	c	d
36	(1)	+	b	(0)
37	(2)	=	(1)	

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 $\rightarrow$ 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

- 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

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

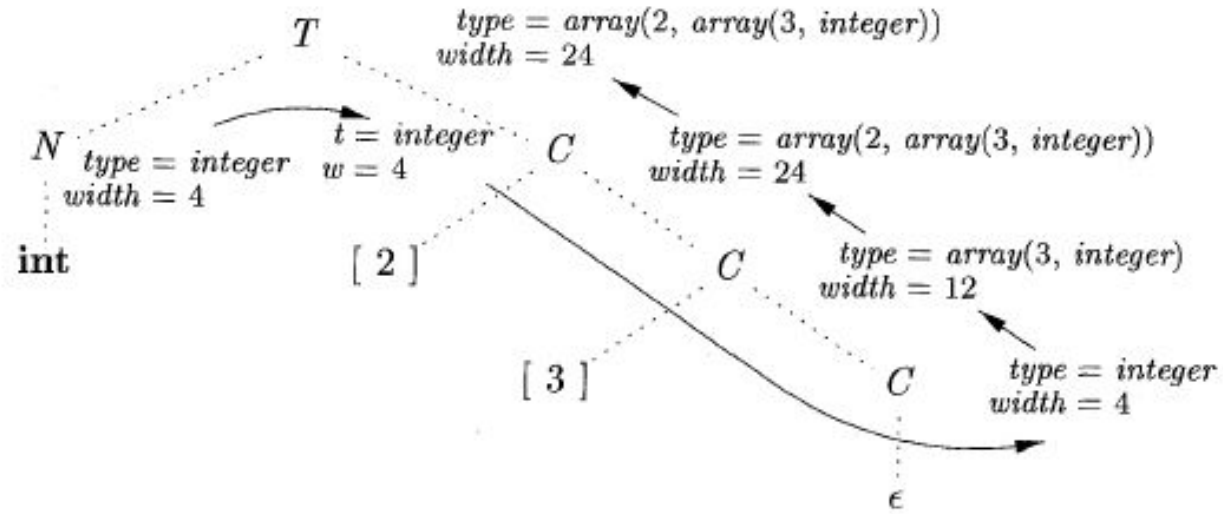
Storage Layout for Local Names

- Computing types and their widths

$$
\begin{array}{lll}
T \rightarrow B & \{\ t = B.type;\ w = B.width;\ \} \\
\quad\ C & \\
B \rightarrow \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B \rightarrow \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C \rightarrow \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C \rightarrow [\ \textbf{num}\ ]\ C_1 & \{\ array(\textbf{num}.value,\ C_1.type); \\
 & \quad C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$

Storage Layout for Local Names

- Syntax-directed translation of array types



Sequences of Declarations

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

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

Fields in Records and Classes

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

- $$T \rightarrow \textbf{record}\ '\{' \quad \{\ Env.push(top);\ top = \textbf{new}\ Env();\ Stack.push(offset);\ offset = 0;\ \}$$
  $$D\ '\}' \quad \{\ T.type = record(top);\ T.width = offset;\ top = Env.pop();\ offset = Stack.pop();\ \}$$
  ```

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

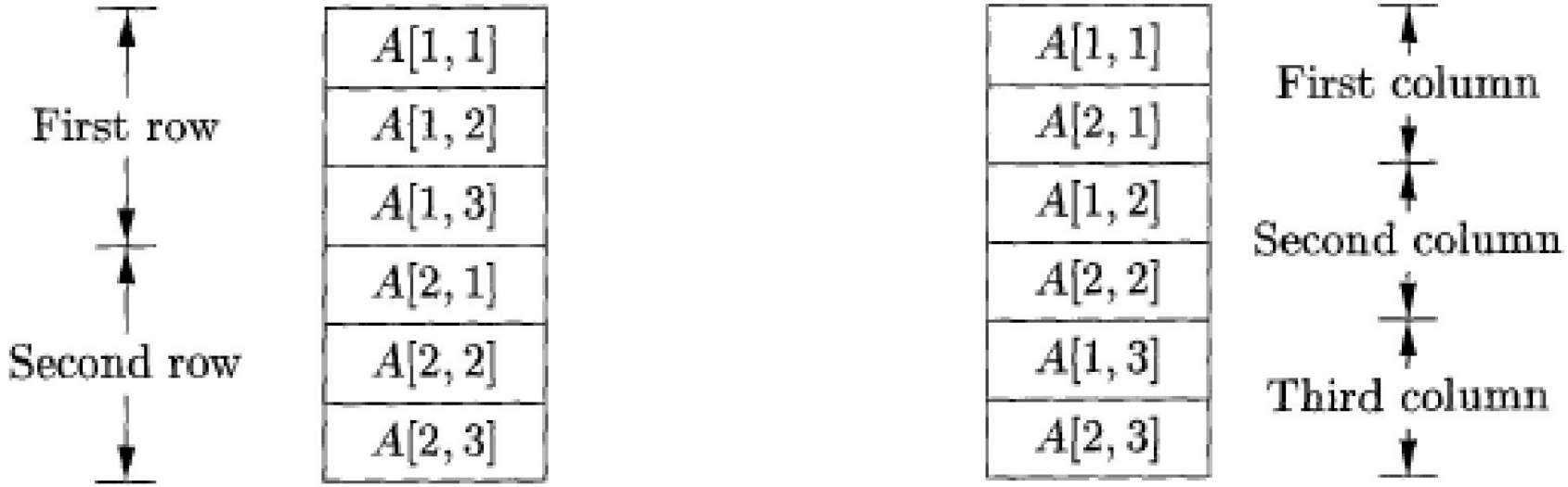
PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \mid\mid$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ - E_1$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

Incremental Translation

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

Addressing Array Elements

- Layouts for a two-dimensional array:



(a) Row Major

(b) Column Major

Semantic actions for array reference

$S \rightarrow \mathbf{id} = E\ ;$ { $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr);$ }

| $L = E\ ;$ { $gen(L.addr.base\ '['\ L.addr\ ']'\ '='\ E.addr);$ }

$E \rightarrow E_1 + E_2$ { $E.addr = \mathbf{new}\ Temp\ ();$
$gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);$ }

| $\mathbf{id}$ { $E.addr = top.get(\mathbf{id}.lexeme);$ }

| L { $E.addr = \mathbf{new}\ Temp\ ();$
$gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');$ }

$L \rightarrow \mathbf{id}\ [\ E\ ]$ { $L.array = top.get(\mathbf{id}.lexeme);$
$L.type = L.array.type.elem;$
$L.addr = \mathbf{new}\ Temp\ ();$
$gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);$ }

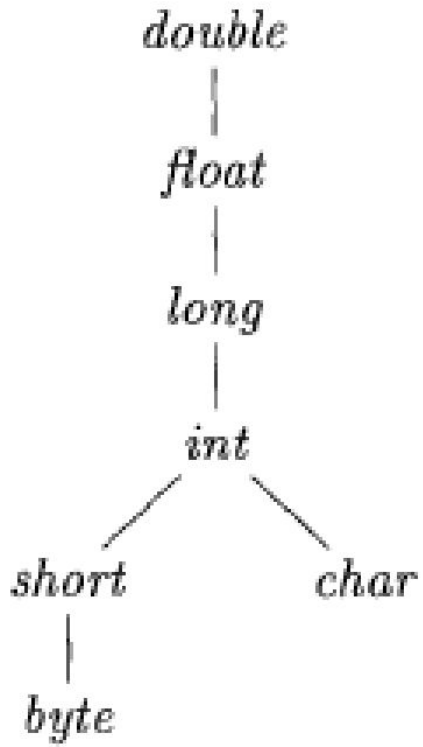
| $L_1\ [\ E\ ]$ { $L.array = L_1.array;$
$L.type = L_1.type.elem;$
$t = \mathbf{new}\ Temp\ ();$
$L.addr = \mathbf{new}\ Temp\ ();$
$gen(t\ '='\ E.addr\ '*'\ L.type.width);$ }
$gen(L.addr\ '='\ L_1.addr\ '+'\ t);$ }

Translation of Array References

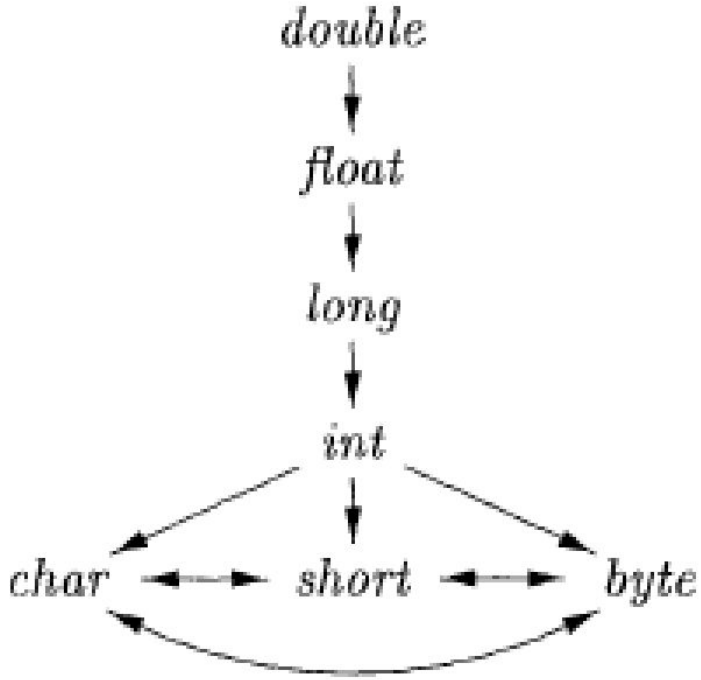
Nonterminal *L* has three synthesized attributes:

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

Conversions between primitive types in Java



(a) Widening conversions



(b) Narrowing conversions

Introducing type conversions into expression evaluation

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

- **Syntax Directed Translations**
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes

Semantic Analysis

- **Semantic Analysis** computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a *Representation Formalism* and an *Implementation Mechanism*.
- As representation formalism this lecture illustrates what are called *Syntax Directed Translations*.

Syntax Directed Translation: Intro

- The **Principle of Syntax Directed Translation** states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- By **Syntax Directed Translations** we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
  - We associate **Attributes** to the grammar symbols representing the language constructs.
  - Values for attributes are computed by **Semantic Rules** associated with grammar productions.

Syntax Directed Translation: Intro (Cont.)

- Evaluation of Semantic Rules may:

– Generate Code;

– Insert information into the Symbol Table;

– Perform Semantic Check;

– Issue error messages;

– etc.

- There are two notations for attaching semantic rules:

1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).

2. **Translation Schemes.** More implementation oriented: the order Indicate in which semantic rules are to be evaluated.

Summary

- Syntax Directed Translations
- **Syntax Directed Definitions**
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes

Syntax Directed Definitions

- **Syntax Directed Definitions** are a generalization of context-free grammars

in which:

1. Grammar symbols have an associated set of **Attributes**;
2. Productions are associated with **Semantic Rules** for computing the values of attributes.

- Such formalism generates **Annotated Parse-Trees** where each node of the

tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).

Syntax Directed Definitions (Cont.)

- The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.
- We distinguish between two kinds of attributes:

1. **Synthesized Attributes.** They are computed from the values of the attributes of the children nodes.
2. **Inherited Attributes.** They are computed from the values of the attributes of both the siblings and the parent nodes.

Form of Syntax Directed Definitions

- Each production, $A \rightarrow \alpha$, is associated with a set of semantic rules:

$b := f(c_1, c_2, \ldots, c_k)$, where f is a function and either

1. b is a **synthesized** attribute of A, and $c_1, c_2, \ldots, c_k$ are attributes of the grammar symbols of the production, or
2. b is an **inherited** attribute of a grammar symbol in α, and $c_1, c_2, \ldots, c_k$ are attributes of grammar symbols in α or attributes of A.

- **Note.** Terminal symbols are assumed to have synthesized attributes supplied by the lexical analyzer.
- Procedure calls (e.g. *print* in the next slide) define values of *Dummy* synthesized attributes of the non terminal on the left-hand side of the production.

Syntax Directed Definitions: An Example

- **Example.** Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

PRODUCTION	SEMANTIC RULE
L → En	*print*(E.val)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
E → T	E.val := T .val
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
T → F	T .val := F.val
F → (E)	F.val := E.val
F → digit	F.val :=digit.*lexval*

S-Attributed Definitions

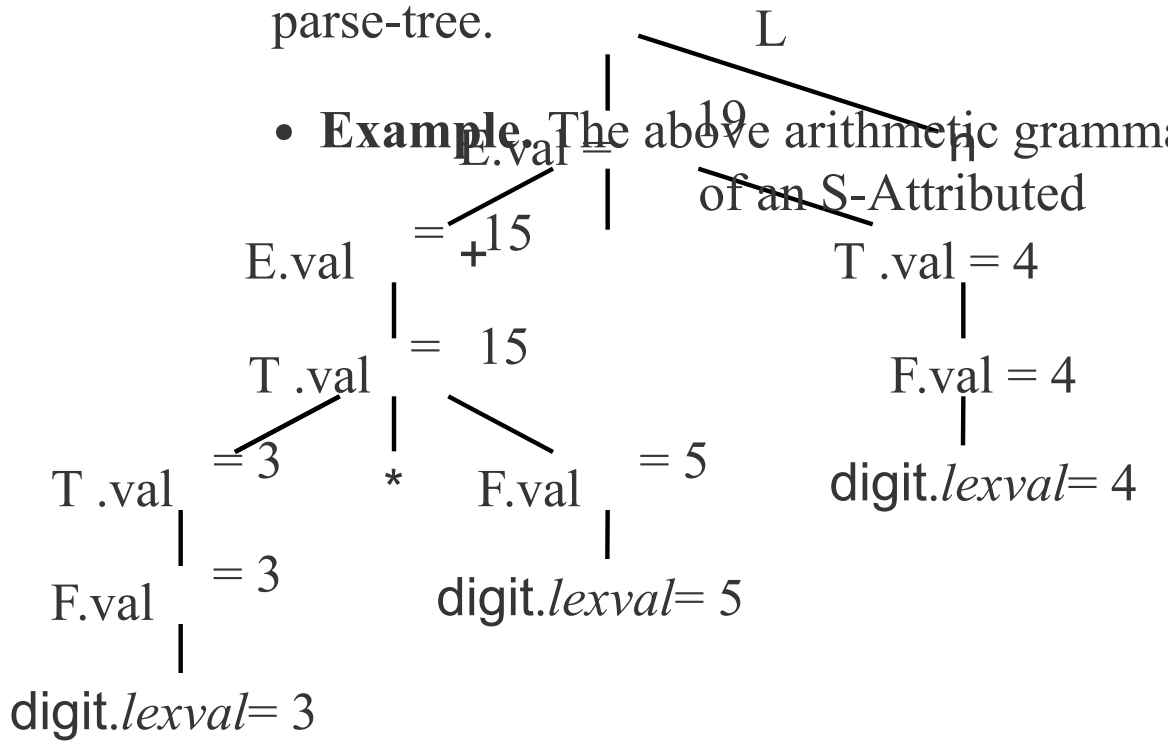
Definition. An **S-Attributed Definition** is a Syntax Directed Definition that uses

only synthesized attributes.

- **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.

- **Example.** The above arithmetic grammar is an example of an S-Attributed

Definition. The annotated parse-tree for the input 3*5+4n is:



Inherited Attributes

- **Inherited Attributes** are useful for expressing the dependence of a construct on the context in which it appears.
- It is always possible to rewrite a syntax directed definition to use only synthesized attributes, but it is often more natural to use both synthesized and inherited attributes.
- **Evaluation Order.** Inherited attributes cannot be evaluated by a simple PreOrder traversal of the parse-tree:

– Unlike synthesized attributes, the order in which the inherited attributes of the children are computed is important!!! Indeed:

* Inherited attributes of the children can depend from both left and right siblings!

Inherited Attributes: An Example

- **Example.** Let us consider the syntax directed definition with both inherited and synthesized attributes for the grammar for "type declarations":

PRODUCTION	SEMANTIC RULE
D → T L	L.in := T .type
T → int	T .type := *integer*
T → real	T .type := *real*
L → L_1 , id	L_1 .in := L.in; *addtype(*id.*entry, L.in)*
L → id	*addtype(*id.*entry, L.in)*

- The non terminal T has a synthesized attribute, *type*, determined by the keyword in the declaration.

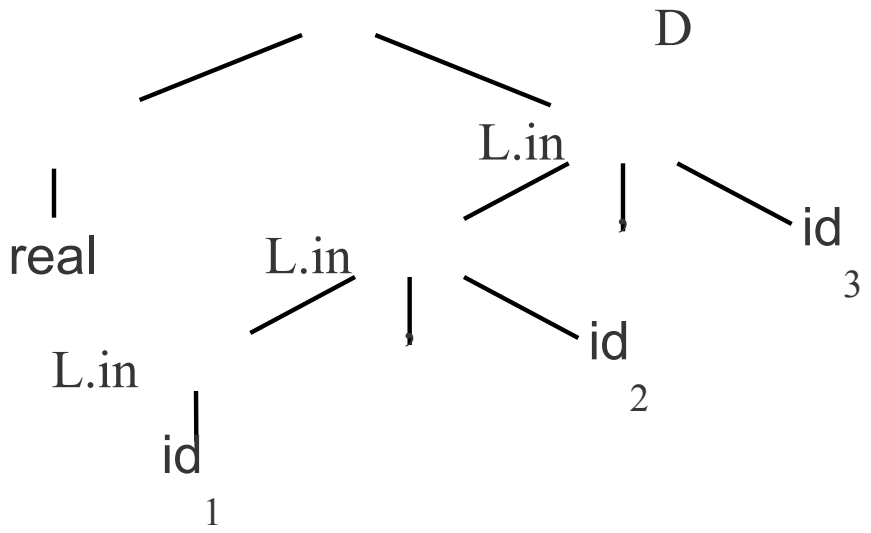
  - The production D → T L is associated with the semantic rule L.in := T .type which set the *inherited* attribute L.in.
  - Note: The production L → L , id distinguishes the two occurrences of L.

Inherited Attributes: An Example (Cont.)

- Synthesized attributes can be evaluated by a PostOrder traversal.

- Inherited attributes that *do not depend from right children* can be evaluated by a classical PreOrder traversal.
- The annotated parse-tree for the input real id_1 , id_2 , id_3 is:

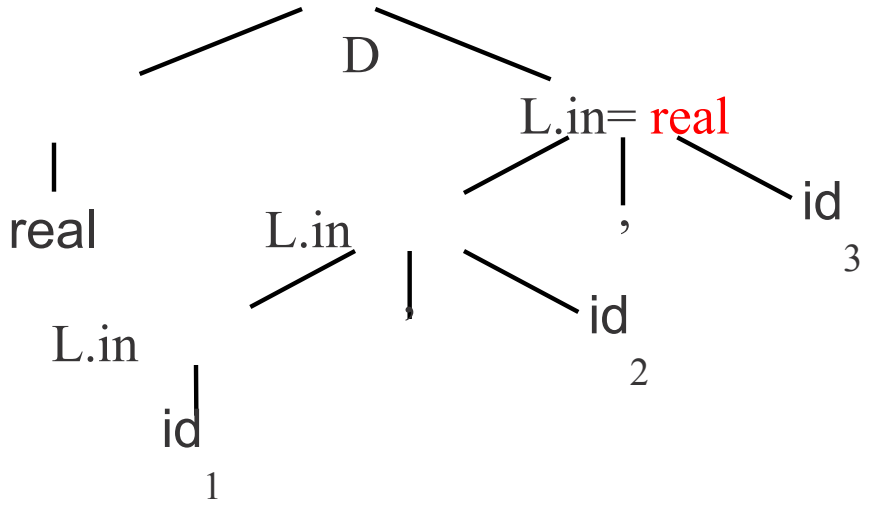
T .type = real



Inherited Attributes: An Example (Cont.)

- Synthesized attributes can be evaluated by a PostOrder traversal.
- Inherited attributes that *do not depend from right children* can be evaluated by a classical PreOrder traversal.
- The annotated parse-tree for the input real id_1 , id_2, id_3 is:

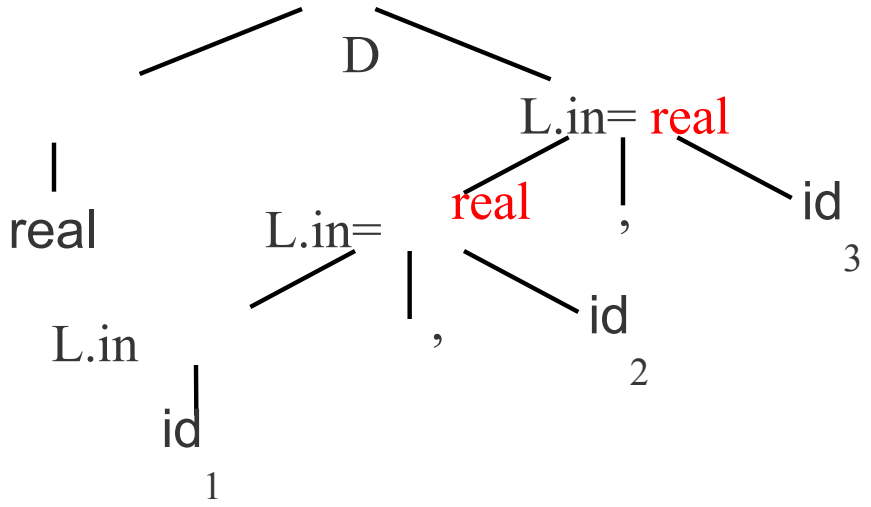
T .type = real



Inherited Attributes: An Example (Cont.)

- Synthesized attributes can be evaluated by a PostOrder traversal.
- Inherited attributes that *do not depend from right children* can be evaluated by a classical PreOrder traversal.
- The annotated parse-tree for the input real id_1 , id_2, id_3 is:

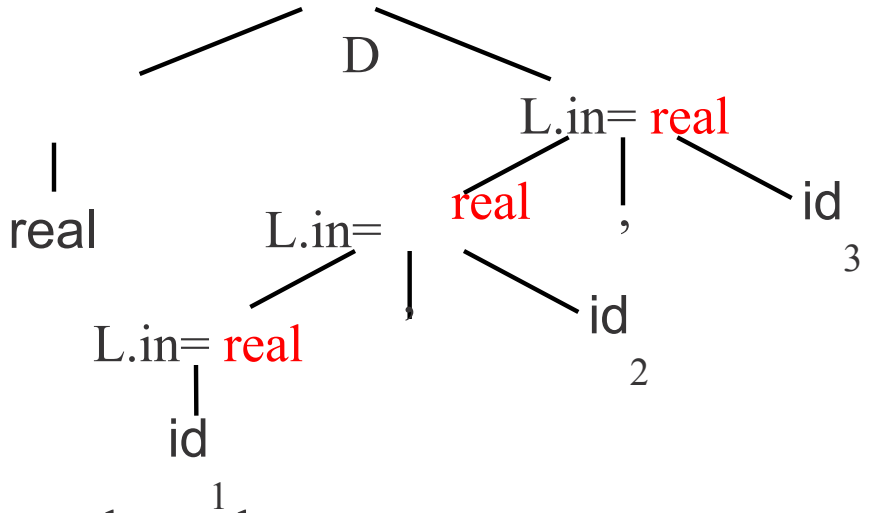
T .type = real



Inherited Attributes: An Example (Cont.)

- Synthesized attributes can be evaluated by a PostOrder traversal.
- Inherited attributes that *do not depend from right children* can be evaluated by a classical PreOrder traversal.
- The annotated parse-tree for the input real id_1 , id_2, id_3 is:

T .type = real



- L.in is then inherited top-down the tree by the other L-nodes.
- At each L-node the procedure *addtype* inserts into the symbol table the type of the identifier.

Summary

- Syntax Directed Translations
- Syntax Directed Definitions
- **Implementing Syntax Directed Definitions**
  - **Dependency Graphs**
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes

Dependency Graphs

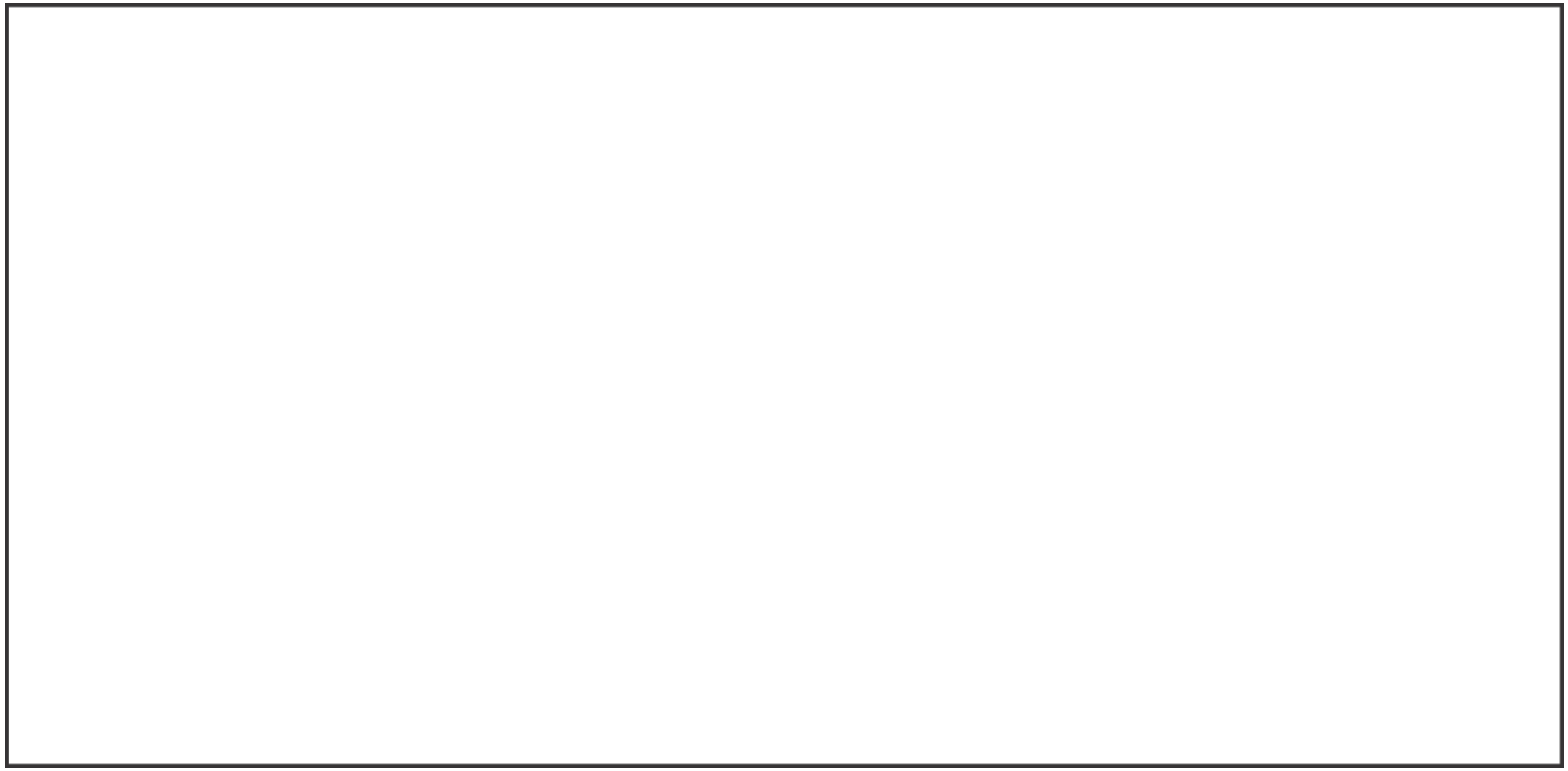
- Implementing a Syntax Directed Definition consists primarily in finding an order for the evaluation of attributes
  - Each attribute value must be available when a computation is performed.
- **Dependency Graphs** are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.
- A Dependency Graph shows the interdependencies among the attributes of the various nodes of a parse-tree.
  - There is a node for each attribute;
  - If attribute b depends on an attribute c there is a link from the node for c to the node for b (b ← c).
- **Dependency Rule:** If an attribute b depends from an attribute c, then we need to fire the semantic rule for c first and then the semantic rule for b.

Evaluation Order

- The evaluation order of semantic rules depends from a *Topological Sort* derived from the dependency graph.
- **Topological Sort:** Any ordering $m_1, m_2, \ldots, m_k$ such that if $m_i \rightarrow m_j$ is a link in the dependency graph then $m_i < m_j$.
- Any topological sort of a dependency graph gives a valid order to evaluate the semantic rules.

Dependency Graphs: An Example

- **Example.** Build the dependency graph for the parse-tree of real id_1, id_2, id_3.



Implementing Attribute Evaluation: General Remarks

- Attributes can be evaluated by building a dependency graph at compile-time and then finding a topological sort.
- **Disavantages**

1. This method fails if the dependency graph has a cycle: We need a test for non-circularity;
2. This method is time consuming due to the construction of the dependency graph.

- **Alternative Approach.** Design the syntax directed definition in such a way that attributes can be evaluated with a *fixed order* avoiding to build the dependency graph (method followed by many compilers).

Strongly Non-Circular Syntax Directed Definitions

- **Strongly Non-Circular Syntax Directed Definitions.** Formalisms for which an attribute evaluation order can be fixed at compiler construction time.
  - They form a class that is less general than the class of non-circular definitions.
  - In the following we illustrate two kinds of strictly non-circular definitions: *S-Attributed* and *L-Attributed Definitions*.

Summary

- Syntax Directed Translations
- Syntax Directed Definitions
- **Implementing Syntax Directed Definitions**
  - Dependency Graphs
  - **S-Attributed Definitions**
  - L-Attributed Definitions
- Translation Schemes

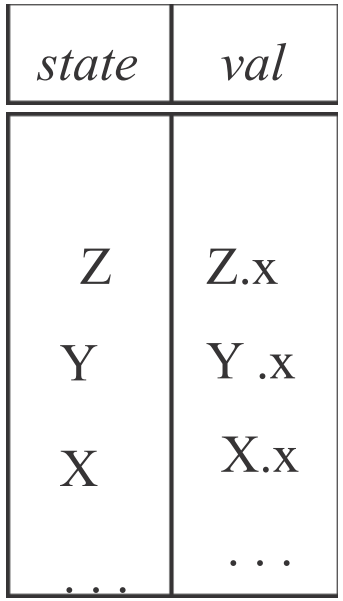
Evaluation of S-Attributed Definitions

- **Synthesized Attributes can be evaluated by a bottom-up parser as the input is being analyzed avoiding the construction of a dependency graph.**
- The parser keeps the values of the synthesized attributes in its stack.
- Whenever a reduction $A \rightarrow \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.
- Thus, a translator for an S-Attributed Definition can be simply implemented by extending the stack of an LR-Parser.

Extending a Parser Stack

- Extra fields are added to the stack to hold the values of synthesized attributes.

- In the simple case of just one attribute per grammar symbol the stack has two fields: *state* and *val*



- The current top of the stack is indicated by the pointer *top*.

- Synthesized attributes are computed just before each reduction:

– Before the reduction A → X Y Z is made, the attribute for A is computed: A.a := f (val[top], val[top − 1], val[top − 2]).

Extending a Parser Stack: An Example

- **Example.** Consider the S-attributed definitions for the arithmetic expressions. To evaluate attributes the parser executes the following code

PRODUCTION	CODE
L → En	*print*(val[top − 1])
$E \rightarrow E_1 + T$	val[ntop] := val[top] + val[top − 2]
E → T	
$T \rightarrow T_1 * F$	val[ntop] := val[top] * val[top − 2]
T → F	
F → (E)	val[ntop] := val[top − 1]
F → digit	

- The variable ntop is set to the *new top of the stack*. After a reduction is done *top* is set to *ntop*: When a reduction $A \rightarrow \alpha$ is done with $|\alpha| = r$, then ntop = top − r + 1.
- During a shift action both the token and its value are pushed into the stack.

Extending a Parser Stack: An Example (Cont.)

- The following Figure shows the moves made by the parser on input 3*5+4n.

– Stack states are replaced by their corresponding grammar symbol;

– Instead of the token digit the actual value is shown.

INPUT	*state*	*val*	PRODUCTION USED
3*5+4n	_	_	
*5+4n	3	3	
*5+4n	*F*	3	*F* → **digit**
*5+4n	*T*	3	*T* → *F*
5+4n	*T* *	3 _	
+4n	*T* * 5	3 _ 5	
+4n	*T* * *F*	3 _ 5	*F* → **digit**
+4n	*T*	15	*T* → *T* * *F*
+4n	*E*	15	*E* → *T*
4n	*E* +	15 _	
n	*E* + 4	15 _ 4	
n	*E* + *F*	15 _ 4	*F* → **digit**
n	*E* + *T*	15 _ 4	*T* → *F*
n	*E*	19	*E* → *E* + *T*
	E n	19 _	
	L	19	*L* → *E* n

Summary

- Syntax Directed Translations
- Syntax Directed Definitions
- **Implementing Syntax Directed Definitions**
  - Dependency Graphs
  - S-Attributed Definitions
  - **L-Attributed Definitions**
- Translation Schemes

L-Attributed Definitions

- **L-Attributed Definitions** contain both synthesized and inherited attributes

but do not need to build a dependency graph to evaluate them.

- **Definition.** A syntax directed definition is *L-Attributed* if each *inherited*

attribute of X_j in a production $A \rightarrow X_1 \ldots X_j \ldots X_n$, depends only on:

1. The attributes of the symbols to the **left** (this is what L in *L-Attributed* stands for) of X_j, i.e., $X_1 X_2 \ldots X_{j-1}$, and
2. The *inherited* attributes of A.

- **Theorem.** Inherited attributes in L-Attributed Definitions can be computed

by a PreOrder traversal of the parse-tree.

Evaluating L-Attributed Definitions

- **L-Attributed Definitions are a class of syntax directed definitions whose attributes can always be evaluated by single traversal of the parse-tree.**

- The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

Algorithm: L-Eval(n: Node)

Input: Node of an annotated parse-tree.

Output: Attribute evaluation.

```
Begin
        For each child m of n, from left-to-right    Do
        Begin
        Evaluate inherited attributes of m;
        L-Eval(m)
        End;
        Evaluate synthesized attributes of n
End.
```

Summary

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- **Translation Schemes**

Translation Schemes

- **Translation Schemes** are more implementation oriented than syntax directed definitions since they **indicate the order** in which semantic rules and attributes are to be evaluated.
- **Definition.** A Translation Scheme is a context-free grammar in which

1. Attributes are associated with grammar symbols;
2. Semantic Actions are enclosed between braces {} and **are inserted within the right-hand side of productions.**

- Yacc uses Translation Schemes.

Translation Schemes (Cont.)

- Translation Schemes deal with both synthesized and inherited attributes.
- Semantic Actions are treated as terminal symbols: Annotated parse-trees contain semantic actions as children of the node standing for the correspond- ing production.
- Translation Schemes are useful to evaluate L-Attributed definitions at parsing time (even if they are a general mechanism).
  - **An L-Attributed Syntax-Directed Definition can be turned into a Translation Scheme.**

Translation Schemes: An Example

- Consider the Translation Scheme for the L-Attributed Definition for "type declarations":

$D \rightarrow$ T {L.in := T .type} L

$T \rightarrow$ int {T .type :=*integer*}

$T \rightarrow$ real {T .type :=*real*}

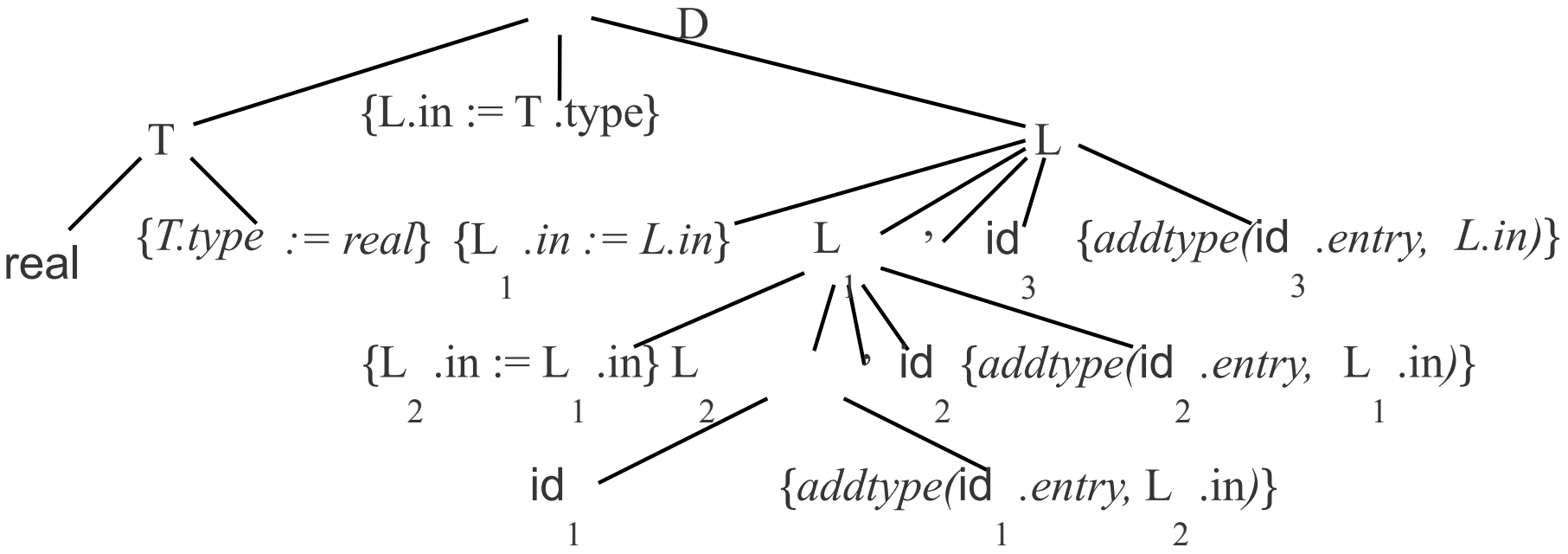
$L \rightarrow$ {L_1 .in := L.in} L_1 , id {*addtype(*id.*entry, L.in)*}

$L \rightarrow$ id {*addtype(*id.*entry, L.in)*}

Translation Schemes: An Example (Cont.)

- **Example (Cont).** The parse-tree with semantic actions for the input

real id_1, id_2, id_3 is:



- **Traversing the Parse-Tree in depth-first order (PostOrder) we can evaluate the attributes.**

Design of Translation Schemes

- When designing a Translation Scheme we must be sure that an attribute value is available when a semantic action is executed.
- **When the semantic action involves only synthesized attributes: The action can be put at the end of the production.**

– **Example.** The following Production and Semantic Rule:

$$T \rightarrow T_1 * F \qquad T.val := T_1.val * F.val$$

yield the translation scheme:

$$T \rightarrow T_1 * F \qquad \{T.val := T_1.val * F.val\}$$

Design of Translation Schemes (Cont.)

- **Rules for Implementing L-Attributed SDD's.** If we have an L-Attibuted Syntax-Directed Definition we must enforce the following restrictions:

1. An inherited attribute for a symbol in the right-hand side of a production must be computed in an action before the symbol;
2. A synthesized attribute for the non terminal on the left-hand side can only be computed when all the attributes it references have been computed: The action is usually put at the end of the production.

Compile-Time Evaluation of Translation Schemes

- Attributes in a Translation Scheme following the above rules can be computed at compile time similarly to the evaluation of S-Attributed Definitions.
- **Main Idea.** Starting from a Translation Scheme (with embedded actions) we introduce a transformation that makes all the actions occur at the right ends of their productions.

– For each embedded semantic action we introduce a new *Marker* (i.e., a non terminal, say M) with an empty production (M → ϱ);

  – The semantic action is attached at the end of the production M → ϱ.

Compile-Time Evaluation of Translation Schemes (Cont.)

- **Example.** Consider the following translation scheme:

$S \rightarrow aA\{C.i = f(A.s)\}C$

$S \rightarrow bAB\{C.i = f(A.s)\}C$

$C \rightarrow c\{C.s = g(C.i)\}$

Then, we add new markers M_1, M_2 with:

$S \rightarrow aAM_1C$

$S \rightarrow bABM_2C$

$M_1 \rightarrow \varepsilon$ $\quad\{M_1.s := f(val[top])\}$

$M_2 \rightarrow \varepsilon$ $\quad\{M_2.s := f(val[top-1])\}$

$C \rightarrow c$ $\quad\{C.s := g(val[top-1])\}$

The inherited attribute of C is the synthesized attribute of either M_1 or M_2.

The value of C.i is *always* in *val*[*top -1*] when $C \rightarrow c$ is applied.

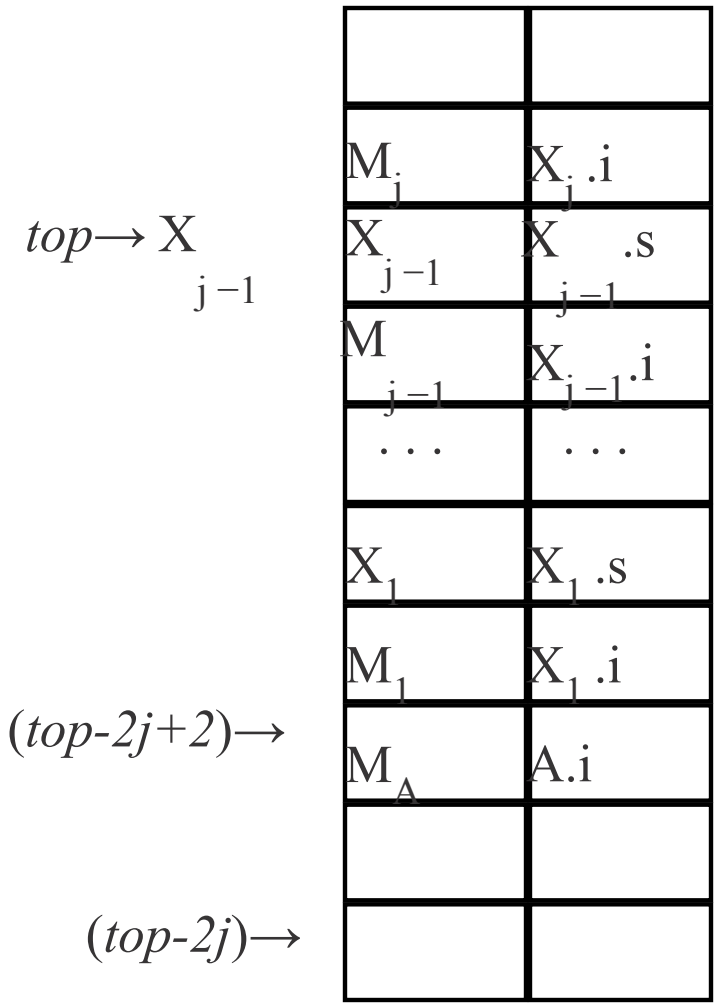
Compile-Time Evaluation of Translation Schemes (Cont.)

General rules to compute translations schemes during bottom-up parsing assuming an L-attributed grammar.

- For every production $A \to X_1 \ldots X_n$ introduce n new markers $M_1, \ldots, M_n$ and replace the production by $A \to M_1 X_1 \ldots M_n X_n$.
- Thus, we know the position of every synthesized and inherited attribute of X_j and A:
  1. $X_j.s$ is stored in the *val* entry in the parser stack associated with X_j;
  2. $X_j.i$ is stored in the *val* entry in the parser stack associated with M_j;
  3. A.i is stored in the *val* entry in the parser stack immediately before the position storing M_1.
- **Remark 1.** Since there is only one production for each marker a grammar remains LL(1) with addition of markers.
- **Remark 2.** Adding markers to an LR(1) Grammar can introduce conflicts for not L-Attributed SDD's!!!

Compile-Time Evaluation of Translation Schemes (Cont.)

Example. Computing the inherited attribute X_j.i after reducing with $M_j \rightarrow Q$.



- A.i is in val[top − 2j + 2];
- X_j.i is in val[top − 2j + 3];
- X_1.s is in val[top − 2j + 4];
- X_2.i is in val[top − 2j + 5];
- and so on.

Summary

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes

Boolean Expression

- The translation of if-else-statements and while-statements is tied to the translation of boolean expressions.
- In programming languages, boolean expressions are used to
  - *1. Alter the flow of control.*
  - *2. Compute logical values.*

Boolean Expression

1. Alter the flow of control.

- Boolean expressions are used as conditional expressions in statements that alter the flow of control.
- The value of such boolean expressions is implicit in a position reached in a program.

For example, in **if** (E) *S,* the expression *E* must be true if statement S is reached.

- *2. Compute logical values.*
  - A boolean expression can represent *true* or *false* as values.
  - Boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

Boolean Expression

- The intended use of boolean expressions is determined by its syntactic context.
- For example, an expression following the keyword if is used to alter the flow of control, while an expression on the right side of an assignment is used to denote a logical value.
- Such syntactic contexts can be specified in a number of ways:
  - we may use two different nonterminals, use inherited attributes, or set a flag during parsing.
  - Alternatively we may build a syntax tree and invoke different procedures for the two different uses of boolean expressions.

Boolean expressions

- Focus on the use of boolean expressions to alter the flow of control
- Boolean expressions are composed of the boolean operators
  - && - AND, I I - OR, and ! - NOT
  - applied to elements that are boolean variables or relational expressions.
  - Relational expressions are of the form El **re1** *E2,* where *El* and E2 are arithmetic expressions
- Grammar to generate Boolean expression:

$B \rightarrow B_1$ or B_2 | $B \rightarrow B_1$ and B_2 | $B \rightarrow$ not B_1 | $B \rightarrow (B_1)$ | $B \rightarrow E_1$ relop E_2 |

$B \rightarrow$ false | $B \rightarrow$ true

Boolean expressions

- Semantic definition of the programming language determines
  - whether all parts of a boolean expression must be evaluated.
- If the language definition permits (or requires) portions of a boolean expression to go unevaluated
  - the compiler can optimize the evaluation of boolean expressions by computing only enough of an expression to determine its value.
- In an expression such as B1 I I B2, neither B1 nor B2 is necessarily evaluated fully.
  - If we determine that B1 is true, then we can conclude that the entire expression is true without having to evaluate B2.
- Similarly, given B1&&B2, if B1 is false, then the entire expression is false.

Short-Circuit Code

- In short-circuit (or jumping) code, the boolean operators &&, I I , and ! translate into jumps.
- The operators themselves do not appear in the code
- Instead, the value of a boolean expression is represented by a position in the code sequence.

Short-Circuit Code

- The statement

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

might be translated into the code

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

- In this translation, the Boolean expression is true if control reaches label L2.
- If the expression is false, control goes immediately to L1, skipping L2 and the assignment x = 0.

Flow-of-Control Statements

- Translation of boolean expressions into three-address code in the context of statements generated using the following grammar

  S -> if (B) S1

  S -> if (B) S1 else S2

  S -> while (B) S1

- Non terminal B represents a boolean expression and nonterminal S represents a statement

Boolean Expression

- Synthesized attributes associated with Boolean expression and Statement
  - B.code ,S.code- gives the translation into three-address instructions
- Inherited attributes associated with Boolean expression and Statement
  - B.true - label to which control flows if *B* is true
  - *B.false* - label to which control flows if *B* is false
  - *S.next* - *label* attached to the first 3-address statement to be executed immediately after the code for *S.*
- Functions used in Syntax directed definitions
- *newlabel()* creates a new label each time it is called
- *label(L)* attaches label *L* to the next three-address instruction to be generated

Syntax directed definition for P □ S

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$

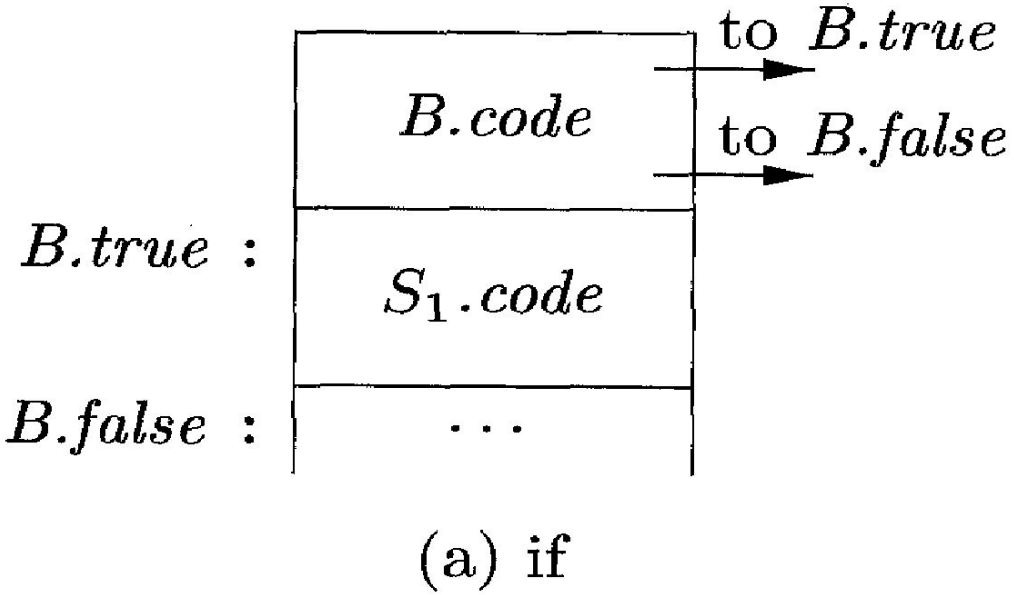
- **P-> S** : Program **P** consist of statement **S**
- **S.next** is initialized to to a new label
- **P.code** consists of **S.code** followed by the new label **S.next**
- **assign** is a placeholder for assignment statements

Syntax directed definition for S □ S_1S_2

$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

- S1 followed by the code for S2
- The first instruction after the code for S1 is the beginning of the code for S2
- The instruction after the code for S2 is also the instruction after the code for S.

Syntax directed definition for Boolean Expression **S -> if (B) S1**



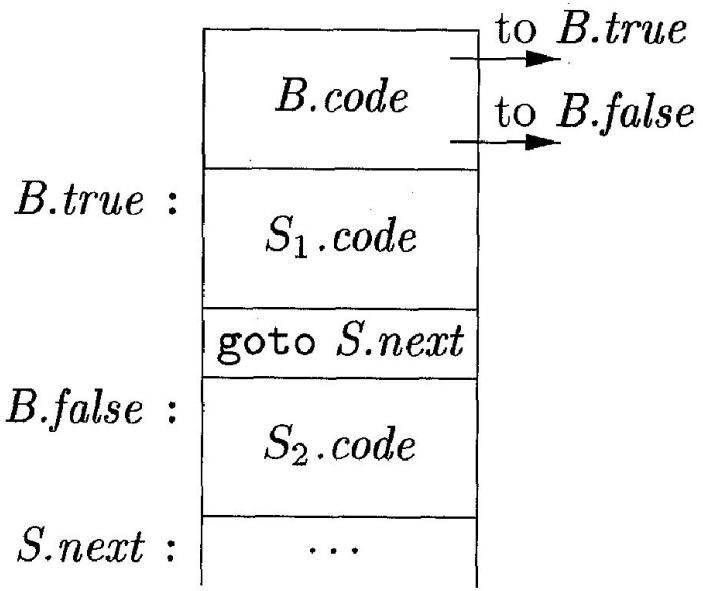
(a) if

Syntax-directed definition

$S \rightarrow \text{if}\,(B)\,S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \,\|\|\, label(B.true) \,\|\|\, S_1.code$

- Create a new label B.true and attach it to the first three-address instruction generated for the statement S1
- Within B. code are jumps based on the value of B.
- If B is true, control flows to the first instruction of S1 .code, and
- If B is false, control flows to the instruction immediately following S1.code.
- By setting B.false to S.next, we ensure that control will skip the code for S1 if B evaluates to false.

Syntax directed definition for **Boolean Expression**
S -> if (B) S1 else S2



(b) if-else

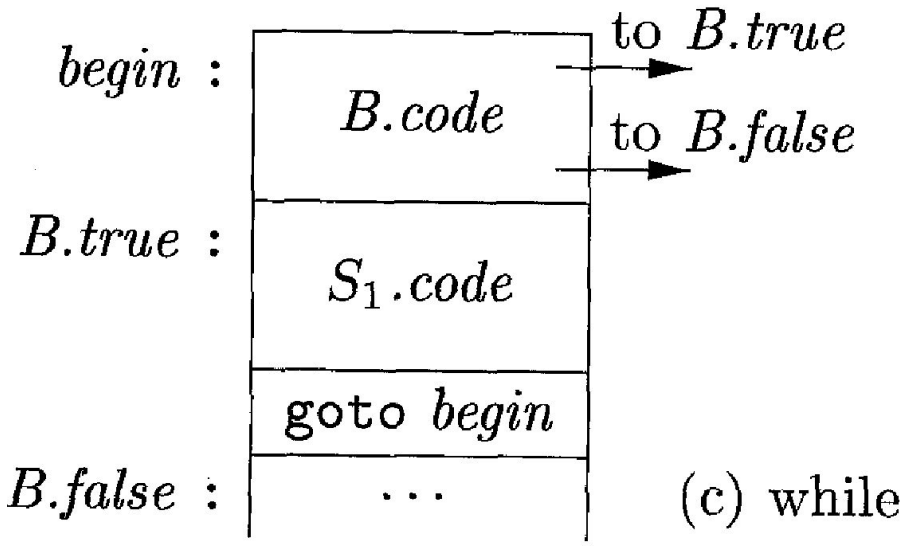
Syntax-directed definition

$S \rightarrow \textbf{if}\ (\ B\ )\ S_1\ \textbf{else}\ S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\|\| label(B.true) \|\| S_1.code$ $\|\| gen('\texttt{goto}'\ S.next)$ $\|\| label(B.false) \|\| S_2.code$

- Create two labels B.true and B.false
- If B is true, B jumps to the first instruction of the code for S1
- If B is false, jumps to the first instruction of the code for S2
- Control flows from both S1 and S2 to the three-address instruction immediately following the code for S - its label is given by the inherited attribute S.next.
- An explicit goto S.next appears after the code for S1 to skip over the code for S2.
- No goto is needed after S2, since S2. next is the same as S. next.

Boolean Expression S □ while (B) S1

Syntax-directed definition



(c) while

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

- begin – local variable that holds a new label attached to the first instruction for the while-statement, also the first instruction for B.
- begin- is a variable rather than an attribute, because begin is local to the semantic rules for this production.
- S.next - marks the instruction that control must flow to if B is false; hence, B. false is set to be S.next
- B. true - Code for B generates a jump to this label if B is true and attached to the first instruction for S1
- goto begin - causes a jump back to the beginning of the code for the boolean expression.
- S1 .next is set to this label begin, so jumps from within S1. code can go directly to begin

Control-Flow Translation of Boolean Expressions

- Boolean expression B is translated into three-address instructions
- B is evaluated using conditional and unconditional jumps to one of two labels:
  - *B.true* - *if B* is true
  - *B.fa1se* - if *B* is false

Control-Flow Translation of Boolean Expressions

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$

- If *B1* is true, then *B* itself is true, so *B1.true* is the same as *B.true*.
- If *B1* is false, then *B2* must be evaluated, so make *B1.false* be the label of the first instruction in the code for *B2.*
- The true and false exits of *B2* are the same as the true and false exits of *B,* respectively.

Control-Flow Translation of Boolean Expressions

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.true) \;\|\|\; B_2.code$

- If *B1* is true, then *B2* must be evaluated so make *B1.true* is assigned as the label of the first instruction in the code for *B2*
- If *B1* is false, then *B* itself is false, so *B1.false* is the same as *B.false*
- If *B2* is true, then *B* itself is true, so *B2.true* is the same as *B.true*
- If *B2* is false, then *B* itself is false, so *B2.false* is the same as *B.false*

Control-Flow Translation of Boolean Expressions

PRODUCTION	SEMANTIC RULES
$B \rightarrow\ !\ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$

- No code is needed for an expression B of the form ! B1
- Interchange the true and false exits of B to get the true and false exits of B1.

Control-Flow Translation of Boolean Expressions

PRODUCTION	SEMANTIC RULES
$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \ \|\| \ E_2.code$ $\|\| \ gen(\text{'}\texttt{if}\text{'} \ E_1.addr \ \mathbf{rel}.op \ E_2.addr \ \text{'}\texttt{goto}\text{'} \ B.true)$ $\|\| \ gen(\text{'}\texttt{goto}\text{'} \ B.false)$

- Translated directly into a comparison three-address instruction with jumps to the appropriate places
- For instance, *B* of the form *a* < b translates into

```
if a < b goto B.true
goto B.false
```

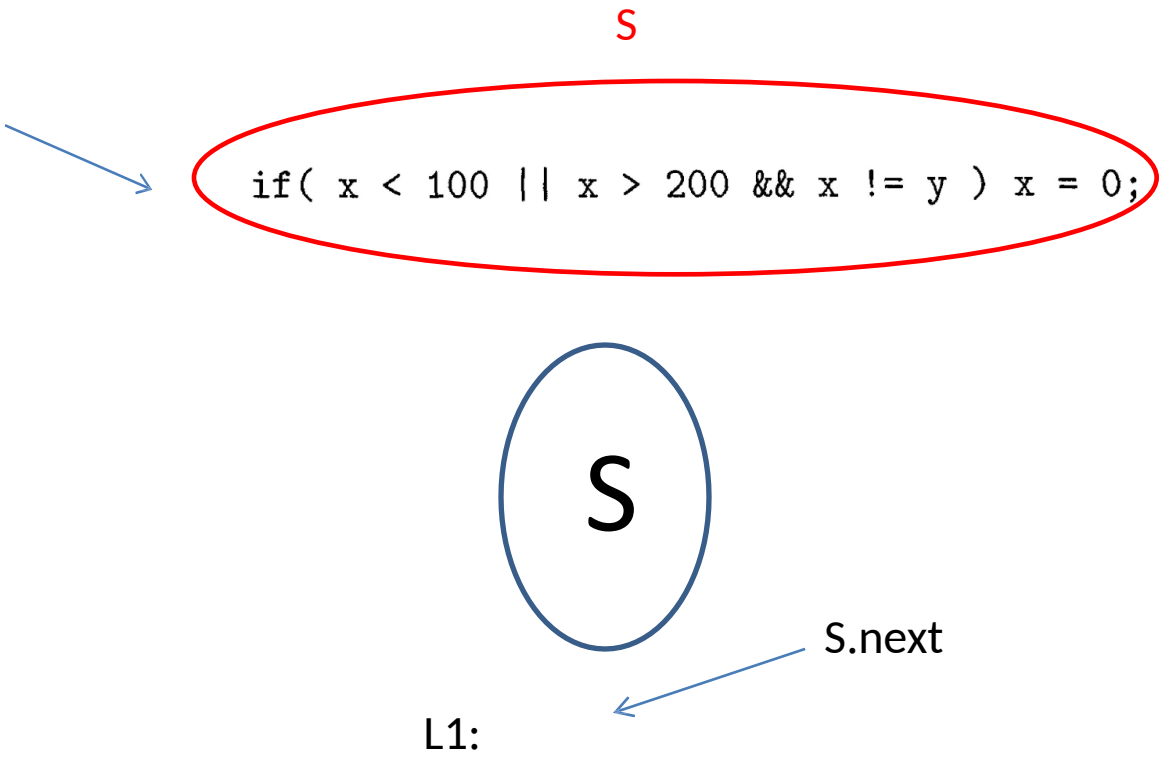
Control-Flow Translation of Boolean Expressions

PRODUCTION	SEMANTIC RULES
$B \rightarrow \textbf{true}$	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow \textbf{false}$	$B.code = gen('\texttt{goto}'\ B.false)$

The constants **true** and **false** translate into jumps to **B.true** and **B.false**, respectively

Control-Flow Translation of Boolean Expressions -Example

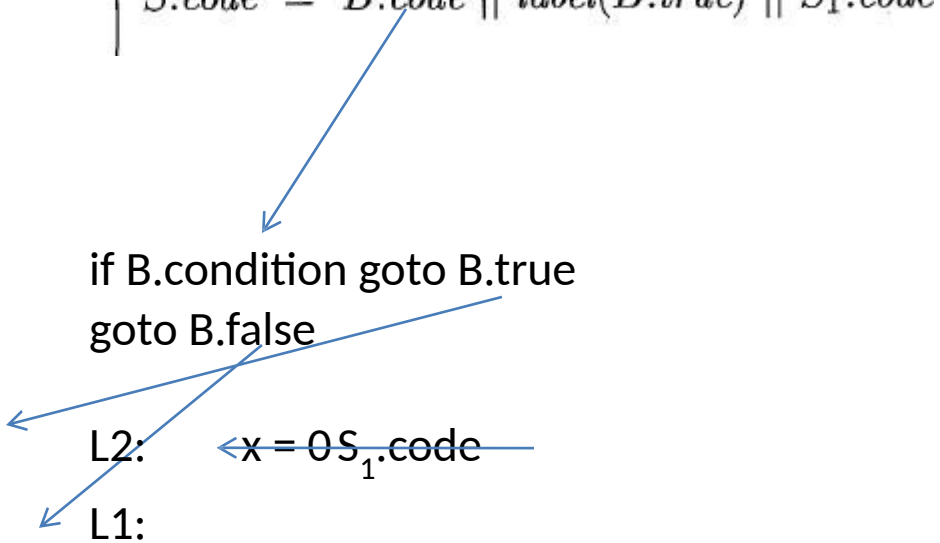
Consider the expression



B S_1

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

$S \rightarrow \textbf{if}\ (\ B\ )\ S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$



B_1 B_2

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

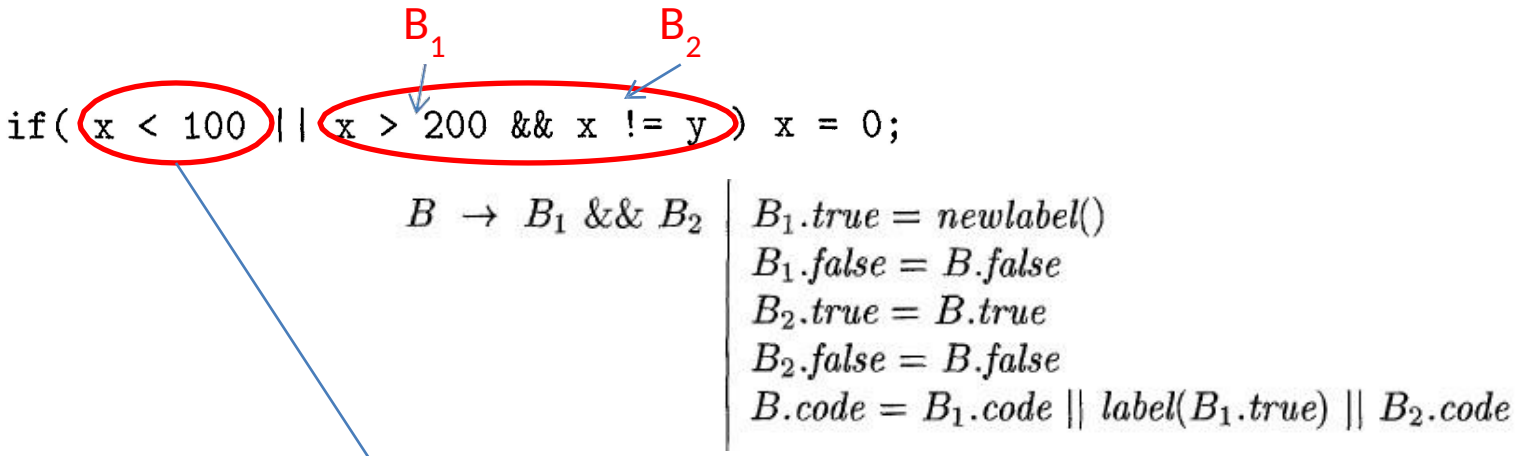
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$

If B1.condition goto L2
goto L3
L3:If B2.condition goto L2
goto L1

B.true → L2: x = 0

B.false → L1:

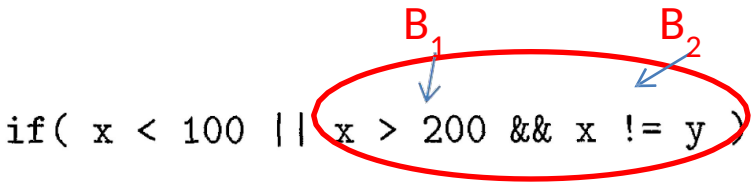
B_1.true = B.true = L2
B_1.false = L3
B_2.true = B.true = L2
B_2.false = B.false = L1



If x< 100 goto L2
goto L3
L3: If B2.condition goto L2
goto L1

L2: x = 0

L1:



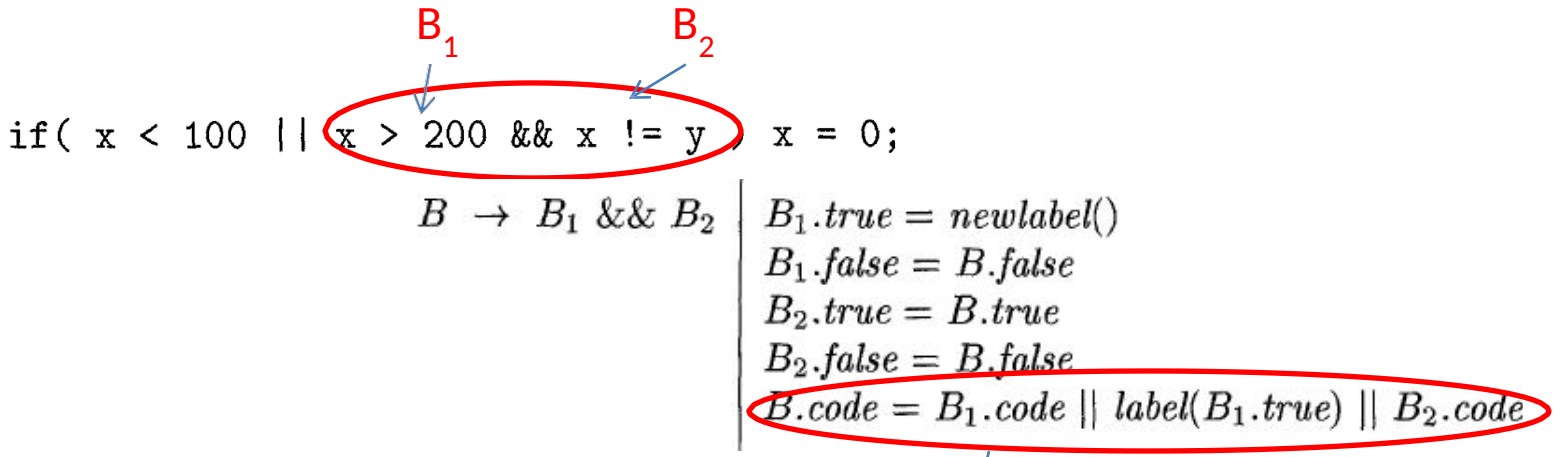
$B \rightarrow B_1 \;\&\&\; B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \;\|\|\; label(B_1.true) \;\|\|\; B_2.code$

If X < 100 goto L2
goto L3
L3: If B2.condition goto L2
goto L1

L2: x = 0

L1:

B_1.true = L4
B_1.false = B.false = L1
B_2.true = B.true = L2
B_2.false = B.false = L1



Control-Flow Translation of the above statement is

```
      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:
```

B_1.true = L4
B_1.false = B.false = L1
B_2.true = B.true = L2
B_2.false = B.false = L1

If B1.condition goto L4
goto L1
L4: If B2.condition goto L2
goto L1

Avoiding Redundant Gotos

```
if x > 200 goto L4
goto L1
L4:   …
□
ifFalse x > 200 goto L1
L4:   …
```

(fall through)

- Translated code of Boolean expressions are not optimized ones
- Redundant gotos are noticed
- Can be avoided using fallthrough

Using Fall Through

We now adapt the semantic rules for boolean expressions to allow control to fall through whenever possible.

$S \rightarrow$ if (E) S_1

{*E.true* = fall; // not *newlabel*; *E.false* =

S.next;

S_1.*next* = *S.next*;

S.code = *E.code* || S_1.*code* }

Similarly, the rules for if-else- and while-statements also set E.true to fall.

Using Fall Through Cont'd

$E \rightarrow E_1 \,||\, E_2$

{E_1.*true* = if (*E.true = fall)* then *newlabel() else E.true*; E_1.*false* = *fall;*

E_2.*true* = *E.true*; E_2.*false* = *E.false*;

E.code = if (E.true = fall) then E_1.*code* || E_2.*code* ||

label(E_1.true)

else E_1.*code* || E_2.*code* }

$E \rightarrow E_1 \,\&\&\, E_2$

{E_1.*true = fall;*

E_1.*false* = if (*E*.false = *fall)* then *newlabel() else E.false*;

E_2.*true* = *E.true*; E_2.*false* = *E.false*;

E.code = if (E.false = fall) then E_1.*code* || E_2.*code* ||

label(E_1.false)

else E_1.*code* || E_2.*code* }

Using Fall Through Cont'd

$E \rightarrow E_1$ relop E_2

{test = E_1.addr relop E_2.addr

s = if E.true != fall and E.false != fall then

gen('if' test 'goto', *E.true*) || *gen*('goto', *E.false*) else if (E.true != fall) then *gen*('if' test 'goto', *E.true*)

else if (E.false != fall) then *gen*('ifFalse' test 'goto', *E*.false)

else ''

E.code := E_1.code || E_2 .code || s

}

Using Fall Through Example

```
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:
```

Problems to be solved

- Translate the following expressions with and without fall through
  - If (a==b && c==d || e==f) x==1;
  - If (a==b || c==d || e==f) x==1;
  - If (a==b && c==d && e==f) x==1;
  - If a < b or c < d and e < f
  -
```
while a < b do
    if c<d then
        x=y+z
    else
        x=y-z
```

Backpatching

- A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump.
- For example, the translation of the boolean expression B in if (B) *S* contains a jump, for when B is false, to the instruction following the code for S.
- In a one-pass translation, B must be translated before S is examined.
- What then is the target of the goto that jumps over the code for S?

Backpatching

- For the examples of the previous lectures for implementing syntax-directed definitions, the easiest way is to use two passes.
  - First syntax tree is constructed
  - Then traversed in depth-first order to compute the translations given in the definition.
- Problem in generating three address codes in a single pass for Boolean expressions and flow of control statements is :
  - We may not know the labels that control must go to at the time jump statements are generated.

Backpatching

- This problem is solved by generating a series of branch statements with the targets of the jumps temporarily left unspecified.
- Each such statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined.
- This subsequent filling of addresses for the determined labels is called BACKPATCHING.

Backpatching

- For implementing Backpatching
  - Instructions are generated into an instruction array
  - Labels act as indices to this array.
- To manipulate list of labels , three functions are used:
  - makelist(i)
  - merge(p_1,p_2) and
  - backpatch(p,i)

Backpatching

- makelist(i) : creates a new list containing only i, an index into the array of instructions and returns pointer to the list it has made.
- merge(i,j) – concatenates the lists pointed to by i and j ,and returns a pointer to the concatenated list.
- backpatch(p,i) – inserts i as the target label for each of the statements on the list pointed to by p.

Backpatching

- Let's now try to construct the translation scheme for Boolean expression.
- Let the grammar be:

$B \rightarrow B_1$ or MB_2

$B \rightarrow B_1$ and MB_2

$B \rightarrow$ not B_1

$B \rightarrow (B_1)$

$B \rightarrow id_1$ relop id_2

$B \rightarrow$ false

$B \rightarrow$ true

$M \rightarrow \varepsilon$

Backpatching

- What we have done is inserting a marker non-terminal M into the grammar to cause a semantic action to pick up, at appropriate times the index of the next instructionto be generated.
- This is done by the semantic action:

  { M.instr = nextinstr; } for the rule $M \rightarrow \varepsilon$

Backpatching

- Two synthesized attributes truelist and falselist of non-terminal B are used to generate jumping code for Boolean expressions.
- B.truelist : Contains the list of all the jump statements left incomplete to be filled by the label for the start of the code for B=true.
- B.falselist : Contains the list of all the jump statements left incomplete to be filled by the label for the start of the code for B=false.

Backpatching

- The variable nextinstr holds the index of the next instruction to follow.
- This value will be backpatched onto B_1.truelist in case of $B \rightarrow B_1$ and MB_2 where it contains the address of the first statement of B_2.code.
- This value will be backpatched onto B_1.falselist in case of $B \rightarrow B_1$ or MB_2 where it contains the address of the first statement of B_2.code.

Backpatching-Boolean expressions

- We use the following semantic actions for the above grammar :
- 1) $B \rightarrow B_1$ or M B_2

  backpatch(B_1.falselist, M.instr)

  B.truelist = merge(B_1.truelist, B_2.truelist)

  B.falselist = B_2.falselist

- If ***B1*** is **true**, then **B** is also true, so the jumps on ***B1. truelist*** become part of ***B.truelist***.
- If ***B1*** is **false**, we must next test ***B2,*** so the target for the jumps **B1.falselist** must be the beginning of the code generated for **B2**.
- This target is obtained using the marker nonterminal ***M***.
- ***M*** produces, as a synthesized attribute **M.instr**, the index of the next instruction, just before **B2** code starts being generated
- The value **M.instr** will be backpatched onto the **B1 .falselist** (i.e., each instruction on the list ***B1.*** **falselist** will receive **M.instr** as its target label) when we have seen the remainder of the production **B -> B1 or M B2.**

Backpatching -Boolean expressions

2)$B \rightarrow B_1$ and M B_2

backpatch(B_1.truelist, M.instr)

B.truelist = B_2.truelist

B.falselist = merge(B_1.falselist, B_2.falselist)

- If ***B1*** is **true**, we must next test ***B2,*** so the target for the jumps **B1. truelist** must be the beginning of the code generated for **B2**.
- This target is obtained using the marker nonterminal **M.instr** , the index of the next instruction, just before **B2** code starts being generated
- The value **M.instr** will be backpatched onto the **B1.truelist.**
- If ***B1*** is **false**, then **B** is also **false**, so the jumps on ***B1. falselist*** become part of ***B.falselist***.

Backpatching -Boolean expressions

3) $B \rightarrow \text{not } B_1$

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) $B \rightarrow id_1 \text{ relop } id_2$

B.truelist = makelist(nextinstr)

B.falselist = makelist(nextinstr+1)

emit(if id_1.place relop id_2.place goto __)

emit(goto ___)

Swaps the true and false lists

Ignores parenthesis

- generates two instructions, a conditional goto and an unconditional one.
- Both gotos have unfilled targets
- These instructions are put on B.truelist and B.falselist, respectively

Backpatching-Boolean expressions

6) B → true

B.truelist = makelist(nextinstr)

emit(goto ___)

7) B → false

E.falselist = makelist(nextinstr)

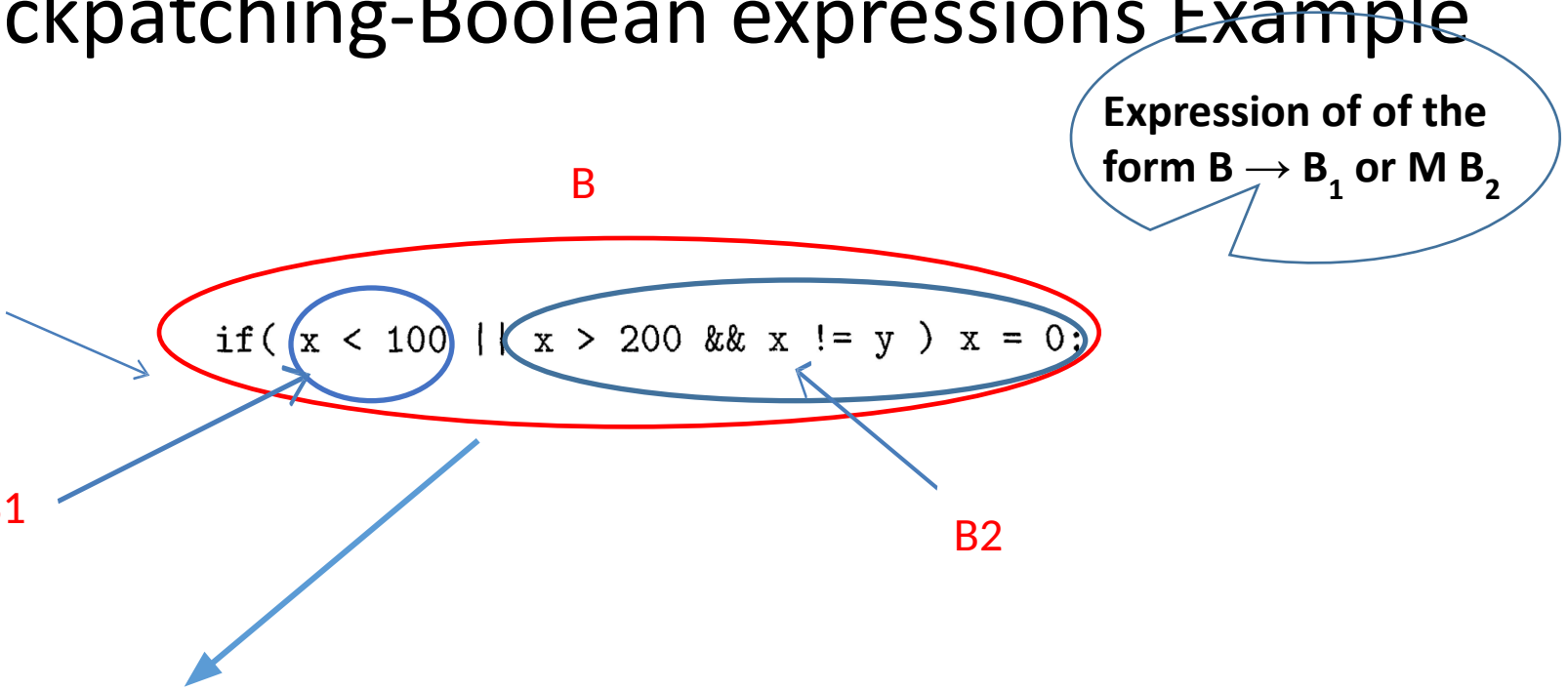
emit(goto ___)

8) M → ε

M.instr = nextinstr;

Backpatching-Boolean expressions Example

Consider the expression



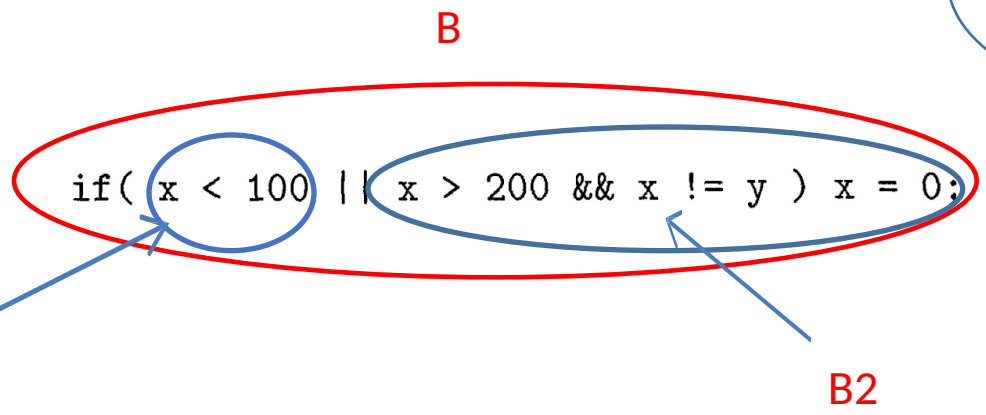
$B \rightarrow B_1$ or M B_2

backpatch(B_1.falselist, M.instr)

B.truelist = merge(B_1.truelist, B_2.truelist)

B.falselist = B_2.falselist

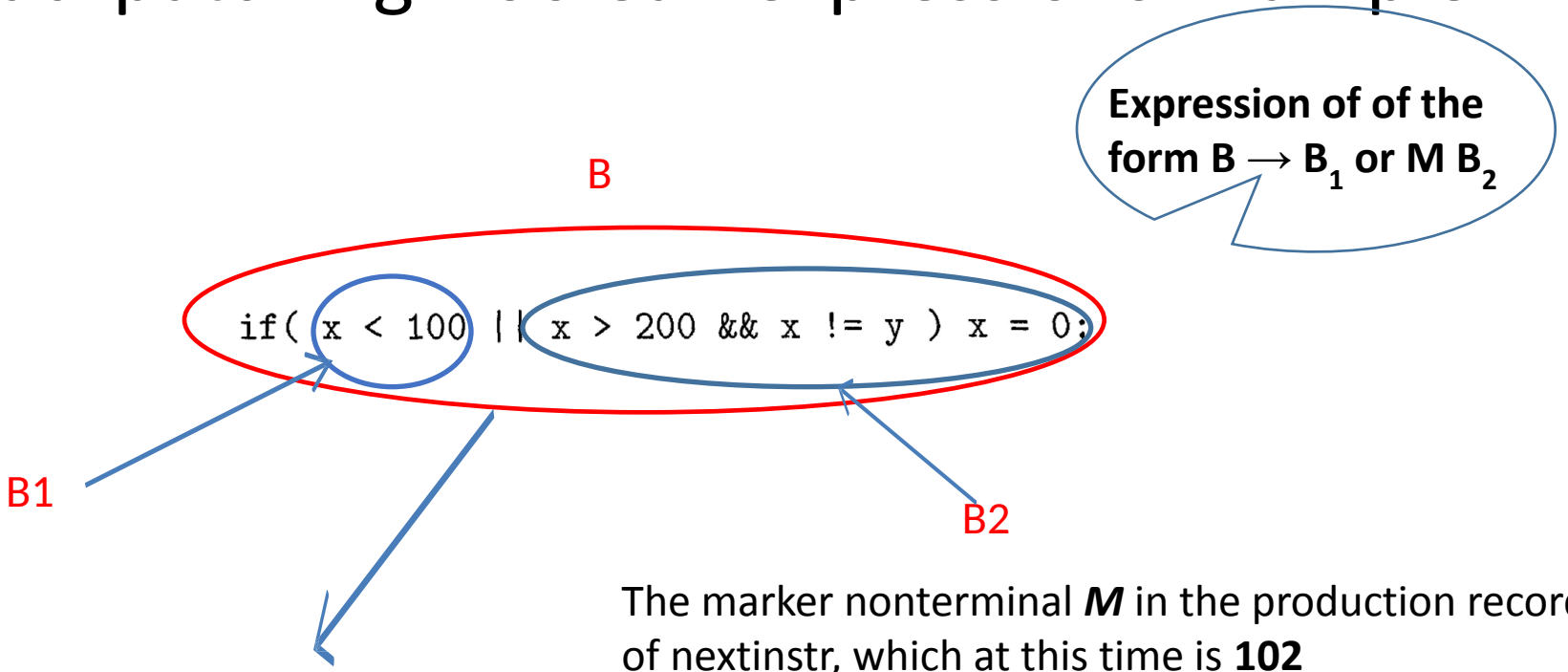
Backpatching-Boolean expressions Example



$B \rightarrow id_1$ relop id_2
B.truelist = makelist(nextinstr)
B.falselist = makelist(nextinstr+1)
emit(if id_1.place relop id_2.place goto __)
emit(goto ___)

```
100:   if x < 100 goto _
101:   goto _
```

Backpatching-Boolean expressions Example



The marker nonterminal ***M*** in the production records the value of nextinstr, which at this time is **102**

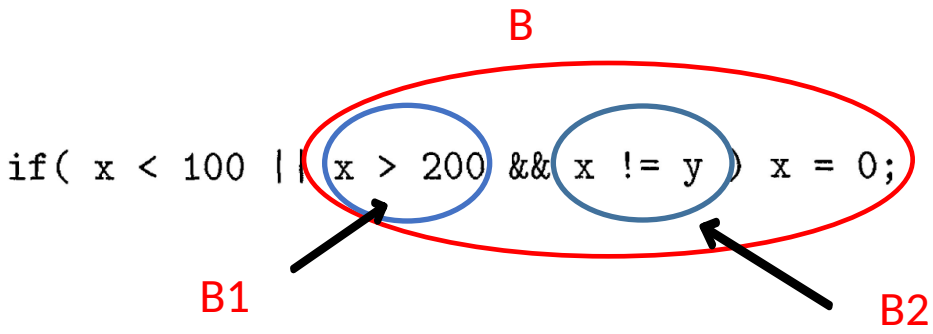
$B \rightarrow B_1$ or $M\ B_2$

backpatch(B_1.falselist, M.instr)

B.truelist = merge(B_1.truelist, B_2.truelist)

B.falselist = B_2.falselist

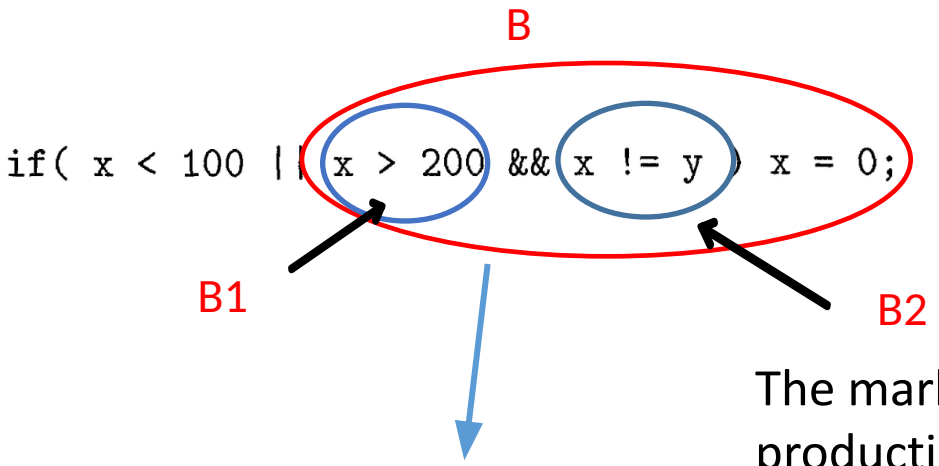
Backpatching-Boolean expressions Example



B → id_1 relop id_2
B.truelist = makelist(nextinstr)
B.falselist = makelist(nextinstr+1)
emit(if id_1.place relop id_2.place goto __)
emit(goto ___)

```
102:   if x > 200 goto _
103:   goto _
```

Backpatching-Boolean expressions Example



The marker nonterminal ***M*** in the production records the value of nextinstr, which at this time is **104**

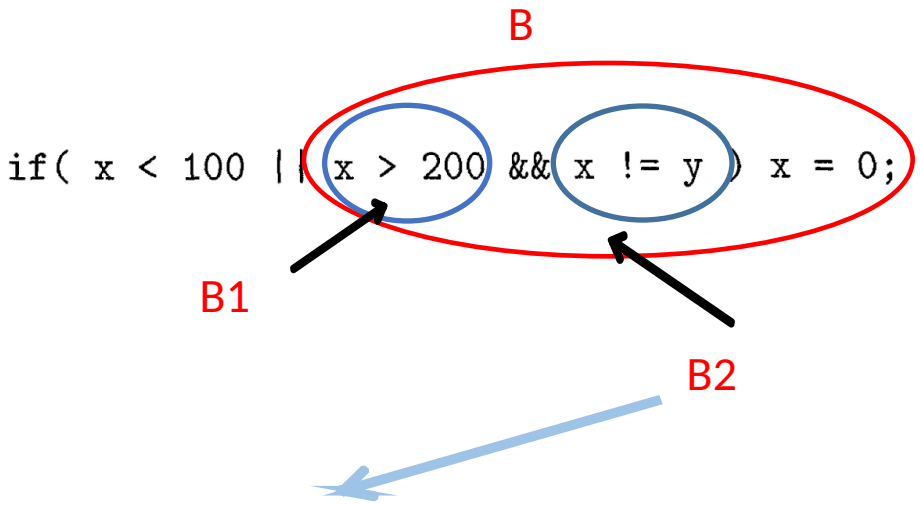
2)$B \rightarrow B_1$ and M B_2

backpatch(B_1.truelist, M.instr)

B.truelist = B_2.truelist

B.falselist = merge(B_1.falselist, B_2.falselist)

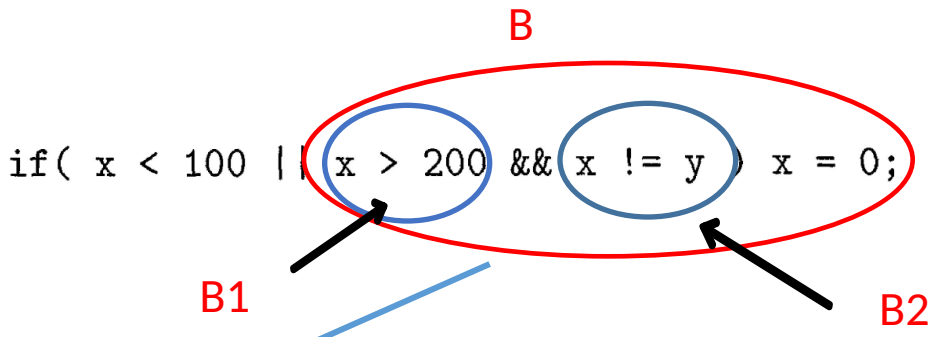
Backpatching-Boolean expressions Example



B → id_1 relop id_2
B.truelist = makelist(nextinstr)
B.falselist = makelist(nextinstr+1)
emit(if id_1.place relop id_2.place goto __)
emit(goto ___)

```
104:   if x != y goto _
105:   goto _
```

Backpatching -Boolean expressions Example



2)B $\rightarrow$ B_1 and M B_2

backpatch(B_1.truelist, M.instr)

B.truelist = B_2.truelist

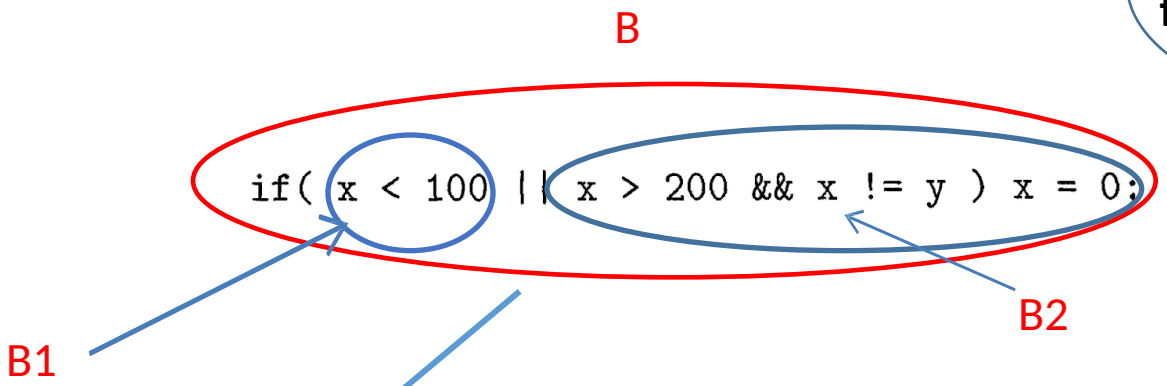
B.falselist = merge(B_1.falselist, B_2.falselist)

B_1.truelist =102
M.instr=104
backpatch(102,104) – instruction in 102 is filled with 104

```
100:   if x < 100 goto _
101:   goto
102:   if x > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

Backpatching-Boolean expressions Example

Expression of of the form $B \rightarrow B_1$ or $M\ B_2$



B_1.falselist =101
M.instr=102
backpatch(101,102) – instruction in 101 is filled with 102

$B \rightarrow B_1$ or $M\ B_2$
backpatch(B_1.falselist, M.instr)
B.truelist = merge(B_1.truelist, B_2.truelist)
B.falselist = B_2.falselist

```
100:   if x < 100 goto _
101:   goto 102
102:   if y > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

Backpatching-Boolean expressions Example

B

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

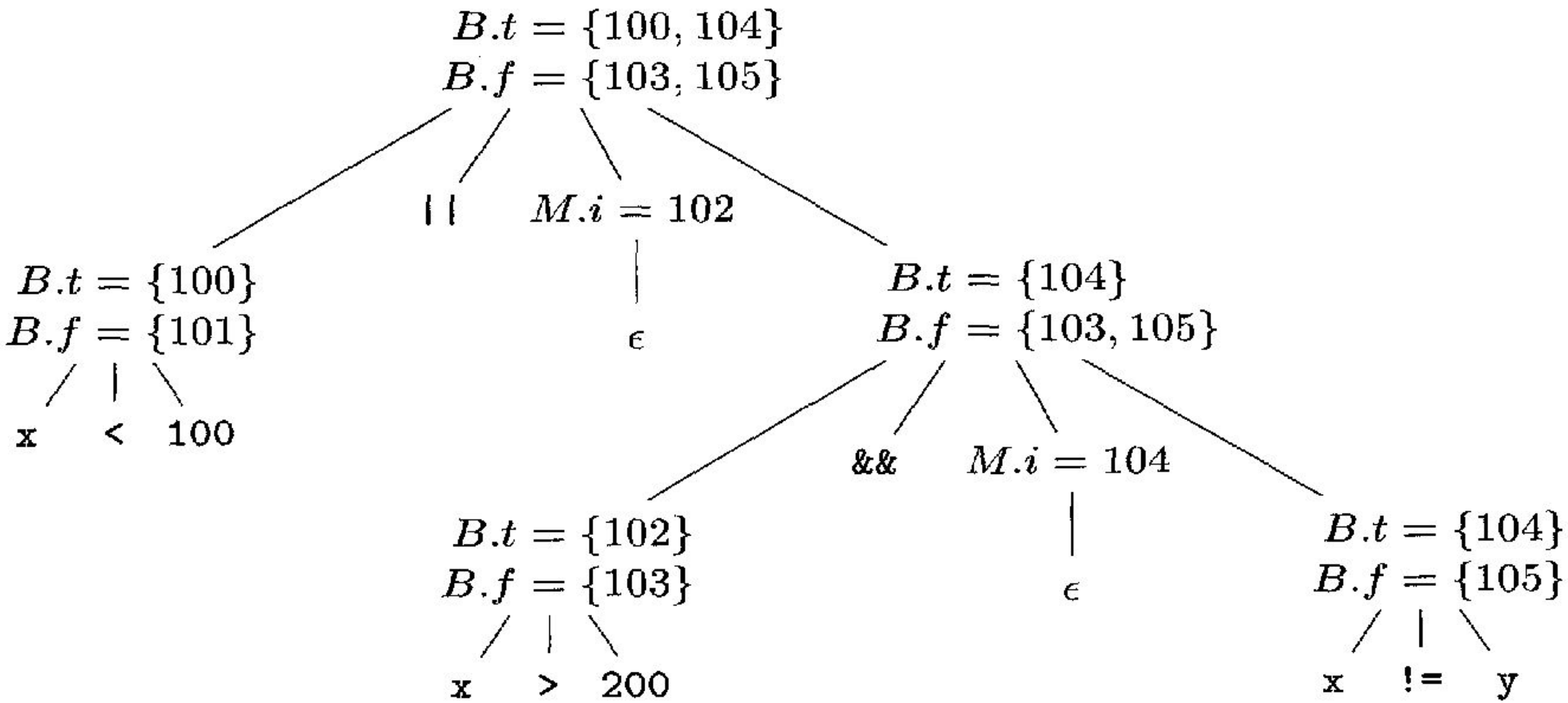
The **entire expression is true** if and only if the gotos of instructions **100** or **104** are reached

Expression is false if and only if the gotos of instructions **103** or **105** are reached.

```
100:   if x < 100 goto _
101:   goto 102
102:   if y > 200 goto 104
103:   goto _
104:   if x != y goto _
105:   goto _
```

Backpatching-Boolean expressions Example

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



Backpatching-Flow of control

- Backpatching can be used to translate flow-of-control statements in one pass.
- Consider statements generated by the following grammar:

$$S \rightarrow \textbf{if}\,(B)\,S \mid \textbf{if}\,(B)\,S\,\textbf{else}\,S \mid \textbf{while}\,(B)\,S \mid \{\,L\,\} \mid A\;;$$

$$L \rightarrow L\,S \mid S$$

- S denotes a statement
- L denotes statement list
- B denotes boolean expression

Backpatching-Flow of control

- Boolean expressions generated by nonterminal B have two lists of jumps
  - B. truelist -Contains the list of all the jump statements left incomplete to be filled by the label for the start of the code for B=true.
  - B.falselist-Contains the list of all the jump statements left incomplete to be filled by the label for the start of the code for B=false.
- Statements generated by nonterminals S and L have a list of unfilled jumps
  - Eventually filled by backpatching.
  - S.nextlist - list of all conditional and unconditional jumps to the instruction following the code for statement S in execution order.
  - L.nextlist - list of all conditional and unconditional jumps to the instruction following the code for statement L in execution order

Backpatching-Translation of Flow of control statements

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

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

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

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

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

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

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

Backpatching-Flow of control

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

- In the above production
  - Two occurrences of the marker nonterminal ***M***
  - Used to record the instruction numbers of the beginning of the code for **B** and the beginning of the code for **S1**
  - For the rule **M → ε** , semantic action is { **M.instr = nextinstr**; }
- After the body ***S1*** of the while-statement is executed, control flows to the beginning.
- When the entire production is reduced to **S,** ***S1.nextlist*** is backpatched to make all targets on that list be *M1.instr.*
- An explicit jump to the beginning of the code for *B* is appended after the code for ***S1*** because control may also "fall out the bottom."
- ***B.truelist*** is backpatched to go to the beginning of ***S1*** by making jumps on ***B. truelist*** go to ***M2 . instr.***

Backpatching-Flow of control

$$S \rightarrow \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$$

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

- If control "falls out the bottom" of S1, as when S1 is an assignment
  - code has to be included at the end of S1 to jump to S2
  - A new nonterminal N with production **N → ε** is introduced for this purpose
  - **N** has attribute **N.nextlist**, which will be a list consisting of the instruction number of the jump goto - that is generated by the semantic action (7) for N.
- When **B is true**, jumps are backpatched to the instruction **M1.instr**;
- Similarly, when **B is false** jumps are backpatched to the beginning of the code for **S2**.
- **S.nextlist** includes all jumps out of S1 and *S2,* as well as the jump generated by N.
- **temp** is a temporary variable that is used only for merging lists.

Backpatching-Flow of control

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

$$L \rightarrow S \qquad \{ \; L.nextlist \; = \; S.nextlist; \; \}$$

- Above two productions handle sequences of statements.
- In **L -> L_1 M S**
  - Instruction following the code for **L1** in order of execution is the beginning of **S.**
  - Thus the **L1 .nextlist** list is backpatched to the beginning of the code for **S**, which is given by **M. instr**.
- In **L -> S** , **L. nextlist** is the same as **S.nextlist**

Backpatching-Flow of control

- No new instructions are generated anywhere in these semantic rules, except for rules **(3)** and (7).
- All other code is generated by the semantic actions associated with assignment-statements and expressions.
- The flow of control causes the proper backpatching so that the assignments and Boolean expression evaluations will connect properly.

Labels and Goto Statements

- Labels are stored in the symbol table (and associated with intermediate language labels).
- Generated as soon as a jump or a declaration is met (to avoid one additional pass)
- For “goto L”, we have to change L into the address of the three-address code for the statement where L is attached.
- When L’s address has been found, we can do this easily with the information in the symbol table.
- Otherwise, we have to use backpatching. We keep the list of address to be filled in the symbol table

Break and Continue Statements

- Break/exit: pass an additional (inherited) attribute to the translation function of loops with the label a break/exit should jump to.
- A new label is passed when entering a new loop
- A break-statement is a jump to the first instruction after the code for the enclosing statement S.
  - keep track of the enclosing statement S
  - generate an unfilled jump for the break- statement
  - put the unfilled jump on S.nextlist
  - backpatch when the exit place is known

Break and Continue Statements

- Continue statements :
  - can be handled in a manner analogous to the break statement.
- The main difference between the two is that the target of the generated jump is different

Translation of a switch-statement

- The "switch" or "case" statement is available in a variety of languages.
- switch-statement syntax is shown below .

```
switch ( E ) {
        case V1: S1
        case V2: S2
                 ...
        case Vn−1: Sn−1
        default: Sn
}
```

- There is a selector expression E, which is to be evaluated
- Followed by n constant values Vl , V2, . . - , Vn that the expression might take
- Has a default "value," which always matches the expression if no other value does.

Translation of a switch-statement

- The intended translation of a switch is code to:
- 1. Evaluate the expression E.
- 2. Find the value V, in the list of cases that is the same as the value of the expression. Recall that the default value matches the expression if none of the values explicitly mentioned in cases does.
- **3.** Execute the statement *Sj* associated with the value found.

Translation of a switch-statement

- Step (2) is an n-way branch, which can be implemented in one of several ways.
- If the number of cases is small (say 10 at most)
  - use a sequence of conditional jumps
  - each jump tests for an individual value and transfers to the code for the corresponding statement.
  - Way to implement this sequence of conditional jumps is to create a table of pairs
    - each pair consisting of a value and a label for the corresponding statement's code.
    - The value of the expression itself, paired with the label for the default statement is placed at the end of the table at run time.
    - A simple loop generated by the compiler compares the value of the expression with each value in the table
    - if no other match is found, the last (default) entry is sure to match

Translation of a switch-statement

- If the number of values exceeds 10
  - Create hash table for the values, with the labels of the various statements as entries.
  - If no entry for the value possessed by the switch expression is found, a jump to the default statement is generated
- Special case that can be implemented even more efficiently than by an n-way branch.
  - If the values all lie in range, say min to max
  - number of different values is a reasonable fraction of max - min
  - then we can construct an array of max-min "buckets," where bucket *j* - min contains the label of the statement with value *j*
  - any bucket that would otherwise remain unfilled contains the default label.

Translation of a switch-statement

- To perform the switch, evaluate the expression to obtain the value *j;*
  - Check that it is in the range min to max
  - transfer indirectly to the table entry at offset *j* - min.
  - For example, if the expression is of type character, a table of, say, 128 entries (depending on the character set) may be created and transferred through with no range testing

Syntax-Directed Translation of Switch-Statements

- A convenient translation of the switch statement is

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

- All tests appear at the end
- A simple code generator can recognize the multiway branch and generate efficient code for it

Syntax-Directed Translation of Switch-Statements

- When the keyword switch is noticed
  - generate two new labels test and next, and a new temporary *t*.
  - parse the expression E and generate code to evaluate E into *t*.
  - After processing E, generate the jump goto test .
- When the keyword case is noticed
  - create a new label L_i and enter it into the symbol table.
  - Each value-label pair consisting of the value V_i of the case constant and L_i (or a pointer to the symbol-table entry for L_i) is placed in a queue
  - We process each statement case $V_i : S_i$ by emitting the label L_i attached to the code for S_i followed by the jump goto next.

Syntax-Directed Translation of Switch-Statements

- When the end of the switch is found
  - generate the code for the n-way branch.
  - Read the queue of value-label pairs
  - generate a sequence of three-address statements of the form shown below.

```
case t V1 L1
case t V2 L2
...
case t Vn-1 Ln-1
case t t Ln
label next
```

  - t is the temporary holding the value of the selector expression E
  - L_n, is the label for the default statement

Syntax-Directed Translation of Switch-Statements

- Another form of translation of switch case

```
        code to evaluate E into t
        if t != V1 goto L1
        code for S1
        goto next
L1:     if t != V2 goto L2
        code for S2
        goto next
L2:
        ...
Ln-2:   if t != Vn-1 goto Ln-1
        code for Sn-1
        goto next
Ln-1:   code for Sn
next:
```

- Compiler should emit code for each of the statements S_i as it notices.
- Hence it is tedious for one-pass compiler to place the branching statements at the beginning
- This form of translation is not effective

Procedure Calls

- Must generate the calling sequence and returning sequence.
- Can be done either at the calling or at the called procedure – depending on machine and OS.

Procedure Calls

S → call id (Elist)

Elist → Elist , E

Elist → E

- Calling sequence
  - allocate space for activation record
  - evaluate arguments
  - establish environment pointers
  - save status and return address
  - jump to the beginning of the procedure

Procedure Calls …

Example

- parameters are passed by reference
- storage is statically allocated
- use param statement as place holder for the arguments
- called procedure is passed a pointer to the first parameter
- pointers to any argument can be obtained by using proper offsets

Code Generation for procedure calls

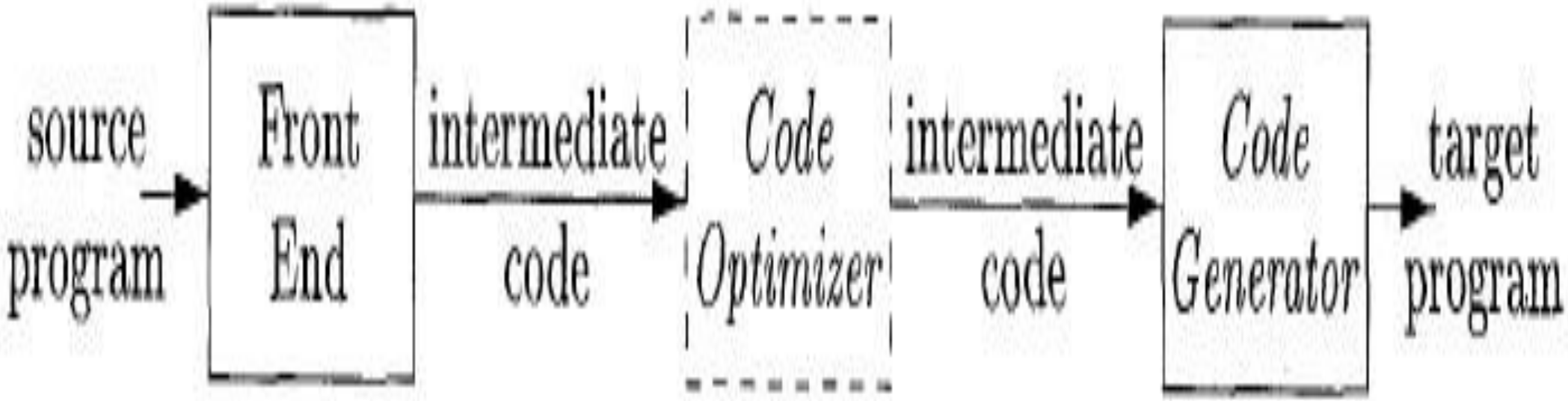
- Generate three address code needed to evaluate arguments which are expressions
- Generate a list of param three address statements
- Store arguments in a list

S → call id (Elist)
    for each item p on queue do emit('param' p)
    emit('call' id.place)

Elist → Elist , E
    append E.place to the end of queue

Elist → E
    initialize queue to contain E.place

Code Generation



Code Generation

Requirements

- Preserve semantic meaning of source program
- Make effective use of available resources of target machine
- Code generator itself must run efficiently

Challenges

- Problem of generating optimal target program is undecidable
- Many subproblems encountered in code generation are computationally

Main Tasks of Code Generator

- **Instruction selection**: choosing appropriate target-machine instructions to implement the IR statements
- **Registers allocation and assignment**: deciding what values to keep in which registers
- **Instruction ordering**: deciding in what order to schedule the execution of instructions

Issues in the Design of a Code Generator

- **Input to the Code Generator**
- **The Target Program**
- **Instruction Selection**
- **Register Allocation**
- **Evaluation Order**

Input to the Code Generator

- The input to the code generator is the intermediate representation of the source program

Three-address representations	**quadruples, triples, indirect triples;**
Virtual machine representations	**Bytecodes and stack-machine code**
Linear representations	**Postfix notation**
Graphical representations	**Syntax trees and DAG's**

- Here it is assume that all syntactic and static semantic errors have been detected, that the necessary type checking has taken place, and that type conversion operators have been inserted wherever necessary.
- The code generator can therefore proceed on the assumption that its **input is free of these kinds of errors**.

The Target Program

- The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code.
- The most common target-machine architectures are
  - RISC (reduced instruction set computer)
    - It has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture
  - CISC (complex instruction set computer),
    - It has few registers, two-address instructions, a variety of addressing modes, several

register classes, variable-length instructions, and instructions with side effects

  - Stack based
    - operations are done by pushing operands onto a stack and then performing the

operations on the operands at the top of the stack

The Target Program (Cont)

- To overcome the high performance penalty of interpretation, *just-in-time* (JIT) Java compilers have been created.
- These JIT compilers translate bytecodes during run time to the native hardware instruction set of the target machine
- Producing an **absolute machine-language** program as output has the advantage that it can be placed in a fixed location in memory and immediately executed
- Producing a **relocatable machine-language** program (often called an *object module)* as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

Instruction Selection

- The code generator must map the IR program into a code sequence that can be executed by the target machine.
- The complexity of performing this mapping is determined by a factors such as
  - The level of the IR
  - The nature of the instruction-set architecture
  - The desired quality of the generated code.

Instruction Selection

- If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement by-statement code generation, however, often produces poor code that needs
- If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences.
- The quality of the generated code is usually determined by its speed and

size.

- On most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations

Register Allocation

- A key problem in code generation is deciding what values to hold in what registers.
- Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values.
- Values not held in registers need to reside in memory.
- Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.
- The use of registers is often subdivided into two sub-problems:
  - **Register allocation**, during which we select the set of variables that will reside in registers at each

point in the program.

  - **Register assignment**, during which we pick the specific register that a variable will reside in.
- Finding an optimal assignment of registers to variables is difficult, even with single-register machines.
- Mathematically, the problem is **NP-complete**.
- The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-usage conventions be observed.

Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation orders require fewer registers to hold intermediate results than others.
- However, picking a best order in the general case is a **difficult NP-complete problem**.
- Initially, we shall **avoid** the problem **by generating code for the three- address statements** in the order in which they have been produced by the intermediate code generator.

Simple Code Generator

Content..

- ✔Introduction(Simple Code Generator)
- ✔Register and Address Descriptors
- ✔A Code-Generation Algorithm
- ✔The Function getreg
- ✔Generating Code for Other Types of Statements

Introduction

- **Code Generator:**

✔ Code generator can be considered as the final phase of compilation.

✔ It generates a target code for a sequence of three-address statement.

✔ It consider each statement in turn, remembering if any of the operands of the statement are currently in registers, and taking advantage of that fact if possible.

Introduction

- We assume that computed results can be left in registers an long as possible.
- Storing them only

a) if their register is needed for another computation (or)

b) Just before a procedure call, jump, or labeled statement

- Condition (b)implies that everything must be stored just before the end of a basic block.

Introduction

- The reason we must do so is that, after leaving a basic block, we may be able to go to several different blocks, or we may go to one particular block that can be reached from several others.
- In either case, we cannot, without extra effort, assume that a datum used by a block appears in the same register no matter how control reached that block.
- Thus, to avoid a possible error, our simple code- generator algorithm stores everything when moving across basic block boundaries as well as when procedure calls are made.

Introduction

- We can produce reasonable code for a three- address statement **a:=b+c**
- If we generate the single instruction **ADD Rj,Ri**
- with cost one
- If Ri contain b but c is in a memory location, we can generate the sequence

Add c,Ri **cost=2**

(or)

Mov c,Rj
Add Rj, Ri **cost=3**

Register and Address Descriptors

- The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:
  - **Register descriptor**
  - **Address descriptor**

Register descriptor

- Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.

  - Keep track of what is currently in each register.
  - Initially all the registers are empty

Address descriptor

- Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

  - Keep track of location where current value of the name can be found at runtime
  - The location might be a register, stack, memory address or a set of those

A Code-Generation Algorithm

- Basic blocks comprise of a sequence of three- address instructions. Code generator takes these sequence of instructions as input.
- For each three-address statement of the form

x:=y op z

we perform the following actions

I. Call function getReg, to decide the location of L.

II. Determine the present location (register or memory) of **y** by consulting the Address Descriptor of **y**. If **y** is not presently in register **L**, then generate the following instruction to copy the value of **y** to **L**

MOV y', L

where **y'** represents the copied value of **y**.

III. Determine the present location of **z** using the same method used in step 2 for **y** and generate the following instruction:

OP z', L

where **z'** represents the copied value of **z**.

IV. Now L contains the value of y OP z, that is intended to be assigned to **x**. So, if L is a register, update its descriptor to indicate that it contains the value of **x**. Update the descriptor of **x** to indicate that it is stored at location **L**.

If y and z has no further use, they can be given back to the system.

The Function getreg

- Code generator uses *getReg* function to determine the status of available registers and the location of name values. *getReg* works as follows:
- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

The Function getreg

1. If Y is in register (that holds no other values) and Y is not live and has no next use after

X = Y op Z

then return register of Y for L.

2. Failing (1) return an empty register

3. Failing (2) if X has a next use in the block or op requires register then get a register R, store its content into M (by Mov R, M) and use it.

4. else select memory location X as L

Example

- For example, the assignment d := (a - b) + (a -

c) + (a - c) might be translated into the following three-address code sequence:

- $t_1 = a - b$
- $t_2 = a - c$
- $t_3 = t_1 + t_2$
- $d = t_3 + t_2$

Example

Stmt	code	reg desc	addr desc
t_1=a-b	mov a,R_0	R_0 contains t_1	t_1 in R_0
	sub b,R_0		
t_2=a-c	mov a,R_1	R_0 contains t_1	t_1 in R_0
	sub c,R_1	R_1 contains t_2	t_2 in R_1
t_3=t_1+t_2	add R_1,R_0	R_0 contains t_3	t_3 in R_0
		R_1 contains t_2	t_2 in R_1
d=t_3+t_2	add R_1,R_0	R_0 contains d	d in R_0
	mov R_0,d		d in R_0 and memory

Generating Code for Other Types of Statements

- Indexed assignment

a:=b[i] and

a[i]:=b

- Pointer assignment

a:=*p and

*p:=a

⑥ Assignment statements

- can be a type of integer, real, array & record.

Syntax-directed translation scheme for generation of 3-address code for assignment stmts

$S \rightarrow id = E$

$E \rightarrow E_1 + E_2$

~~$E \rightarrow E_1 * E_2$~~

$E \rightarrow -E_1$

$E \rightarrow (E_1)$

$E \rightarrow id$

- Following syntax-directed defn builds up 3-addr code for an assignment stmt S using attribute 'code' for S and attribute 'code' and 'addr' for expression E.

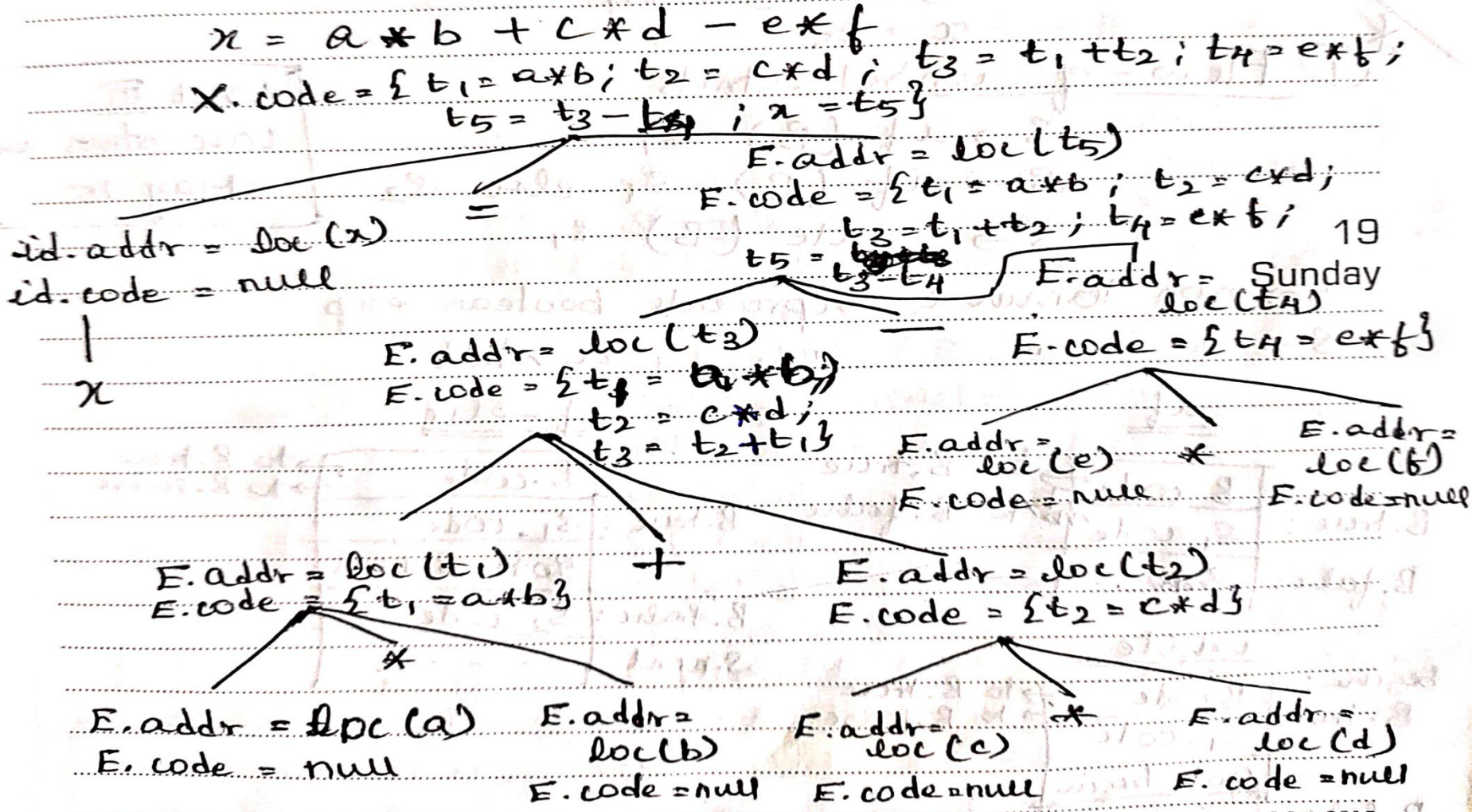
Friday

- Attributes S.code and E.code denote 3-addr code for S and E.
- Att E.addr denotes addr that will hold the value of E.
addr can be a name, a constant or a compiler-generated temporary.

Production	Semantic rules
$S \to id = E;$	S.code = E.code || gen(top.get('id.lexeme') '=' E.addr)
$E \to E_1 + E_2$	E.addr = new Temp() E.code = E_1.code || E_2.code || gen(E.addr '=' E_1.addr '+' E_2.addr)
$\mid -E_1$	E.addr = new Temp() E.code = E_1.code || gen(E.addr '=' 'minus' E_1.addr)
$\mid (E_1)$	E.addr = E_1.addr E.code = E_1.code
$\mid id$	E.addr = top.get('id.lexeme') E.code = ''

E → id

- Consider E → id. When an exp is a single identifier, say x, then x itself holds the value of the expression.
- Semantic rules for this prodn define E.addr to point to symbol-table entry for this instance of id. Let 'top' denote current symbol table.



∴ 3-addr code for above stmt is

$t_1 = a * b$

$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = e * f$

$t_5 = t_3 - t_4$

$x = t_5$

Cross Compilation

- A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is run.

- Cross compiler tools are used to generate executables for embedded system or multiple platforms.

- It is used to compile for a platform upon which it is not feasible to do the compiling, like microcontrollers that don't support an operating system.

Uses of cross compilers

- The fundamental use of a cross compiler is to separate the build environment from target environment. This is useful in a number of situations:
  - Embedded computers where a device has extremely limited resources.
  - For example, a microwave oven will have an extremely small computer to read its touchpad and door sensor, provide output to a digital display and speaker, and to control the machinery for cooking food.
- This computer will not be powerful enough to run a compiler, a file system, or a development environment.
- Since debugging and testing may also require more resources than are available on an embedded system, cross-compilation can be less involved and less prone to errors than native compilation.

•Compiling for multiple machines

- For example, a company may wish to support several different versions of an operating system or to support several different operating systems.
- By using a cross compiler, a single build environment can be set up to compile for each of these targets.
- Compiling on a server farm.
- Similar to compiling for multiple machines, a complicated build that involves many compile operations can be executed across any machine that is free, regardless of its underlying hardware or the operating system version that it is running.

•GCC and cross compilation

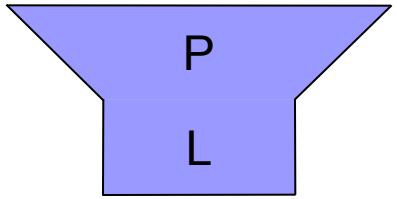
- GCC, a free software collection of compilers, can be set up to cross compile. It supports many platforms and languages.

T-diagrams

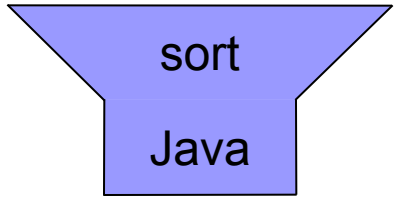
- Different diagrams for different kinds of programs
- Visual explanation of interactions involving compilers and interpreters

Programs

- Program P written in language L

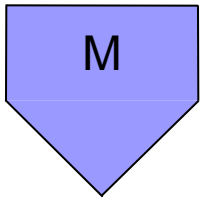


- Example: Sort program written in Java

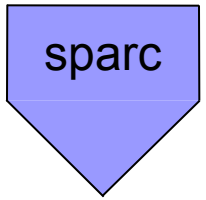


Machines

- Machine executing language M

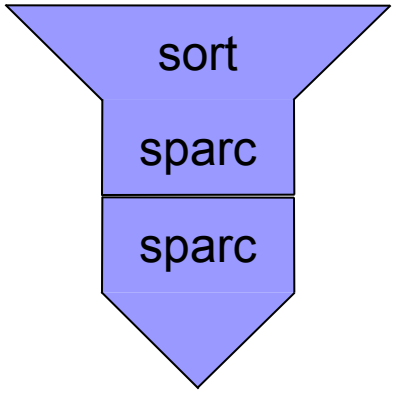


- Example: Sun workstation executing SPARC machine code



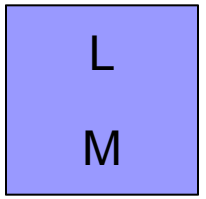
Executing Programs

- Program implementation language must match machine

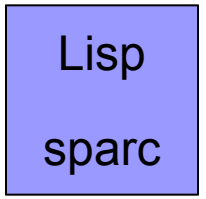


Interpreters

- Interpreter executing language L written in language M

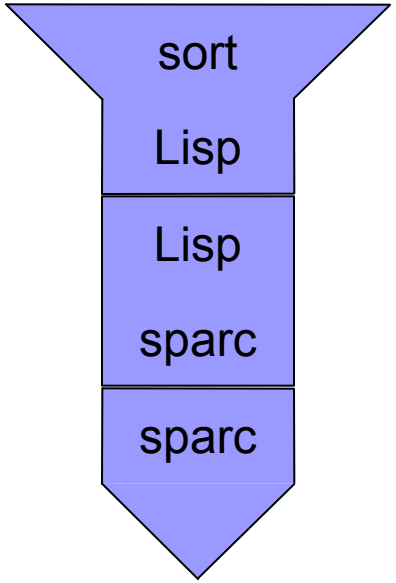


- Example: Lisp interpreter running on sparc



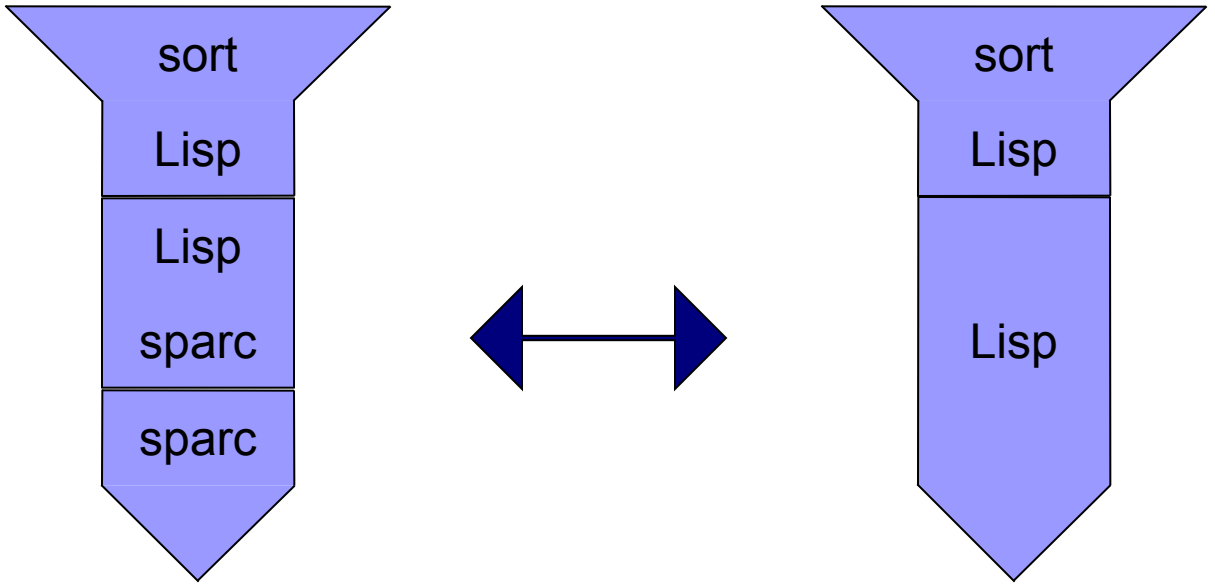
Interpreting Programs

- Interpreter mediates between programming language and machine language



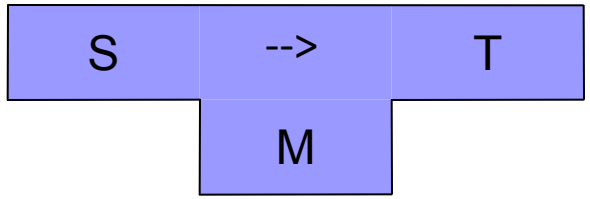
Virtual Machines

- Interpreter creates a “virtual machine”

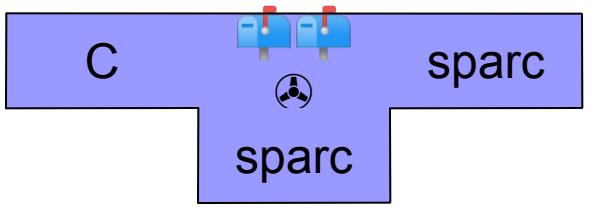


Compilers

- Compiler translating from source language S to target language T implemented in M

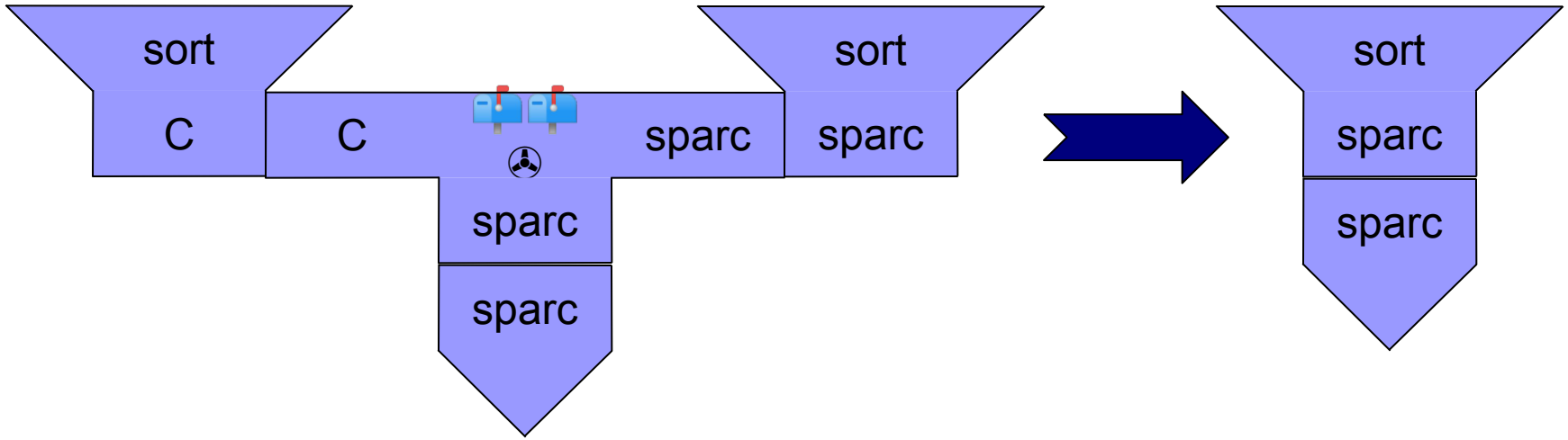


- Example: C compiler for sparc platform



Compiling Programs

- Compiler inputs program in source language, outputs in target language

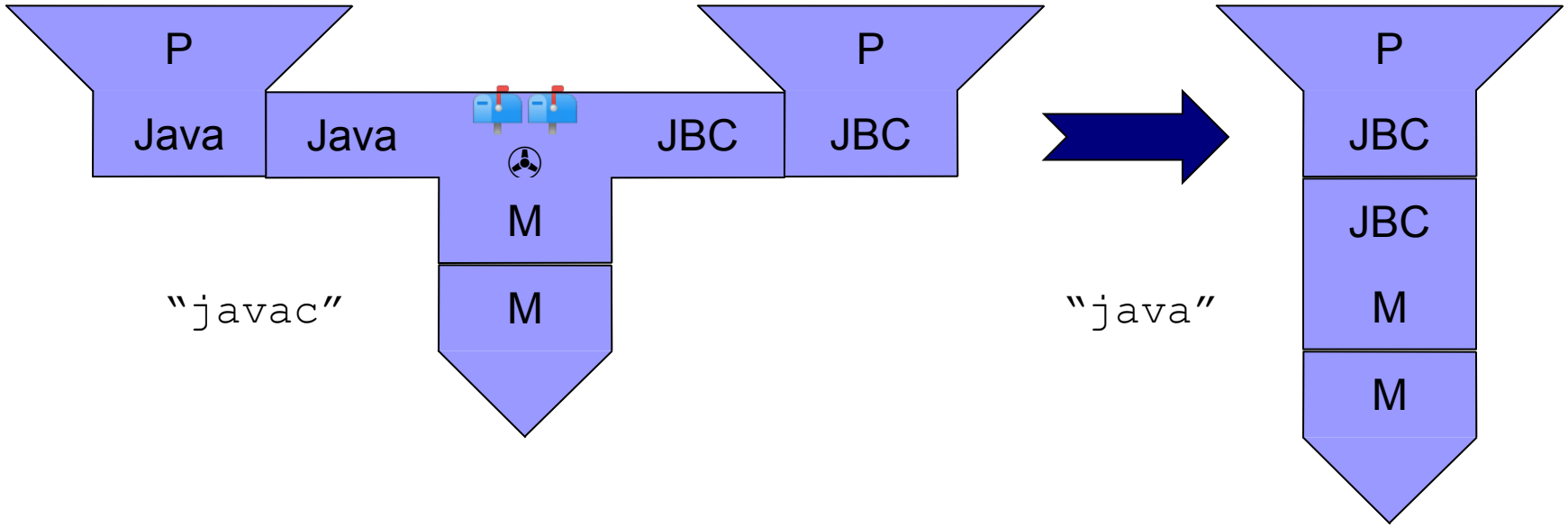


Compiler Correctness

- Compiled target program has the same meaning as the source program

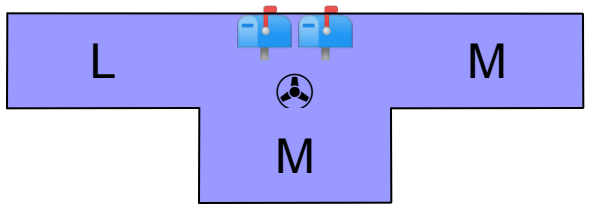
Java Programming Environment

- `Javac`: Java to Java byte code (JBC) compiler
- `Java`: Java Virtual Machine byte code interpreter



Where Do Compilers Come From?

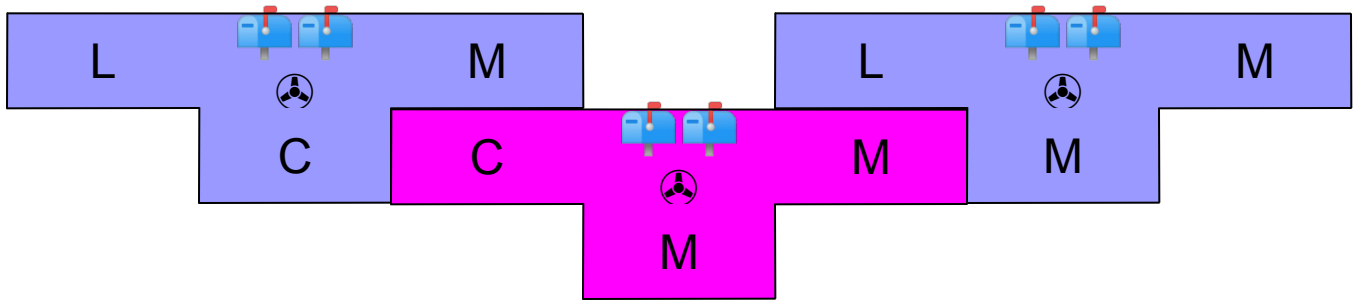
1. Write it in machine code



A lot of work

Where Do Compilers Come From?

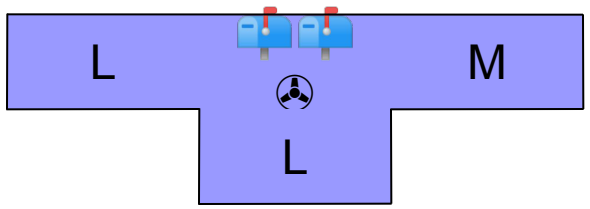
1. Write it in machine code
2. Write it in a lower level language and compile it using an existing compiler



But where did the C compiler come from?

Where Do Compilers Come From?

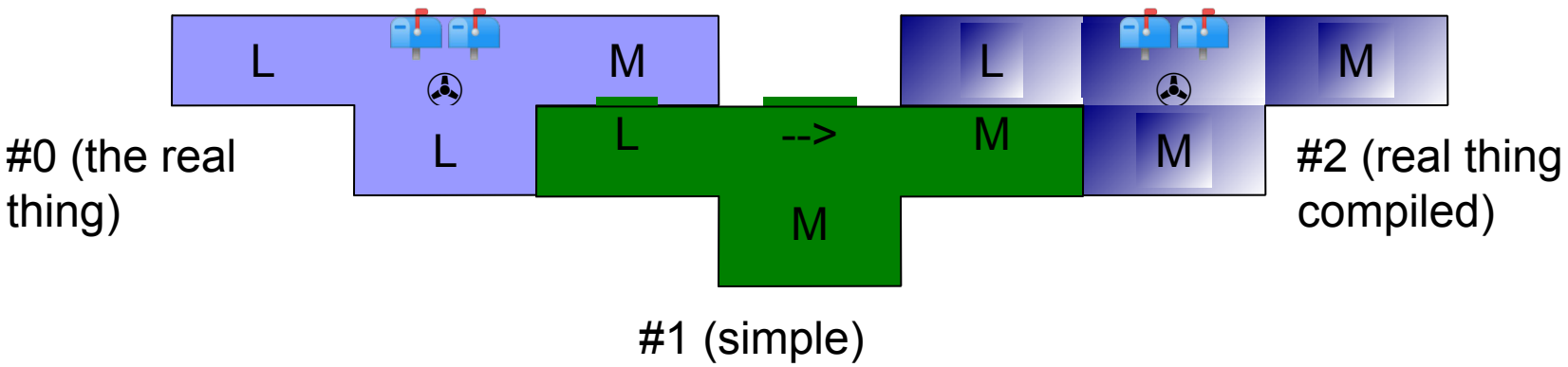
1. Write it in machine code
2. Write it in a lower level language and compile it using an existing compiler
3. Write it in the same language that it compiles and ***bootstrap***



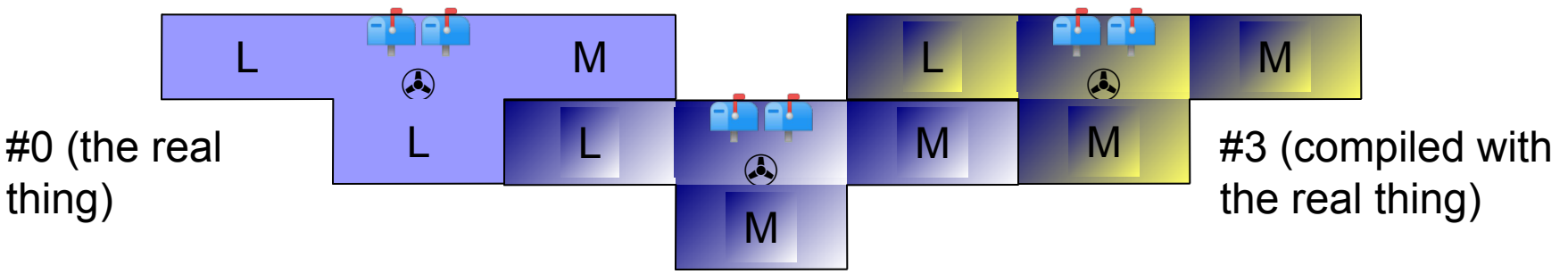
Bootstrapping a Compiler

- Write the compiler in its own language (#0)
- Write a simple native compiler (#1)
- Use compiler #1 to compile #0 to get native compiler with more frills (#2)
- Repeat as desired

Bootstrapping a Compiler



Bootstrapping a Compiler, Stage 2



#0 (the real thing)

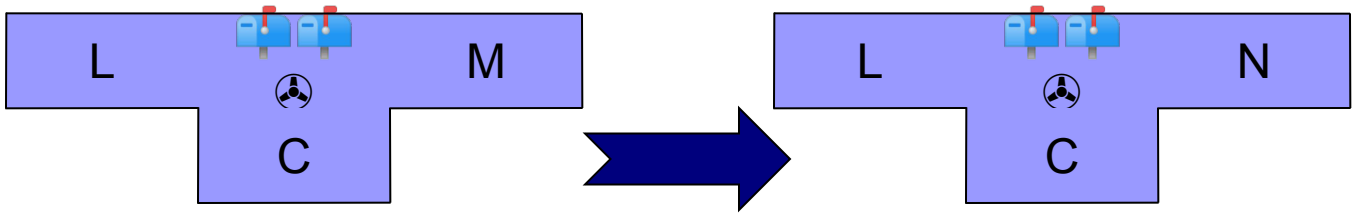
#2 (the real thing compiled)

#3 (compiled with the real thing)

Consistency test: #2 = #3 literally

Porting a Compiler

1. Rewrite back end to target new machine

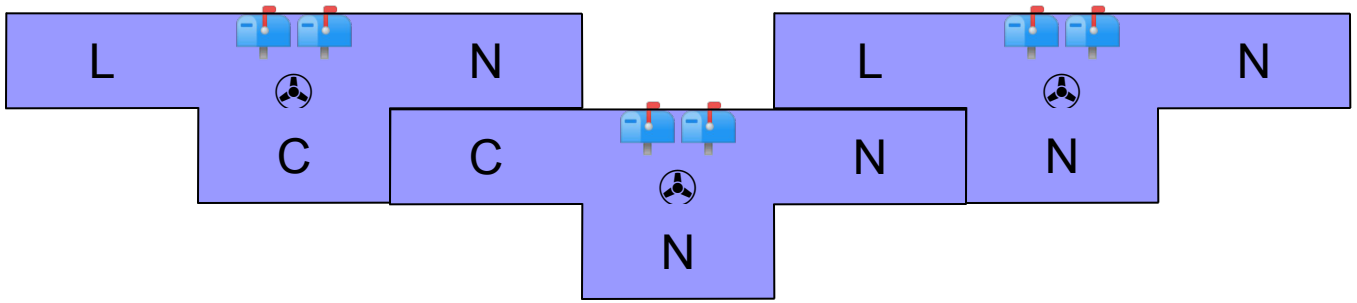


2. Compile on new machine

Porting a Compiler

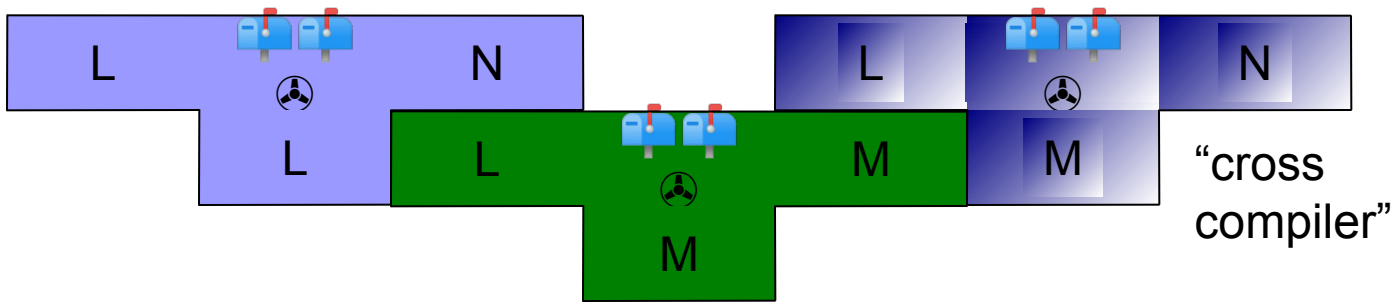
- 1. Rewrite back end to target new machine

- 2. Compile on new machine



Porting a Compiler II

- Rewrite back end to target new machine
- Compile using native compiler



Cross Compilers

- A cross compiler compiles to a target language different from the language of the machine it runs on

Porting a Compiler II

- Rewrite back end to target new machine
- Compile using native compiler
- Recompile using cross compiler