BCSE307L – COMPILER DESIGN

TEXT BOOK:

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers: Principles, Techniques and Tools", Second Edition, Pearson Education Limited, 2014.

Module:4	INTERMEDIATE CODE GENERATION	5 hours
Variants of Syntax trees - Three Address Code- Types – Declarations - Procedures - Assignment Statements - Translation of Expressions - Control Flow - Back Patching- Switch Case Statements.		

Intermediate Code Generation

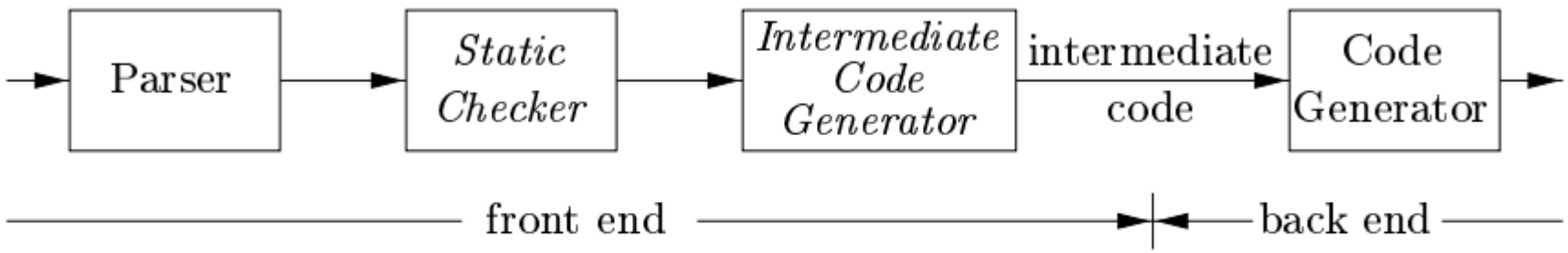


Figure 6.1: Logical structure of a compiler front end

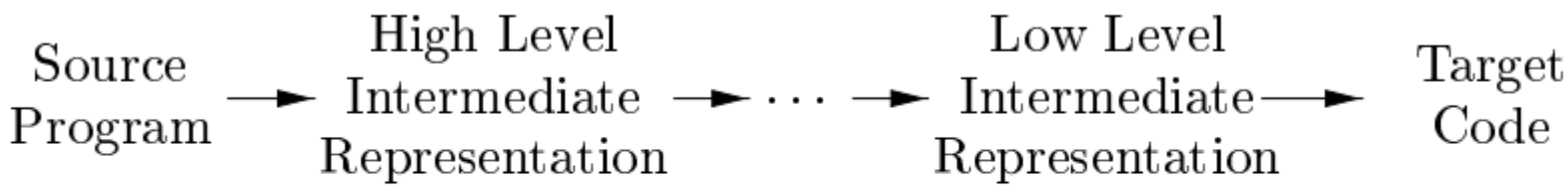


Figure 6.2: A compiler might use a sequence of intermediate representations

A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection. Three-address code can range from high- to low-level, depending on the choice of operators.

- Variants of Syntax Trees
  - Directed Acyclic Graphs for Expressions
  - The Value-Number Method for Constructing

Variants of Syntax Trees

Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct. A directed acyclic graph (hereafter called a *DAG*) for an expression identifies the *common subexpressions* (subexpressions that occur more than once) of the expression. As we shall see in this section, DAG's can be constructed by using the same techniques that construct syntax trees.

Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more succinctly, it

Example 6.1: Figure 6.3 shows the DAG for the expression

```
a + a * (b - c) + (b - c) * d
```

Example 6.1 : Figure 6.3 shows the DAG for the expression

```
a + a * (b - c) + (b - c) * d
```

Example 6.1 : Figure 6.3 shows the DAG for the expression

```
a + a * (b - c) + (b - c) * d
```

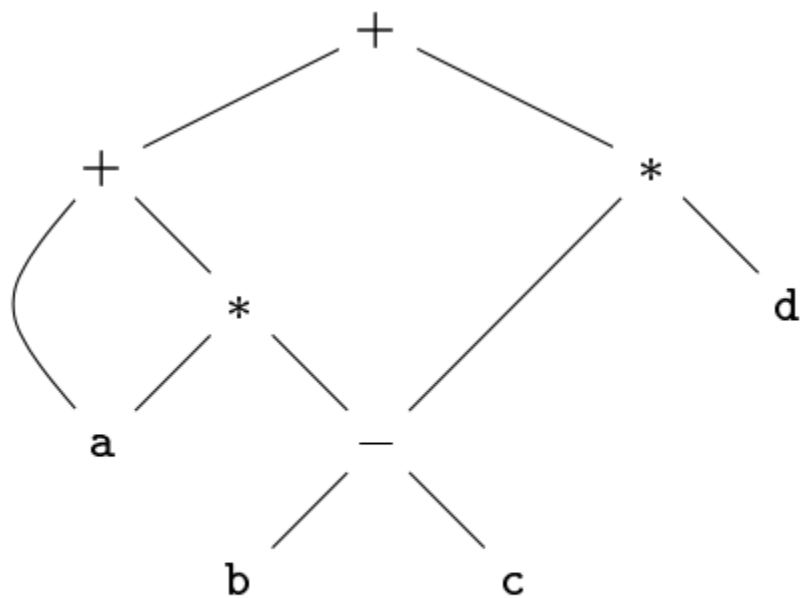


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

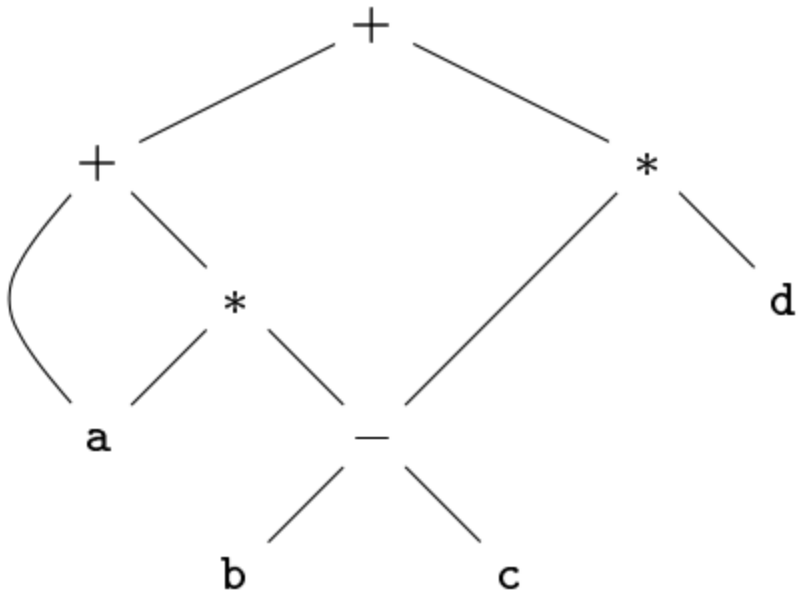
The SDD of Fig. 6.4 can construct either syntax trees or DAG's. It was used to construct syntax trees in Example 5.11, where functions *Leaf* and *Node* created a fresh node each time they were called. It will construct a DAG if, before creating a new node, these functions first check whether an identical node already exists. If a previously created identical node exists, the existing node

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (\ E\ )$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

$$
\begin{aligned}
1)\quad & p_1 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}a) \\
2)\quad & p_2 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}a) = p_1 \\
3)\quad & p_3 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}b) \\
4)\quad & p_4 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}c) \\
5)\quad & p_5 = \mathit{Node}('-', p_3, p_4) \\
6)\quad & p_6 = \mathit{Node}('*', p_1, p_5) \\
7)\quad & p_7 = \mathit{Node}('+', p_1, p_6) \\
8)\quad & p_8 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}b) = p_3 \\
9)\quad & p_9 = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}c) = p_4 \\
10)\quad & p_{10} = \mathit{Node}('-', p_3, p_4) = p_5 \\
11)\quad & p_{11} = \mathit{Leaf}\,(\mathbf{id}, \mathit{entry}\text{-}d) \\
12)\quad & p_{12} = \mathit{Node}('*', p_5, p_{11}) \\
13)\quad & p_{13} = \mathit{Node}('+', p_7, p_{12})
\end{aligned}
$$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3



1) $p_1 = Leaf(\mathbf{id}, entry\text{-}a)$
2) $p_2 = Leaf(\mathbf{id}, entry\text{-}a) = p_1$
3) $p_3 = Leaf(\mathbf{id}, entry\text{-}b)$
4) $p_4 = Leaf(\mathbf{id}, entry\text{-}c)$
5) $p_5 = Node('-', p_3, p_4)$
6) $p_6 = Node('*', p_1, p_5)$
7) $p_7 = Node('+', p_1, p_6)$
8) $p_8 = Leaf(\mathbf{id}, entry\text{-}b) = p_3$
9) $p_9 = Leaf(\mathbf{id}, entry\text{-}c) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf(\mathbf{id}, entry\text{-}d)$
12) $p_{12} = Node('*', p_5, p_{11})$
13) $p_{13} = Node('+', p_7, p_{12})$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

The Value-Number Method for Constructing

Often, the nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 6.6. Each row of the array represents one record, and therefore one node. In each record, the first field is an operation code, indicating the label of the node. In Fig. 6.6(b), leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case), and interior nodes have two additional fields indicating the left and right children.

The Value-Number Method for Constructing

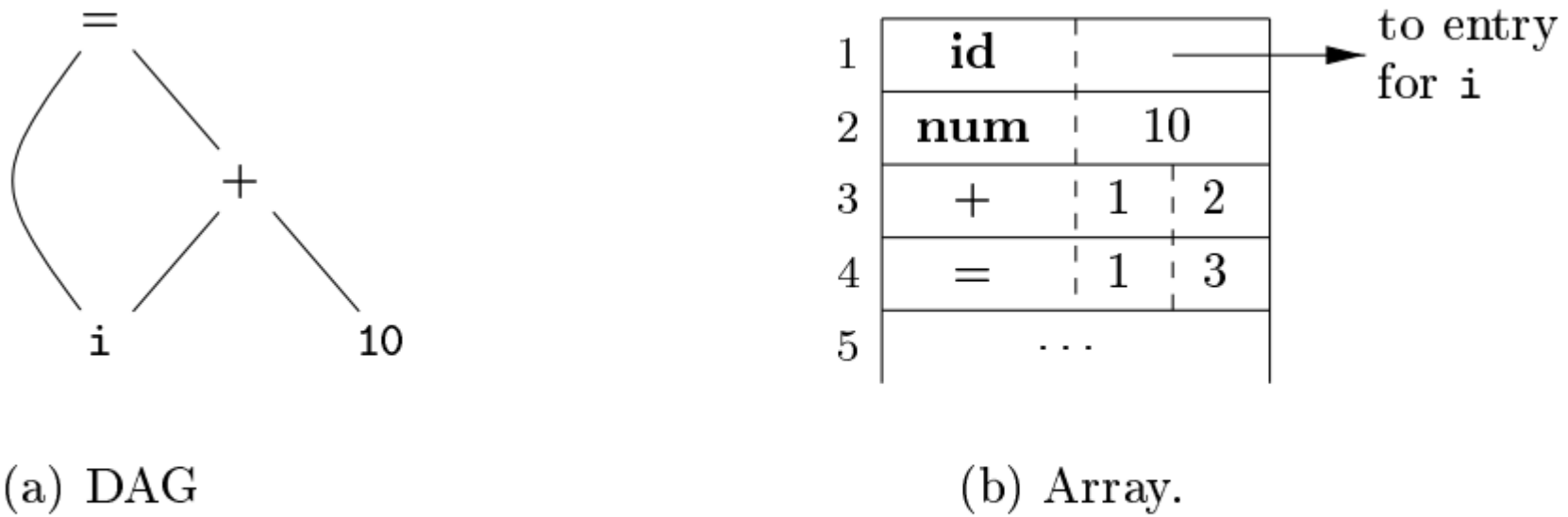
Algorithm 6.3: The value-number method for constructing the nodes of a DAG.

INPUT: Label op, node l, and node r.

OUTPUT: The value number of a node in the array with signature $\langle op, l, r \rangle$.

METHOD: Search the array for a node M with label op, left child l, and right child r. If there is such a node, return the value number of M. If not, create in the array a new node N with label op, left child l, and right child r, and return its value number.

The Value-Number Method for Constructing



(a) DAG (b) Array.

Figure 6.6: Nodes of a DAG for $i = i + 10$ allocated in an array

Three Address Code

- Address and Instructions
- Quadruples
- Triples
- Indirect Triples
- Static Single-Assignment Form

Three Address Code (TAC)

source-language expression like `x+y*z` might be translated into the sequence of three-address instructions

$$t_1 = y * z$$
$$t_2 = x + t_1$$

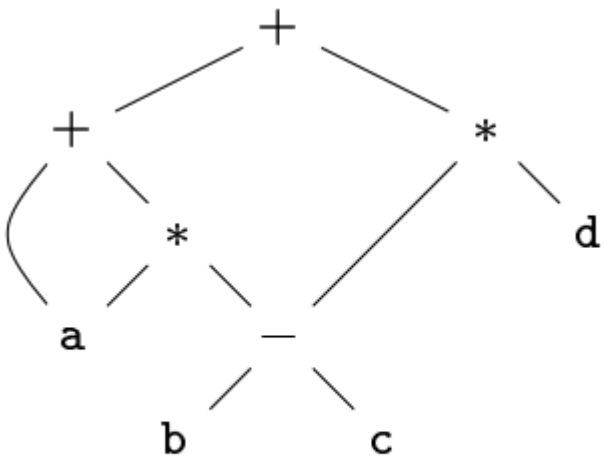
where t_1 and t_2 are compiler-generated temporary names.

TAC

Example 6.4 : Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. The DAG in Fig. 6.3 is repeated in Fig. 6.8, together with a corresponding three-address code sequence. □

TAC

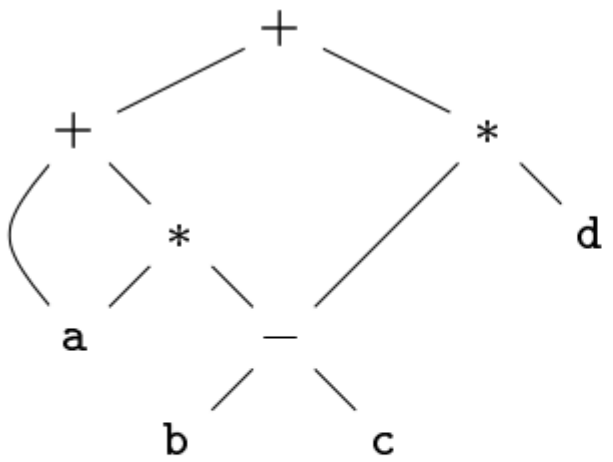
```
a + a * (b - c) + (b - c) * d
```



(a) DAG

(b) Three-address code

TAC



(a) DAG

t_1 = b - c
t_2 = a * t_1
t_3 = a + t_2
t_4 = t_1 * d
t_5 = t_3 + t_4

(b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

Address and Instructions

Address

An address can be one of the following:

- *A name.* For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.

- *A constant.* In practice, a compiler must deal with many different types of constants and variables. Type conversions within expressions are considered in Section 6.5.2.

- *A compiler-generated temporary.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

Three Address Instructions

1. Assignment instructions of the form x = y *op* z, where *op* is a binary arithmetic or logical operation, and x, y, and z are addresses.

2. Assignments of the form x = *op* y, where *op* is a unary operation. Essential unary operations include unary minus, logical negation, and conversion operators that, for example, convert an integer to a floating-point number.

3. *Copy instructions* of the form x = y, where x is assigned the value of y.

4. An unconditional jump `goto` L. The three-address instruction with label L is the next to be executed.

5. Conditional jumps of the form `if` x `goto` L and `ifFalse` x `goto` L. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

Three Address Instructions

6. Conditional jumps such as `if` x *relop* y `goto` L, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y. If not, the three-address instruction following `if` x *relop* y `goto` L is executed next, in sequence.

7. Procedure calls and returns are implemented using the following instructions: `param` x for parameters; `call` p, n and y = `call` p, n for procedure and function calls, respectively; and `return` y, where y, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

```
param x1
param x2
...
param xn
call p, n
```

generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$.

Three Address Instructions

8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$. The instruction $x = y[i]$ sets x to the value in the location i memory units beyond location y. The instruction $x[i] = y$ sets the contents of the location i units beyond x to the value of y.

9. Address and pointer assignments of the form $x = \&y$, $x = *y$, and $*x = y$. The instruction $x = \&y$ sets the r-value of x to be the location (l-value) of y.[2] Presumably y is a name, perhaps a temporary, that denotes an expression with an l-value such as `A[i][j]`, and x is a pointer name or temporary. In the instruction $x = *y$, presumably y is a pointer or a temporary whose r-value is a location. The r-value of x is made equal to the contents of that location. Finally, $*x = y$ sets the r-value of the object pointed to by x to the r-value of y.

Three Address Instructions

Example 6.5 : Consider the statement

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

Three Address Instructions

Example 6.5 : Consider the statement

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

Two possible translations of this statement are shown in Fig. 6.9. The translation in Fig. 6.9(a) uses a symbolic label `L`, attached to the first instruction. The translation in (b) shows position numbers for the instructions, starting arbitrarily at position 100. In both translations, the last instruction is a conditional jump to the first instruction. The multiplication `i*8` is appropriate for an array of elements that each take 8 units of space.

Three Address Instructions

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

(a) Symbolic labels.

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

(b) Position numbers.

Figure 6.9: Two ways of assigning labels to three-address statements

Quadruples

A *quadruple* (or just "*quad*") has four fields, which we call *op*, arg_1, arg_2, and *result*. The *op* field contains an internal code for the operator. For instance, the three-address instruction $x = y + z$ is represented by placing + in *op*, y in arg_1, z in arg_2, and x in *result*. The following are some exceptions to this rule:

Quadruples

1. Instructions with unary operators like x = `minus` y or x = y do not use arg_2. Note that for a copy statement like x = y, op is =, while for most other operations, the assignment operator is implied.

2. Operators like `param` use neither arg_2 nor *result*.

3. Conditional and unconditional jumps put the target label in *result*.

Quadruples

Example 6.6 : Three-address code for the assignment `a = b * - c + b * - c ;` appears in Fig. 6.10(a). The special operator `minus` is used to distinguish the unary minus operator, as in `- c`, from the binary minus operator, as in `b - c`. Note that the unary-minus "three-address" statement has only two addresses, as does the copy statement `a = t5`.

The quadruples in Fig. 6.10(b) implement the three-address code in (a).

Quadruples

For readability, we use actual identifiers like `a`, `b`, and `c` in the fields arg_1, arg_2, and *result* in Fig. 6.10(b), instead of pointers to their symbol-table entries. Temporary names can either by entered into the symbol table like programmer-defined names, or they can be implemented as objects of a class *Temp* with its own methods.

Quadruples

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

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

(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

Triples

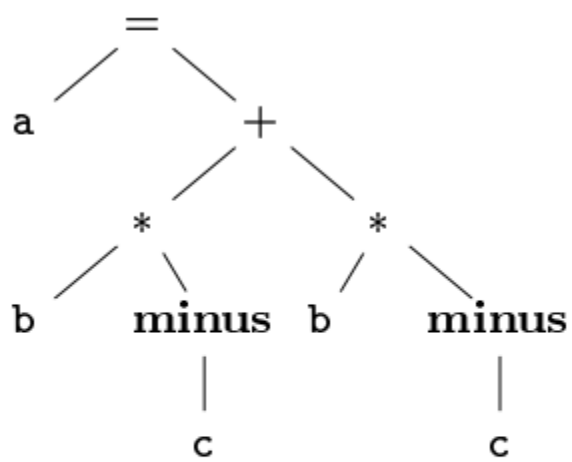
A *triple* has only three fields, which we call *op*, arg_1, and arg_2. Note that the *result* field in Fig. 6.10(b) is used primarily for temporary names. Using triples, we refer to the result of an operation $x\ op\ y$ by its position, rather than by an explicit temporary name. Thus, instead of the temporary $\mathtt{t}_1$ in Fig. 6.10(b), a triple representation would refer to position (0). Parenthesized numbers represent pointers into the triple structure itself. In Section 6.1.2, positions or pointers to positions were called value numbers.

Triples

Example 6.7: The syntax tree and triples in Fig. 6.11 correspond to the three-address code and quadruples in Fig. 6.10. In the triple representation in Fig. 6.11(b), the copy statement `a = t`$_5$ is encoded in the triple representation by placing `a` in the arg_1 field and (4) in the arg_2 field.

Triples

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



(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

Figure 6.11: Representations of `a = b * - c + b * - c ;`

A ternary operation like $x[i] = y$ requires two entries in the triple structure; for example, we can put x and i in one triple and y in the next. Similarly, $x = y[i]$ can implemented by treating it as if it were the two instructions $t = y[i]$ and $x = t$, where t is a compiler-generated temporary.

- Triple representation
  - x[i] = y

	op	arg1	arg2
(0)	[ ] =	x	i
(1)	assign	(0)	y

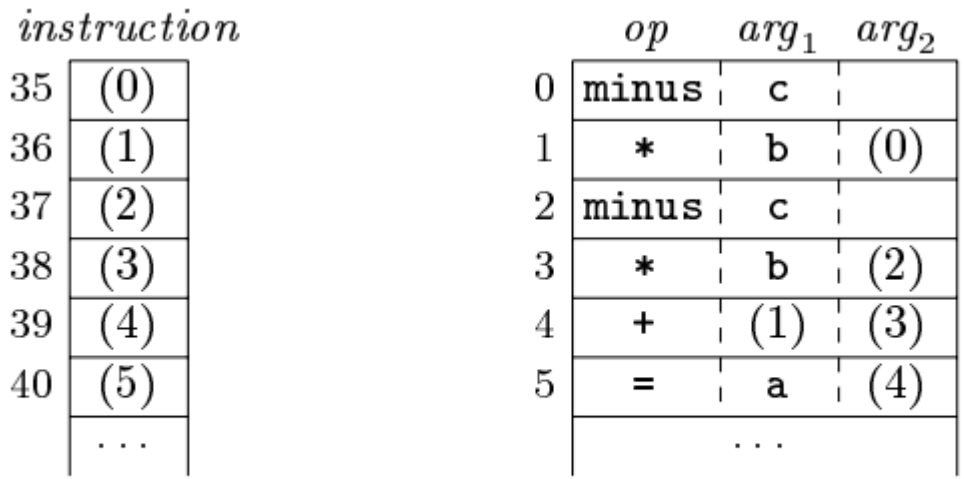
- Triple representation
  - x = y[i]

	op	arg1	arg2
(0)	= [ ]	y	i
(1)	assign	x	(0)

Indirect Triples

Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves. For example, let us use an array *instruction* to list pointers to triples in the desired order. Then, the triples in Fig. 6.11(b) might be represented as in Fig. 6.12.

Indirect Triples



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

	op	arg_1	arg_2
0	`minus`	`c`	
1	`*`	`b`	(0)
2	`minus`	`c`	
3	`*`	`b`	(2)
4	`+`	(1)	(3)
5	`=`	`a`	(4)
	...		

Figure 6.12: Indirect triples representation of three-address code

Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code. The first is that all assignments in SSA are to variables with distinct names; hence the term *static single-assigment*. Figure 6.13 shows the same intermediate program in three-address code and in static single-assignment form. Note that subscripts distinguish each definition of variables p and q in the SSA representation.

Static Single-Assignment Form

```
p = a + b
q = p - c
p = q * d
p = e - p
q = p + q
```

(a) Three-address code.

```
p1 = a + b
q1 = p1 - c
p2 = q1 * d
p3 = e - p2
q2 = p3 + q1
```

(b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

SSA

The same variable may be defined in two different control-flow paths in a program. For example, the source program

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

Types and Declarations

- Type Expression
- Type Equivalence
- Declarations
- Storage layout for Local Names
- Sequences of Declarations
- Fields in Records and Classes

Types and Declarations

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

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

Type Expression

Types have structure, which we shall represent using *type expressions*: a type expression is either a basic type or is formed by applying an operator called a *type constructor* to a type expression. The sets of basic types and constructors depend on the language to be checked.

Type Expression

Example 6.8: The array type `int[2][3]` can be read as "array of 2 arrays of 3 integers each" and written as a type expression *array*(2, *array*(3, *integer*)). This type is represented by the tree in Fig. 6.14. The operator *array* takes two parameters, a number and a type.

Type Expression

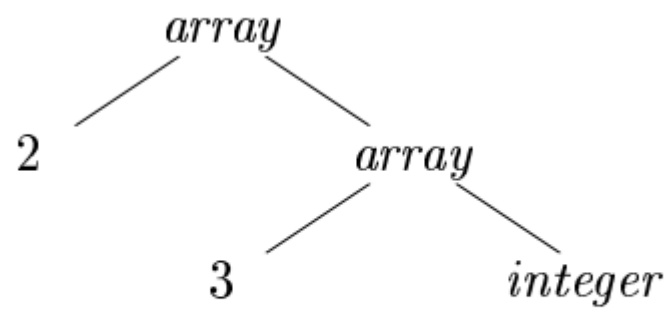


Figure 6.14: Type expression for `int[2][3]`

Type Expression

We shall use the following definition of type expressions:

- A basic type is a type expression. Typical basic types for a language include *boolean*, *char*, *integer*, *float*, and *void*; the latter denotes "the absence of a value."
- 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 fields. A type expression can be formed by applying the *record* type constructor to the field names and their types. Record types will be implemented in Section 6.3.6 by applying the constructor *record* to a symbol table containing entries for the fields.
- A type expression can be formed by using the type constructor $\rightarrow$ for function types. We write $s \rightarrow t$ for "function from type s to type t." Function types will be useful when type checking is discussed in Section 6.5.

Type Expression

- If s and t are type expressions, then their Cartesian product $s \times t$ is a type expression. Products are introduced for completeness; they can be used to represent a list or tuple of types (e.g., for function parameters). We assume that $\times$ associates to the left and that it has higher precedence than $\rightarrow$.
- Type expressions may contain variables whose values are type expressions. Compiler-generated type variables will be used in Section 6.5.4.

Type Equivalence

When are two type expressions equivalent? Many type-checking rules have the form, "**if** two type expressions are equal **then** return a certain type **else** error."

When type expressions are represented by graphs, two types are *structurally equivalent* if and only if one of the following conditions is true:

- They are the same basic type.
- They are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

Declarations

The types and declarations using a simplified grammar that declares just one name at a time; declarations with lists of names can be handled as discussed in Example 5.10. The grammar is

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

Declarations

Nonterminal D generates a sequence of declarations. Nonterminal T generates basic, array, or record types. Nonterminal B generates one of the basic types **int** and **float**. Nonterminal C, for "component," generates strings of zero or more integers, each integer surrounded by brackets. An array type consists of a basic type specified by B, followed by array components specified by nonterminal C. A record type (the second production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

Storage layout for Local Names

From the type of a name, we can determine the amount of storage that will be needed for the name at run time. At compile time, we can use these amounts to assign each name a relative address. The type and relative address are saved in the symbol-table entry for the name. Data of varying length, such as strings, or data whose size cannot be determined until run time, such as dynamic arrays, is handled by reserving a known fixed amount of storage for a pointer to the data.

Storage layout for Local Names

The *width* of a type is the number of storage units needed for objects of that type. A basic type, such as a character, integer, or float, requires an integral number of bytes. For easy access, storage for aggregates such as arrays and classes is allocated in one contiguous block of bytes.[4]

Storage layout for Local Names

The translation scheme (SDT) in Fig. 6.15 computes types and their widths for basic and array types; record types will be discussed in Section 6.3.6. The SDT uses synthesized attributes *type* and *width* for each nonterminal and two variables t and w to pass type and width information from a B node in a parse tree to the node for the production $C \rightarrow \epsilon$. In a syntax-directed definition, t and w would be inherited attributes for C.

Storage layout for Local Names

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

Figure 6.15: Computing types and their widths

Storage layout for Local Names

Example 6.9 : The parse tree for the type `int [2] [3]` is shown by dotted lines in Fig. 6.16. The solid lines show how the type and width are passed from B, down the chain of C's through variables t and w, and then back up the chain as synthesized attributes *type* and *width*. The variables t and w are assigned the values of $B.type$ and $B.width$, respectively, before the subtree with the C nodes is examined. The values of t and w are used at the node for $C \rightarrow \epsilon$ to start the evaluation of the synthesized attributes up the chain of C nodes.

Storage layout for Local Names

Storage layout for Local Names

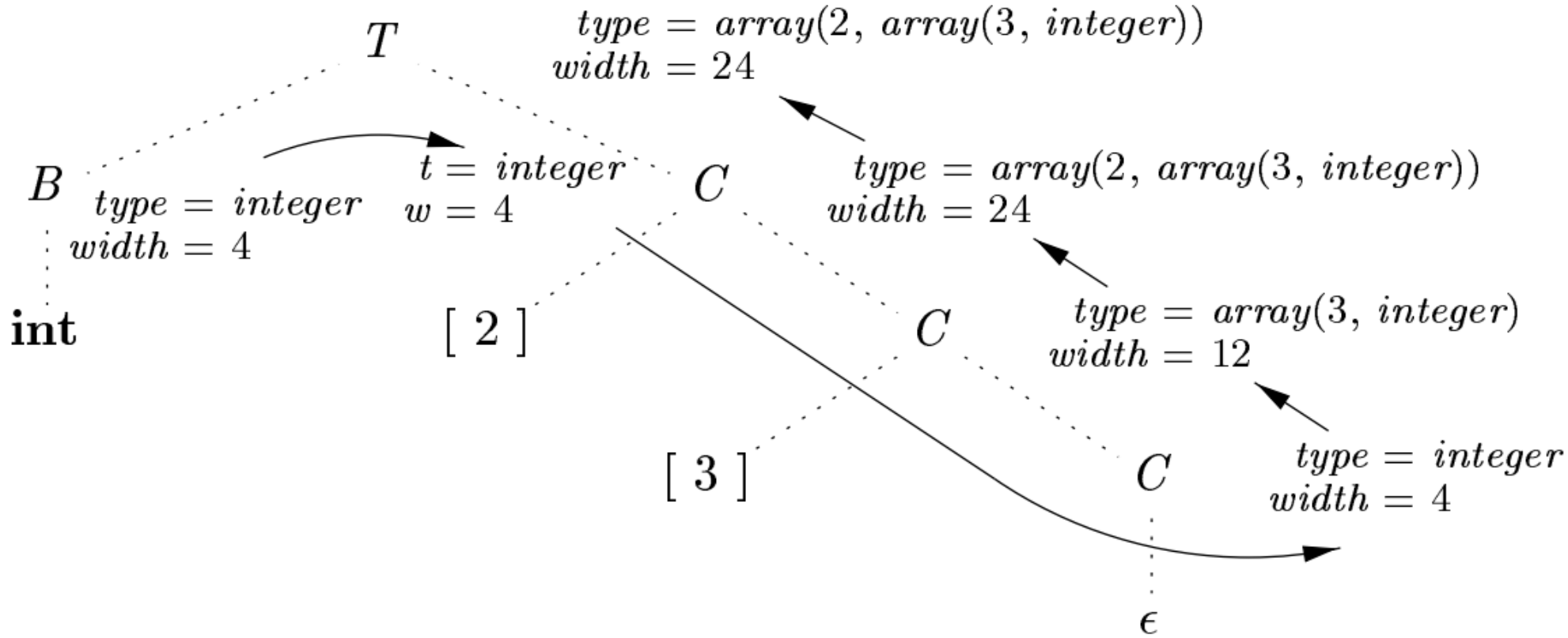


Figure 6.16: Syntax-directed translation of array types

Sequences of Declarations

The translation scheme of Fig. 6.17 deals with a sequence of declarations of the form T **id**, where T generates a type as in Fig. 6.15. Before the first declaration is considered, *offset* is set to 0. As each new name x is seen, x is entered into the symbol table with its relative address set to the current value of *offset*, which is then incremented by the width of the type of x.

Sequences of Declarations

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

Figure 6.17: Computing the relative addresses of declared names

Sequences of Declarations

The semantic action within the production $D \rightarrow T\ \mathbf{id}\ ;\ D_1$ creates a symbol-table entry by executing $top.put(\mathbf{id}.lexeme,\ T.type,\ offset)$. Here top denotes the current symbol table. The method $top.put$ creates a symbol-table entry for $\mathbf{id}.lexeme$, with type $T.type$ and relative address $offset$ in its data area.

The initialization of $offset$ in Fig. 6.17 is more evident if the first production appears on one line as:

$$P \quad \rightarrow \quad \{\ offset = 0;\ \}\ \ D \tag{6.1}$$

Nonterminals generating ϵ, called marker nonterminals, can be used to rewrite productions so that all actions appear at the ends of right sides; see Section 5.5.4. Using a marker nonterminal M, (6.1) can be restated as:

$$\begin{array}{lcll} P & \rightarrow & M\ D & \\ M & \rightarrow & \epsilon & \{\ offset = 0;\ \} \end{array}$$

Fields in Records and Classes

The translation of declarations in Fig. 6.17 carries over to fields in records and classes. Record types can be added to the grammar in Fig. 6.15 by adding the following production

$$T \rightarrow \textbf{record}\ '\{'\ D\ '\}'$$

Fields in Records and Classes

The fields in this record type are specified by the sequence of declarations generated by D. The approach of Fig. 6.17 can be used to determine the types and relative addresses of fields, provided we are careful about two things:

- The field names within a record must be distinct; that is, a name may appear at most once in the declarations generated by D.
- The offset or relative address for a field name is relative to the data area for that record.

Fields in Records and Classes

Example 6.10: The use of a name x for a field within a record does not conflict with other uses of the name outside the record. Thus, the three uses of `x` in the following declarations are distinct and do not conflict with each other:

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

A subsequent assignment `x = p.x+q.x;` sets variable `x` to the sum of the fields named `x` in the records `p` and `q`. Note that the relative address of `x` in `p` differs from the relative address of `x` in `q`. □

Fields in Records and Classes

The translation scheme in Fig. 6.18 consists of a single production to be added to the productions for T in Fig. 6.15. This production has two semantic actions. The embedded action before D saves the existing symbol table, denoted by *top* and sets *top* to a fresh symbol table. It also saves the current *offset*, and sets *offset* to 0. The declarations generated by D will result in types and relative addresses being put in the fresh symbol table. The action after D creates a record type using *top*, before restoring the saved symbol table and offset.

Fields in Records and Classes

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

Figure 6.18: Handling of field names in records

Translation of Expressions

- Operations within Expressions
- Incremental Translation
- Addressing Array Elements
- Translation of Array References

Translation of Expressions

- An expression with more than one operator, like a*b+c will translate into instructions with at most one operator per instruction
- An array reference a[i][j] will expand into a sequence of three-address instructions that calculate an address for the reference.

Operations within Expressions

- SDD builds up the three address code for an assignment statement S

Operations within Expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	
$E \rightarrow E_1 + E_2$	
$\mid\ - E_1$	
$\mid\ (E_1)$	
$\mid\ \mathbf{id}$	

Operations within Expressions

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

Figure 6.19: Three-address code for expressions

Operations within Expressions

Example 6.11: The syntax-directed definition in Fig. 6.19 translates the assignment statement `a = b + - c ;` into the three-address code sequence

```
t1 = minus c
t2 = b + t1
a = t2
```

Incremental Translation

The translation scheme in Fig. 6.20 generates the same code as the syntax-directed definition in Fig. 6.19. With the incremental approach, the *code* attribute is not used, since there is a single sequence of instructions that is created by successive calls to *gen*. For example, the semantic rule for $E \rightarrow E_1 + E_2$ in Fig. 6.20 simply calls *gen* to generate an add instruction; the instructions to compute E_1 into $E_1.addr$ and E_2 into $E_2.addr$ have already been generated.

The approach of Fig. 6.20 can also be used to build a syntax tree. The new semantic action for $E \rightarrow E_1 + E_2$ creates a node by using a constructor, as in

$$E \rightarrow E_1 + E_2 \quad \{\ E.addr = \textbf{new}\ Node('+', E_1.addr, E_2.addr);\ \}$$

Here, attribute *addr* represents the address of a node rather than a variable or constant.

Incremental Translation

$S \rightarrow \mathbf{id} = E\ ; \quad \{ \quad \}$

$E \rightarrow E_1 + E_2 \quad \{ \quad \}$

$\mid \quad - E_1 \quad \{ \quad \}$

$\mid \quad (E_1) \quad \{ \quad \}$

$\mid \quad \mathbf{id} \quad \{ \quad \}$

Incremental Translation

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

Figure 6.20: Generating three-address code for expressions incrementally

Addressing Array Elements

If the width of each array element is w, then the ith element of array A begins in location

$$base + i \times w \tag{6.2}$$

where *base* is the relative address of the storage allocated for the array. That is, *base* is the relative address of $A[0]$.

Addressing Array Elements

The formula (6.2) generalizes to two or more dimensions. In two dimensions, let us write $A[i_1][i_2]$, as in C, for element i_2 in row i_1. Let w_1 be the width of a row and let w_2 be the width of an element in a row. The relative address of $A[i_1][i_2]$ can then be calculated by the formula

$$base + i_1 \times w_1 + i_2 \times w_2 \tag{6.3}$$

In k dimensions, the formula is

$$base + i_1 \times w_1 + i_2 \times w_2 + \cdots + i_k \times w_k \tag{6.4}$$

where w_j, for $1 \leq j \leq k$, is the generalization of w_1 and w_2 in (6.3).

Addressing Array Elements

Alternatively, the relative address of an array reference can be calculated in terms of the numbers of elements n_j along dimension j of the array and the width $w = w_k$ of a single element of the array. In two dimensions (i.e., $k = 2$ and $w = w_2$), the location for $A[i_1][i_2]$ is given by

$$base + (i_1 \times n_2 + i_2) \times w \tag{6.5}$$

In k dimensions, the following formula calculates the same address as (6.4):

$$base + ((\cdots((i_1 \times n_2 + i_2) \times n_3 + i_3)\cdots) \times n_k + i_k) \times w \tag{6.6}$$

Addressing Array Elements

one-dimensional array, the array elements are numbered $low, low+1, \ldots, high$ and $base$ is the relative address of $A[low]$. Formula (6.2) for the address of $A[i]$ is replaced by:

$$base + (i - low) \times w \tag{6.7}$$

The expressions (6.2) and (6.7) can be both be rewritten as $i \times w + c$, where the subexpression $c = base - low \times w$ can be precalculated at compile time. Note that $c = base$ when low is 0.

Addressing Array Elements

The above address calculations are based on row-major layout for arrays, which is used in C, for example. A two-dimensional array is normally stored in one of two forms, either *row-major* (row-by-row) or *column-major* (column-by-column). Figure 6.21 shows the layout of a 2×3 array A in (a) row-major form and (b) column-major form. Column-major form is used in the Fortran family of languages.

(a) Row Major (b) Column Major

Figure 6.21: Layouts for a two-dimensional array.

Translation of Array References

Let nonterminal L generate an array name followed by a sequence of index expressions:

$$L \rightarrow L\ [\ E\ ]\ \mid\ \mathbf{id}\ [\ E\ ]$$

Translation of Array References

$$
\begin{aligned}
S &\rightarrow \mathbf{id} = E\ ; \\
&\mid\ L = E\ ; \\
E &\rightarrow E_1 + E_2 \\
&\mid\ \mathbf{id} \\
&\mid\ L
\end{aligned}
$$

Translation of Array References

$$L \rightarrow \mathbf{id}\ [\ E\ ]$$

$$\mid \quad L_1\ [\ E\ ]$$

Translation of Array References

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

Figure 6.22: Semantic actions for array references

Translation of Array References

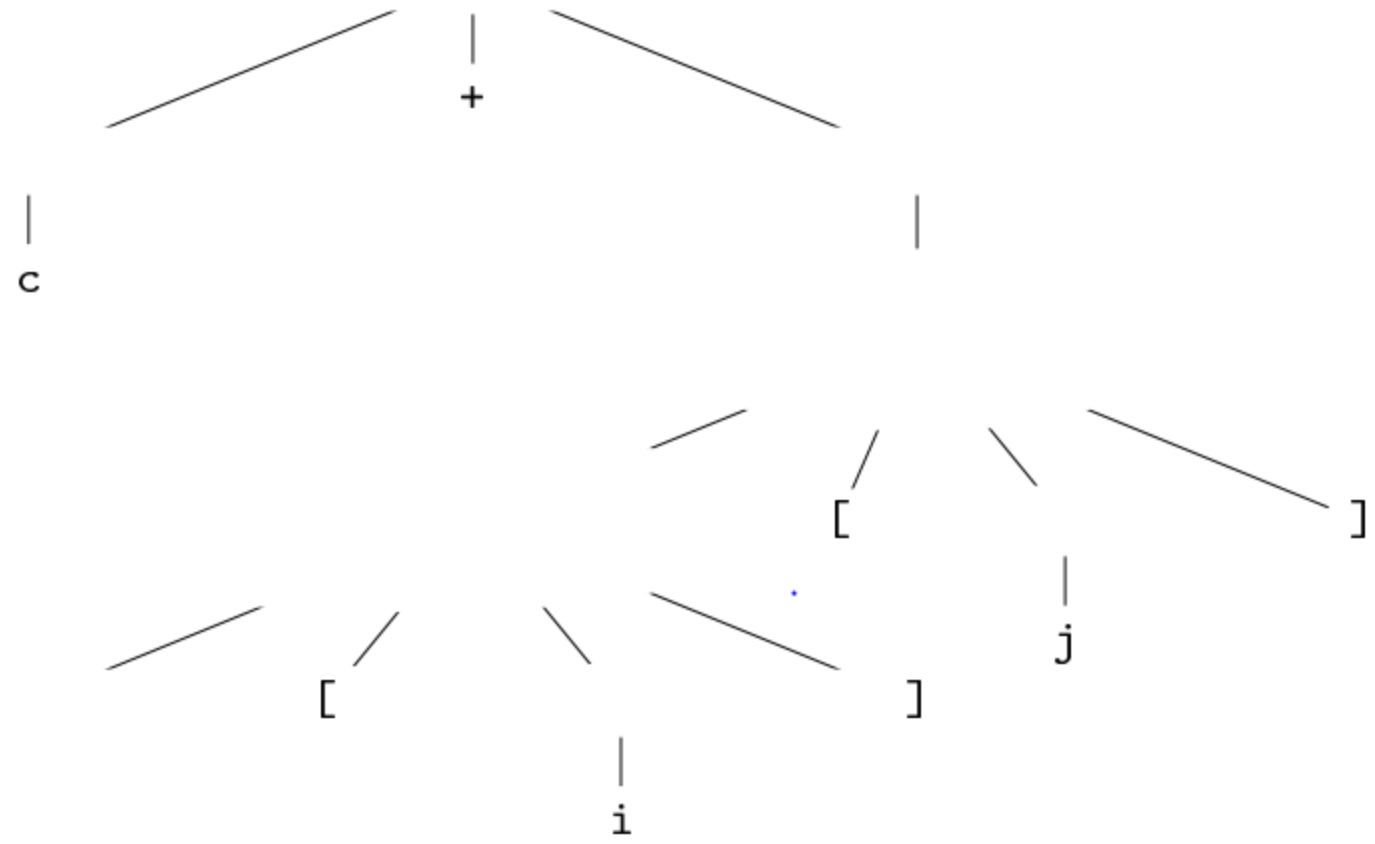
Example 6.12: Let `a` denote a 2×3 array of integers, and let `c`, `i`, and `j` all denote integers. Then, the type of `a` is $array(2, array(3, integer))$. Its width w is 24, assuming that the width of an integer is 4. The type of `a[i]` is $array(3, integer)$, of width $w_1 = 12$. The type of `a[i][j]` is $integer$.

An annotated parse tree for the expression `c + a[i][j]` is shown in Fig. 6.23. The expression is translated into the sequence of three-address instructions in Fig. 6.24. As usual, we have used the name of each identifier to refer to its symbol-table entry. □

Translation of Array References

Translation of Array References

```
c+a[i][j]
```



Translation of Array References

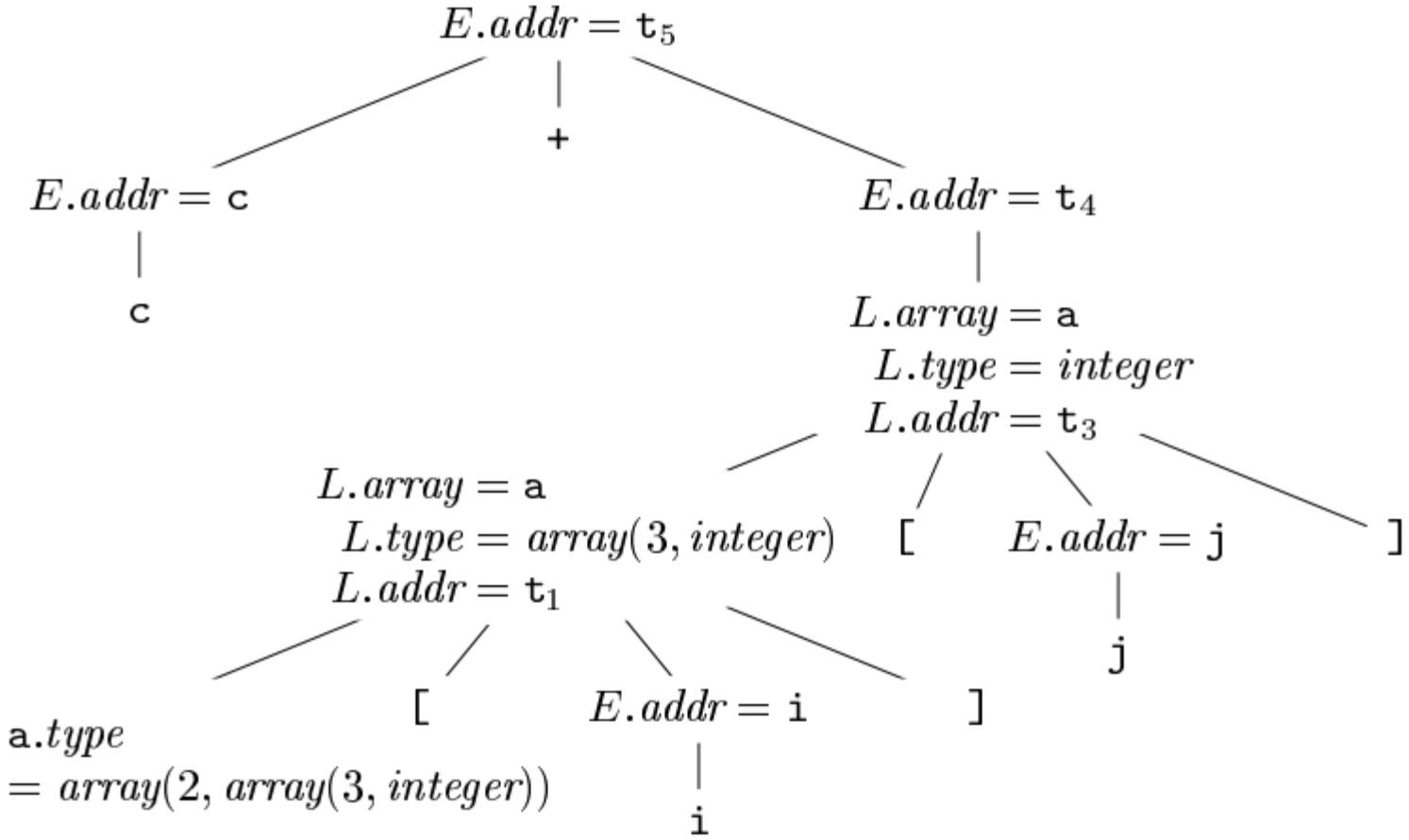


Figure 6.23: Annotated parse tree for `c+a[i][j]`

Translation of Array References

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```

Figure 6.24: Three-address code for expression `c+a[i][j]`

Control Flow

- Boolean Expressions
- Short-Circuit code
- Flow of control statements
- Control Flow translation of Boolean expressions
- Avoiding Redundant Gotos
- Boolean Values and Jumping Code

Boolean Expressions

$$B \rightarrow B \;||\; B \mid B \;\&\&\; B \mid !\, B \mid (\, B\,) \mid E \;\textbf{rel}\; E \mid \textbf{true} \mid \textbf{false}$$

We use the attribute **rel**.*op* to indicate which of the six comparison operators <, <=, =, ! =, >, or >= is represented by **rel**. As is customary, we assume that || and && are left-associative, and that || has lowest precedence, then &&, then !.

Short-Circuit code

In *short-circuit* (or *jumping*) code, the boolean operators &&, ||, 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

Example 6.21 : The statement

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

might be translated into the code of Fig. 6.34. In this translation, the boolean expression is true if control reaches label L_2. If the expression is false, control goes immediately to L_1, skipping L_2 and the assignment `x = 0`. □

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

Figure 6.34: Jumping code

Flow of control statements

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

Flow of control statements

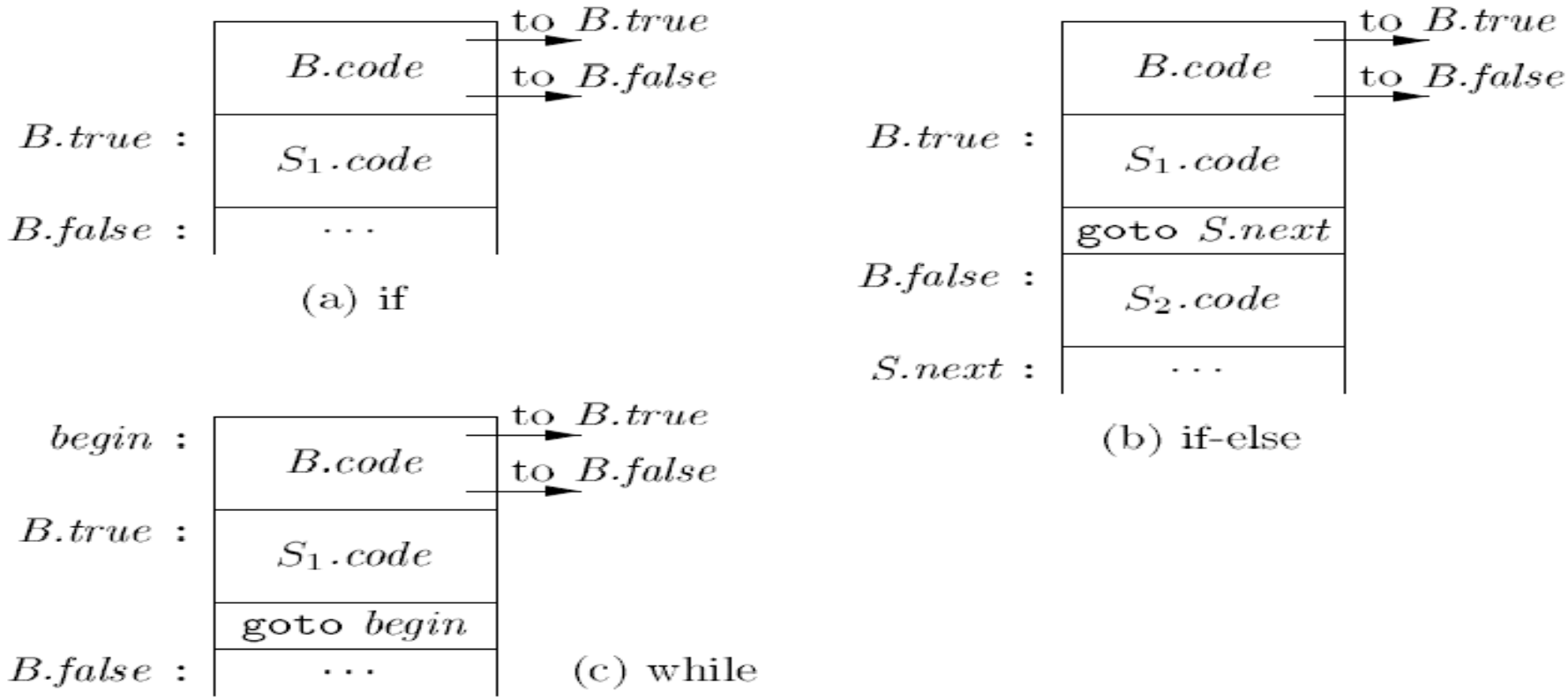


Figure 6.35: Code for if-, if-else-, and while-statements

Flow of control statements

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

Flow of control statements

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

Figure 6.36: Syntax-directed definition for flow-of-control statements.

Control Flow translation of Boolean expressions

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

Figure 6.37: Generating three-address code for booleans

Control Flow translation of Boolean expressions

The fourth production in Fig. 6.37, $B \rightarrow E_1$ **rel** E_2, is 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

Example 6.22: Consider again the following statement from Example 6.21:

```
if( x < 100 || x > 200 && x != y ) x = 0;          (6.13)
```

Control Flow translation of Boolean expressions

Using the syntax-directed definitions in Figs. 6.36 and 6.37 we would obtain the code in Fig. 6.38.

```
        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 :
```

Figure 6.38: Control-flow translation of a simple if-statement

Avoiding Redundant Gotos

```
        if x > 200 goto L4
        goto L1
L4:     ...
```

Instead, consider the instruction:

```
        ifFalse x > 200 goto L1
L4:     ...
```

Avoiding Redundant Gotos

$$
\begin{aligned}
B_1.true &= \textbf{if } B.true \neq fall \textbf{ then } B.true \textbf{ else } newlabel() \\
B_1.false &= fall \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= \textbf{if } B.true \neq fall \textbf{ then } B_1.code \mid\mid B_2.code \\
&\quad \textbf{else } B_1.code \mid\mid B_2.code \mid\mid label(B_1.true)
\end{aligned}
$$

Figure 6.40: Semantic rules for $B \rightarrow B_1 \mid\mid B_2$

Avoiding Redundant Gotos

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

translates into the code of Fig. 6.41.

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

Figure 6.41: If-statement translated using the fall-through technique

Boolean Values and Jumping Code

1. *Use two passes.* Construct a complete syntax tree for the input, and then walk the tree in depth-first order, computing the translations specified by the semantic rules.

2. *Use one pass for statements, but two passes for expressions.* With this approach, we would translate E in **while** (E) S_1 before S_1 is examined. The translation of E, however, would be done by building its syntax tree and then walking the tree.

The following grammar has a single nonterminal E for expressions:

$$S \rightarrow \textbf{id} = E \;; \mid \textbf{if}\,(\,E\,)\,S \mid \textbf{while}\,(\,E\,)\,S \mid S\,S$$
$$E \rightarrow E \,||\, E \mid E \,\&\&\, E \mid E\ \textbf{rel}\ E \mid E + E \mid (E) \mid \textbf{id} \mid \textbf{true} \mid \textbf{false}$$

Nonterminal E governs the flow of control in $S \rightarrow \textbf{while}\ (E)\ S_1$. The same nonterminal E denotes a value in $S \rightarrow \textbf{id} = E\,;$ and $E \rightarrow E + E$.

Boolean Values and Jumping Code

For example, the assignment `x = a<b && c<d` can be implemented by the code in Fig. 6.42.

```
        ifFalse a < b goto L1
        ifFalse c < d goto L1
        t = true
        goto L2
L1:     t = false
L2:     x = t
```

Figure 6.42: Translating a boolean assignment by computing the value of a temporary

Backpatching

- One-Pass Code Generation using Backpatching
- Backpatching for Boolean Expressions
- Flow of control statements

One-Pass Code Generation using Backpatching

Backpatching can be used to generate code for boolean expressions and flow-of-control statements in one pass.

In this section, synthesized attributes *truelist* and *falselist* of nonterminal B are used to manage labels in jumping code for boolean expressions. In particular, $B.truelist$ will be a list of jump or conditional jump instructions into which we must insert the label to which control goes if B is true.

One-Pass Code Generation using Backpatching

1. *makelist*(i) creates a new list containing only i, an index into the array of instructions; *makelist* returns a pointer to the newly created list.
2. *merge*(p_1, p_2) concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.

3. *backpatch*(p, i) inserts i as the target label for each of the instructions on the list pointed to by p.

Backpatching for Boolean Expressions

We now construct a translation scheme suitable for generating code for boolean expressions during bottom-up parsing. A marker nonterminal M in the grammar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated. The grammar is as follows:

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

Backpatching for Boolean Expressions

The translation scheme is in Fig. 6.43.

1) $B \rightarrow B_1$ `||` $M\ B_2$ { $backpatch(B_1.falselist, M.instr)$;
$B.truelist = merge(B_1.truelist, B_2.truelist)$;
$B.falselist = B_2.falselist$; }

2) $B \rightarrow B_1$ `&&` $M\ B_2$ { $backpatch(B_1.truelist, M.instr)$;
$B.truelist = B_2.truelist$;
$B.falselist = merge(B_1.falselist, B_2.falselist)$; }

3) $B \rightarrow$ `!` B_1 { $B.truelist = B_1.falselist$;
$B.falselist = B_1.truelist$; }

4) $B \rightarrow$ `(` B_1 `)` { $B.truelist = B_1.truelist$;
$B.falselist = B_1.falselist$; }

5) $B \rightarrow E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr)$;
$B.falselist = makelist(nextinstr + 1)$;
$gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_')$;
$gen('\texttt{goto}\ \_')$; }

Backpatching for Boolean Expressions

6)	$B \rightarrow \textbf{true}$	$\{\ B.truelist = makelist(nextinstr);$ $gen('\texttt{goto}\ \_');\ \}$
7)	$B \rightarrow \textbf{false}$	$\{\ B.falselist = makelist(nextinstr);$ $gen('\texttt{goto}\ \_');\ \}$
8)	$M \rightarrow \epsilon$	$\{\ M.instr = nextinstr;\ \}$

Figure 6.43: Translation scheme for boolean expressions

Backpatching for Boolean Expressions

Example 6.24: Consider again the expression

$$x < 100 \;||\; x > 200 \;\&\&\; x \;!= \; y$$

An annotated parse tree is shown in Fig. 6.44; for readability, attributes *truelist*, *falselist*, and *instr* are represented by their initial letters. The actions are performed during a depth-first traversal of the tree.

Backpatching for Boolean Expressions

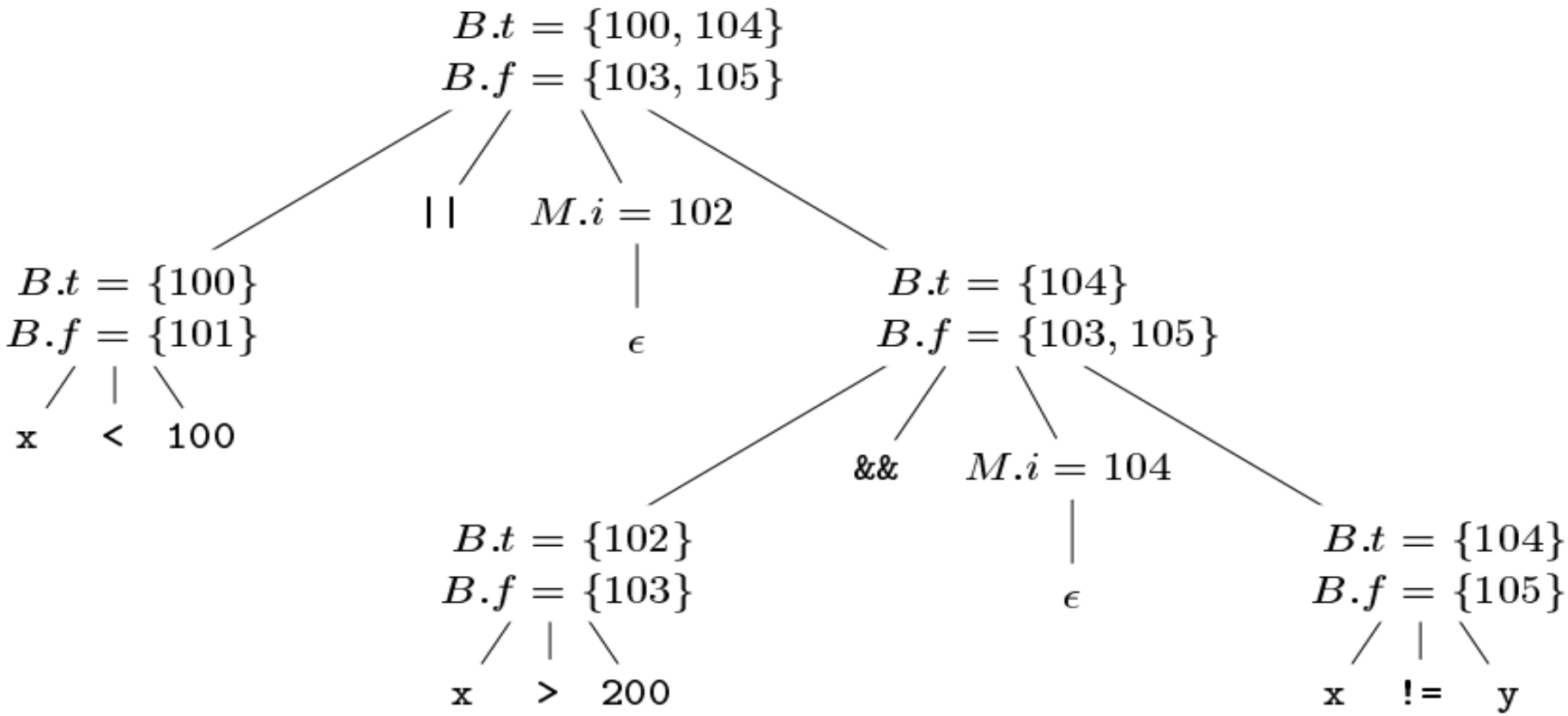


Figure 6.44: Annotated parse tree for $x < 100$ || $x > 200$ && x != y

Backpatching for Boolean Expressions

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

are generated. (We arbitrarily start instruction numbers at 100.) The marker nonterminal M in the production

$$B \rightarrow B_1 \ || \ M \ B_2$$

records the value of *nextinstr*, which at this time is 102. The reduction of $x > 200$ to B by production (5) generates the instructions

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

The subexpression $x > 200$ corresponds to B_1 in the production

$$B \rightarrow B_1 \ \&\& \ M \ B_2$$

The marker nonterminal M records the current value of *nextinstr*, which is now 104. Reducing $x \ != \ y$ into B by production (5) generates

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

Backpatching for Boolean Expressions

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

(a) After backpatching 104 into instruction 102.

Backpatching for Boolean Expressions

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

(b) After backpatching 102 into instruction 101.

Figure 6.45: Steps in the backpatch process

Backpatching for Boolean Expressions

We now reduce by $B \rightarrow B_1 \;\&\&\; M\; B_2$. The corresponding semantic action calls *backpatch*(B_1.*truelist*, M.*instr*) to bind the true exit of B_1 to the first instruction of B_2. Since B_1.*truelist* is {102} and M.*instr* is 104, this call to *backpatch* fills in 104 in instruction 102. The six instructions generated so far are thus as shown in Fig. 6.45(a).

The semantic action associated with the final reduction by $B \rightarrow B_1 \;||\; M\; B_2$ calls *backpatch*({101},102) which leaves the instructions as in Fig. 6.45(b).

The entire expression is true if and only if the gotos of instructions 100 or 104 are reached, and is false if and only if the gotos of instructions 103 or 105 are reached. These instructions will have their targets filled in later in the compilation, when it is seen what must be done depending on the truth or falsehood of the expression. □

Flow of control statements

We now use backpatching to translate flow-of-control statements in one pass. Consider statements generated by the following grammar:

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

Here S denotes a statement, L a statement list, A an assignment-statement, and B a boolean expression.

Flow of control statements

Consider the semantic action (3) in Fig. 6.46. The code layout for production $S \rightarrow \textbf{while}\ (\ B\ )\ S_1$ is as in Fig. 6.35(c). The two occurrences of the marker nonterminal M in the production

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

record the instruction numbers of the beginning of the code for B and the beginning of the code for S_1. The corresponding labels in Fig. 6.35(c) are *begin* and $B.true$, respectively.

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

Flow of control statements

4)	$S \rightarrow \texttt{\{}\ L\ \texttt{\}}$	$\{\ S.nextlist\ =\ L.nextlist;\ \}$
5)	$S \rightarrow A\ \texttt{;}$	$\{\ S.nextlist\ =\ \textbf{null};\ \}$
6)	$M \rightarrow \epsilon$	$\{\ M.instr\ =\ nextinstr;\ \}$
7)	$N \rightarrow \epsilon$	$\{\ N.nextlist\ =\ makelist(nextinstr);$ $gen('\texttt{goto}\ \_');\ \}$
8)	$L \rightarrow L_1\ M\ S$	$\{\ backpatch(L_1.nextlist,\ M.instr);$ $L.nextlist\ =\ S.nextlist;\ \}$
9)	$L \rightarrow S$	$\{\ L.nextlist\ =\ S.nextlist;\ \}$

Figure 6.46: Translation of statements

Flow of control statements

Thank You