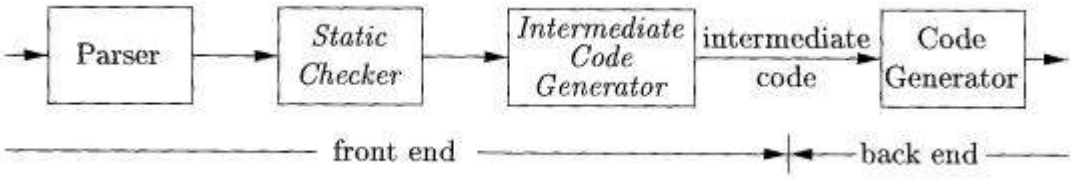
UNIT – IV

Intermediated Code: Generation Variants of Syntax trees 3Addresscode, Types and Deceleration, Translation of Expressions, Type Checking. Canted Flow Back patching?

UNIT 4
INTERMEDIATE CODE

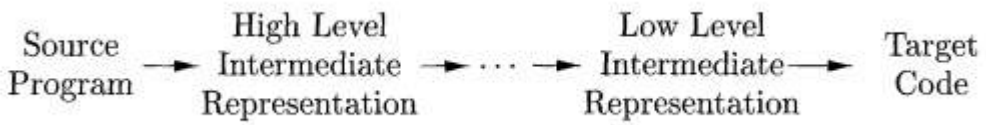
In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. This facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end.

Logical Structure of a Compiler Front End



A compiler front end is organized as in figure above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and traversing the tree.

Static checking includes *type checking,* which ensures that operators are applied to compatible operands. In the process of translating a program in a given source language into code for a given target machine, a compiler construct a sequence of intermediate representations



Sequence of intermediate representations

High-level representations are close to the source language and low-level representations are close to the target machine. A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection.

Variants of Syntax Trees

1 Directed Acyclic Graphs for Expressions

2 The Value-Number Method for Constructing DAG's

1. Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators. The difference is that a node *N* in a DAG has more than one parent if *N* represents a com-mon 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.

Example: Consider expression

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

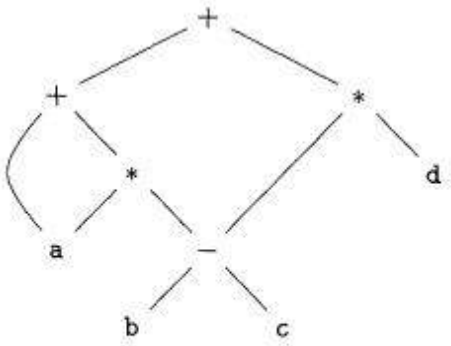
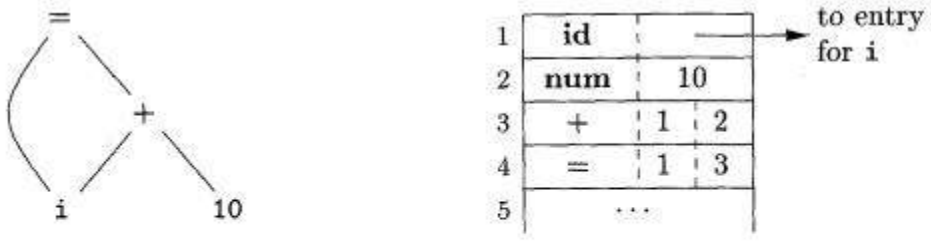


Figure 6.3: Dag for the expression a+a*(b-c)+(b-c)*d

2 The Value-Number Method for Constructing DAG's

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.



(a) DAG (b) Array.

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

In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer is called the value number for the node.

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

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

OUTPUT : The value number of a node in the array with signature (op, l,r).

METHOD : Search the array for a node M with label op, left child I, 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 I, and right child r, and return its value number.

Three-Address Code

1 Addresses and Instructions

2 Quadruples

3 Triples

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a 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 ti and t_2 are compiler-generated temporary names.

1 Addresses and Instructions

An address can be one of the following:

- *A name.* 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.* A compiler must deal with many different types of constants and variables.
- *A compiler-generated temporary*. Useful in optimizing com-pilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

common three-address instruction

1. Assignment Statement: x = y op z and x = op y

Here,x, y and z are the operands. op represents the operator.

2. Copy Statement: x = y

3. Conditional Jump: If x relop y goto X

If the condition "x relop y" gets **satisfied,** then-

The control is sent directly to the location specified by label X.

All the statements in between are skipped.

If the condition "x relop y" **fails**, then-

The control is not sent to the location specified by label X.

The next statement appearing in the usual sequence is executed.

4. Unconditional Jump- goto X

On executing the statement,The control is sent directly to the location specified by label X.

All the statements in between are skipped.

5. Procedure Call- param x call p return y

Here, p is a function which takes x as a parameter and returns y.

For a procedure call p(x1, …, xn)

```
param x1
…
param xn
call p, n
```

6.Indexed copy instructions: x = y[i] and x[i] = y

Left: sets x to the value in the location i memory units beyond y

Right: sets the contents of the location i memory units beyond x to y

7.Address and pointer instructions:

x = &y sets the value of x to be the location (address) of y.

x = *y, presumably y is a pointer or temporary whose value is a location. The value of x is set to the contents of that location.

*x = y sets the value of the object pointed to by x to the value of y.

Data Structure

Three address code is represented as record structure with fields for operator and operands. These records can be stored as array or linked list. Most common implementations of three address code are Quadruples, Triples and Indirect triples.

2. Quadruples

Quadruples consists of four fields in the record structure. One field to store operator op, two fields to store operands or arguments arg1and arg2 and one field to store result res.

res = arg1 op arg2

Example: a = b + c

b is represented as arg1, c is represented as arg2, + as op and a as res.

Unary operators like „-„do not use agr2. Operators like param do not use agr2 nor result. For conditional and unconditional statements res is label. Arg1, arg2 and res are pointers to symbol table or literal table for the names.

Example: a = -b * d + c + (-b) * d

Three address code for the above statement is as follows

t1 = - b

t2 = t1 * d

t3 = t2 + c

t4 = - b

t5 = t4 * d

t6 = t3 + t5

a = t6

Quadruples for the above example is as follows

Op	Arg1	Arg2	Res
-	B		t1
*	t1	d	t2
+	t2	c	t3
-	B		t4
*	t4	d	t5
+	t3	t5	t6
=	t6		a

3 TRIPLES

Triples uses only three fields in the record structure. One field for operator, two fields for operands named as arg1 and arg2. Value of temporary variable can be accessed by the position of the statement the computes it and not by location as in quadruples.

Example: a = -b * d + c + (-b) * d

Triples for the above example is as follows

Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Arg1 and arg2 may be pointers to symbol table for program variables or literal table for constant or pointers into triple structure for intermediate results.

Example: Triples for statement x[i] = y which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	[]=	x	i
(1)	=	(0)	y

Triples for statement x = y[i] which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	=[]	y	i
(1)	=	x	(0)

Triples are alternative ways for representing syntax tree or Directed acyclic graph for program defined names.

Indirect Triples

Indirect triples are used to achieve indirection in listing of pointers. That is, it uses pointers to triples than listing of triples themselves.

Example: a = -b * d + c + (-b) * d

	Stmt no
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

Stmt no	Op	Arg1	Arg2
(10)	-	b	
(11)	*	d	(0)
(12)	+	c	(1)
(13)	-	b	
(14)	*	d	(3)
(15)	+	(2)	(4)
(16)	=	a	(5)

Types and Declarations

1 Type Expressions

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.

Definition

- 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.
- If *s* and *t* are type expressions, then their Cartesian product *s* x *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).
- Type expressions may contain variables whose values are type expressions

2 Type Equivalence

Many type-checking rules have the form, "if two type expressions are equal then return a certain type else error." Potential ambiguities arise when names are given to type expressions. The key issue is whether a name in a type expression stands for itself or whether it is an abbreviation for another type expression.

Since type names denote type expressions, they can set up implicit cycles; see the box on "Type Names and Recursive Types." If edges to type names are redirected to the type expressions denoted by the names, then the resulting graph can have cycles due to recursive types.

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.

If type names are treated as standing for themselves, then the first two conditions in the above definition lead to *name equivalence* of type expressions.

Name-equivalent expressions are assigned the same value number,. Structural equivalence can be tested using the unification algorithm .

3. Declarations

Understand types and declarations using a simplified grammar that declares just one name at a time; The grammar is

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

The fragment of the above grammar that deals with basic and array types.Consider storage layout as well as types.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.

4. 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.

Address Alignment

The storage layout for data objects is strongly influenced by the address-ing constraints of the target machine. For example, instructions to add integers may expect integers to be *aligned,* that is, placed at certain positions in memory such as an address divisible by 4. Although an array of ten characters needs only enough bytes to hold ten characters, a compiler may therefore allocate 12 bytes — the next multiple of 4 — leaving 2 bytes unused. Space left unused due to alignment considerations is referred to as *padding.* When space is at a premium, a compiler may *pack* data so that no padding is left; additional instructions may then need to be executed at run time to position packed data so that it can be operated on as if it were properly aligned.

Suppose that storage comes in blocks of contiguous bytes, where a byte is the smallest unit of addressable memory. 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.

The translation scheme (SDT) computes types and their widths for basic and array types; 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* —> e. In a syntax-directed definition, *t* and *w* would be inherited attributes for *C.*

The body of the T-production consists of nonterminal B, an action, and nonterminal C, which appears on the next line. The action between B and C sets t to B.type and w to B. width. If B —>• int then B.type is set to integer and B.width is set to 4, the width of an integer. Similarly, if B -+ float then B.type is float and B.width is 8, the width of a float.

The productions for C determine whether T generates a basic type or an array type. If C —>• e, then t becomes C.type and w becomes C. width. Otherwise, C specifies an array component. The action for C —> [n u m] C1 forms C.type by applying the type constructor array to the operands num.value and C1.type.

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

Figure 6.15: Computing types and their widths

The width of an array is obtained by multiplying the width of an element by the number of elements in the array. If addresses of consecutive integers differ by 4, then address calculations for an array of integers will include multiplications by 4. Such multiplications provide opportunities for optimization, so it is helpful for the front end to make them explicit.

Example The parse tree for the type i n t [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 —> e to start the evaluation of the synthesized attributes up the chain of C nodes.

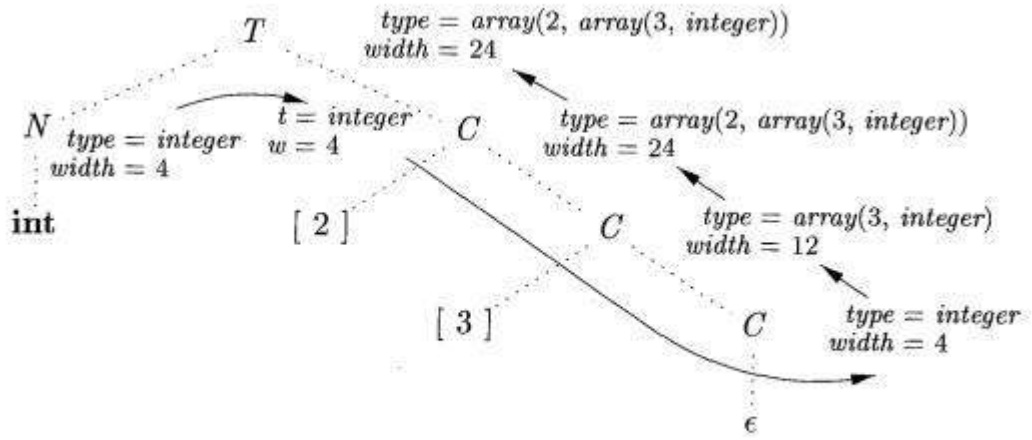


Figure 6.16: Syntax-directed translation of array types

Translations of Expressions

1 Operations Within Expressions
2 Incremental Translation
3 Addressing Array Elements
4 Translation of Array References

1 Operations Within Expressions

The syntax-directed definition builds up the three-address code for an assignment statement *S* using attribute *code* for *S* and attributes *addr* and *code* for anexpression *E.* Attributes *S.code* and *E.code* denote the three-address code for *S* and *E,* respectively. Attribute *E.addr* denotes the address that will hold the value of E.

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

Example The syntax-directed definition in Fig. 6.19 translates the assignment statement a = b + - c; into the TAC

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

2 Incremental Translation

Code attributes can be long strings, so they are generated incrementally In the incremental approach, *gen* not only constructs a three-address instruction, it appends the instruction to the sequence of instructions generated so far. The sequence may either be retained in memory for further processing, or it may be output incrementally.attribute addr represents the address of a node rather than a variable or constant.

$S \rightarrow$	**id** = E ;	{ *gen*(*top.get*(**id**.*lexeme*) '=' E.*addr*); }
$E \rightarrow$	E_1 + E_2	{ E.*addr* = **new** *Temp*(); *gen*(E.*addr* '=' E_1.*addr* '+' E_2.*addr*); }
|	- E_1	{ E.*addr* = **new** *Temp*(); *gen*(E.*addr* '=' '**minus**' E_1.*addr*); }
|	(E_1)	{ E.*addr* = E_1.*addr*; }
|	**id**	{ E.*addr* = *top.get*(**id**.*lexeme*); }

Figure 6.20: Generating three-address code for expressions incrementally

3.Addressing Array Elements

Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive location. Array can be one dimensional or two dimensional.

For one dimensional array:

A: array[low..high] of the ith elements is at:

base + (i-low)*width = i*width + (base - low*width)

Multi-dimensional arrays:

Row major or column major forms

- Row major: a[1,1], a[1,2], a[1,3], a[2,1], a[2,2], a[2,3]
- Column major: a[1,1], a[2,1], a[1, 2], a[2, 2],a[1, 3],a[2,3]
- In row major form, the address of a[i1, i2] is
- Base+((i1-low1)*(high2-low2+1)+i2-low2)*width

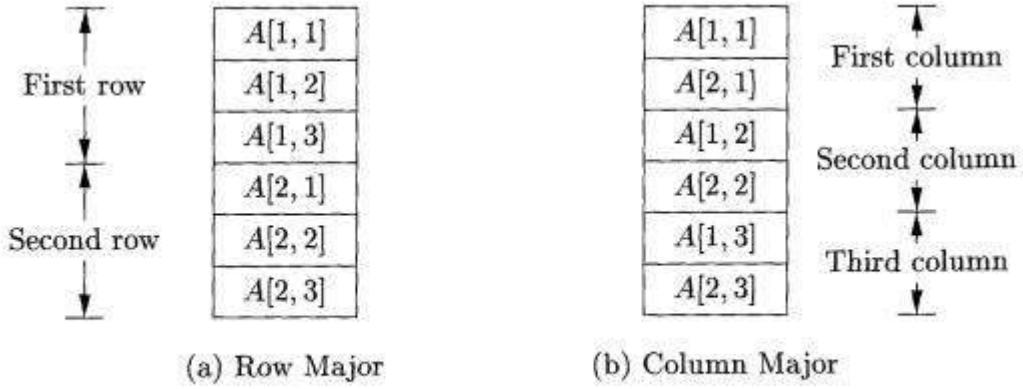


Figure 6.21: Layouts for a two-dimensional array.

4 Translation of Array References

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

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

Calculate addresses based on widths, using the formula rather than on numbers of elements. The translation scheme generates three-address code for expressions with array references. It consists of the productions and semantic actions together with productions involving nonterminal

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

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

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

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

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

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

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

Figure 6.22: Semantic actions for array references

.

Type Checking

1 Rules for Type Checking

2 Type Conversions

3 Overloading of Functions and Operators

4 Type Inference and Polymorphic Functions

5 An Algorithm for Unification

Type checking a compiler needs to assign a type expression to each component of the source program. The compiler must then determine that these type expressions conform to a collection of logical rules that is called the type system for the source language.

1 Rules for Type Checking

Type checking can take on two forms: **synthesis and inference**. *Type synthesis* builds up the type of an expression from the types of its subexpressions. It requires names to be declared before they are used. The type of *E1 + E2* is defined in terms of the types of *E1* and *E2* • A typical rule for type synthesis has the form

$$\begin{array}{l}\textbf{if } f \text{ has type } s \to t \textbf{ and } x \text{ has type } s, \\ \textbf{then} \text{ expression } f(x) \text{ has type } t\end{array} \qquad (6.8)$$

Type inference determines the type of a language construct from the way it is used.Rule for type inference has the form

$$\begin{array}{l}\textbf{if } f(x) \text{ is an expression,} \\ \textbf{then} \text{ for some } \alpha \text{ and } \beta, f \text{ has type } \alpha \to \beta \textbf{ and } x \text{ has type } \alpha\end{array} \qquad (6.9)$$

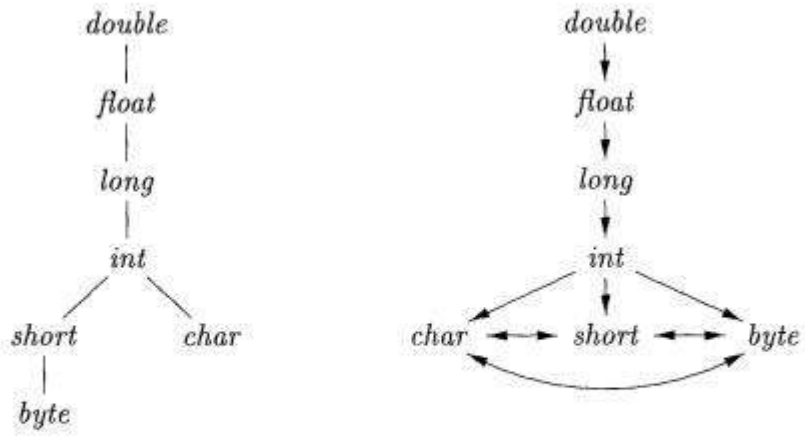
2 Type Conversions

integers are converted to floats when necessary, using a unary operator (f l o a t) . For example, the integer **2** is converted to a float in the code for the expression **2*3.14:**

```
t1 = (float) 2
t2 = t1 * 3.14
```

Type conversion rules vary from language to language. The rules for Java in Fig. 6.25 distinguish between *widening* conversions, which are intended to preserve information, and *narrowing* conversions, which can lose information.

Conversion from one type to another is said to be *implicit* if it is done automatically by the compiler. Implicit type conversions, also called *coercions,*



(a) Widening conversions (b) Narrowing conversions

Figure 6.25: Conversions between primitive types in Java

3 Overloading of Functions and Operators

An *overloaded* symbol has different meanings depending on its context. The + operator in Java denotes either string concatenation or addition

Type-synthesis rule for overloaded functions:

if f can have type $s_i \to t_i$, for $1 \leq i \leq n$, where $s_i \neq s_j$ for $i \neq j$
and x has type s_k, for some $1 \leq k \leq n$ (6.10)
then expression $f(x)$ has type t_k

4 Type Inference and Polymorphic Functions

Type inference is useful for a language like ML, which is strongly typed, but does not require names to be declared before they are used. Type inference ensures that names are used consistently.

The term "polymorphic" refers to any code fragment that can be executed with arguments of different types.

The type of length can be described as, "for any type a, length maps a list of elements of type a to an integer."

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

Figure 6.28: ML program for the length of a list

The program fragment defines function *length* with one parameter *x*. The body of the function consists of a conditional expression. The predefined function *null* tests whether a list is empty, and the predefined function *tl* (short for "tail") returns the remainder of a list after the first element is removed.

5 An Algorithm for Unification

Unification is the problem of determining whether two expressions s and t can be made identical by substituting expressions for the variables in s and t. Testing equality of expressions is a special case of unification; if s and t have constants but no variables, then s and t unify if and only if they are identical. so it can be used to test structural equivalence of circular types . [7]

Graph-theoretic formulation of unification, where types are represented by graphs. Type variables are represented by leaves and type constructors are represented by interior nodes. Nodes are grouped into equiv-alence classes; if two nodes are in the same equivalence class, then the type expressions they represent must unify. Thus, all interior nodes in the same class must be for the same type constructor, and their corresponding children must be equivalent.

Example 6.18 : Consider the two type expressions

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

The following substitution S is the most general unifier for these expressions

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

This substitution maps the two type expressions to the following expression

$$((\alpha_1 \to \alpha_2) \times list(\alpha_1)) \to list(\alpha_2)$$

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

Figure 6.32: Unification algorithm.

The unification algorithm, uses the following two operations on nodes:

find{n) returns the representative node of the equivalence class currently containing node *n.*

union(m, n) merges the equivalence classes containing nodes *m* and n. If one of the representatives for the equivalence classes of m and n is a non-variable node, *union* makes that nonvariable node be the representative for the merged equivalence class; otherwise, *union* makes one or the other of the original representatives be the new representative. This asymme-try in the specification of *union* is important because a variable cannot be used as the representative for an equivalence class for an expression containing a type constructor or basic type. Otherwise, two inequivalent expressions may be unified through that variable.

Control Flow

1 Boolean Expressions

2 Short-Circuit Code

3 Flow-of-Control Statements

4 Control-Flow Translation of Boolean Expressions

In programming languages, boolean expressions are often used to

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)* 5, 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. Such boolean

expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

1 Boolean Expressions

Boolean expressions are composed of the boolean operators (which we denote &&, I I, and !, using the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are boolean variables or relational expressions. Boolean expressions generated by the following grammar:

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

Given the expression Bi I I B2, 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 Bi is false, then the entire expression is false.

2 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.

Example The statement

i f (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_u 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

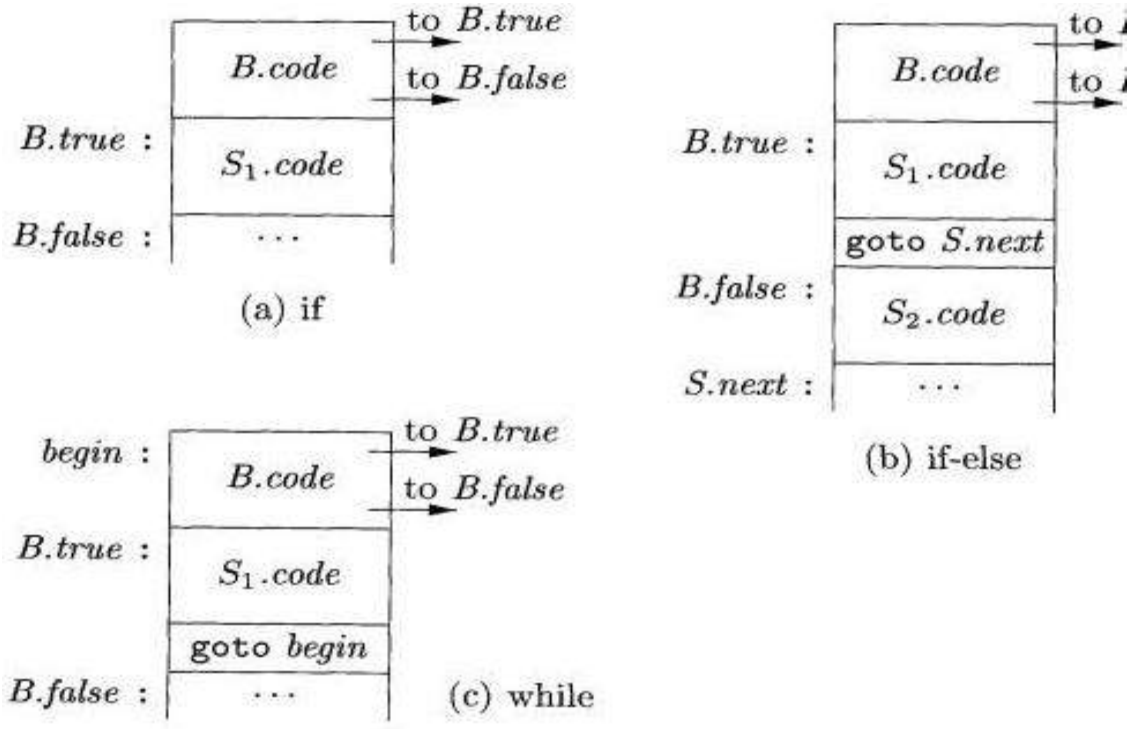
3 Flow-of-Control Statements

$$S \rightarrow \mathbf{if}\ (\ B\ )\ S_1$$
$$S \rightarrow \mathbf{if}\ (\ B\ )\ S_1\ \mathbf{else}\ S_2$$
$$S \rightarrow \mathbf{while}\ (\ B\ )\ S_1$$

In these productions, nonterminal *B* represents a boolean expression and non-terminal S represents a statement.

B and S have a synthesized attribute *code,* which gives the translation into three-address instructions. we build up the translations *B.code* and *S.code* as strings, using syntax directed definitions.

The translation of if (B) S1 consists of B.code followed by Si.code, as illustrated in Fig. 6.35(a). Within B.code are jumps based on the value of B. If B is true, control flows to the first instruction of Si.code, and if B is false, control flows to the instruction immediately following S1 . code.



Code for if, if else, while statements

The syntax-directed definition in Fig. 6.36-6.37 produces three-address code for boolean expressions in the context of if-, if-else-, and while-statements.

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

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

4 Control-Flow Translation of Boolean Expressions

Boolean expression *B* is translated into three-address instructions that evaluate *B* using creates labels only when they are needed. Alternatively, unnecessary labels can be eliminated during a subsequent optimization phase.

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

Figure 6.37: Generating three-address code for booleans

Backpatching

1 One-Pass Code Generation Using Backpatching

The problem in generating three address codes in a single pass is that we may not know the labels that control must go to at the time jump statements are generated. So to get around this

problem a series of branching statements with the targets of the jumps temporarily left unspecified is generated. Back Patching is putting the address instead of labels when the proper label is determined.

To manipulate lists of jumps, Back patching Algorithms perform three types of operations

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(pi,p2) concatenates the lists pointed to by p1 and p2, 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.

2 Backpatching for Boolean Expressions

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:

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

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

6) $B \rightarrow \mathbf{true}$ { $B.truelist = makelist(nextinstr)$;
$emit('\texttt{goto}\ \_')$; }

7) $B \rightarrow \mathbf{false}$ { $B.falselist = makelist(nextinstr)$;
$emit('\texttt{goto}\ \_')$; }

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

Figure 6.43: Translation scheme for boolean expressions

Consider semantic action (1) for the production B -> B1 || M B2. If Bx is true, then B is also true, so the jumps on Bi.truelist become part of B.truelist. If Bi is false, however, we must next test B2, so the target for the jumps B>i .falselist must be the beginning of the code generated for B2 • This target is obtained using the marker nonterminal M. That nonterminal produces, as a synthesized attribute M.instr, the index of the next instruction, just before B2 code starts being generated.

Example

Consider expression

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

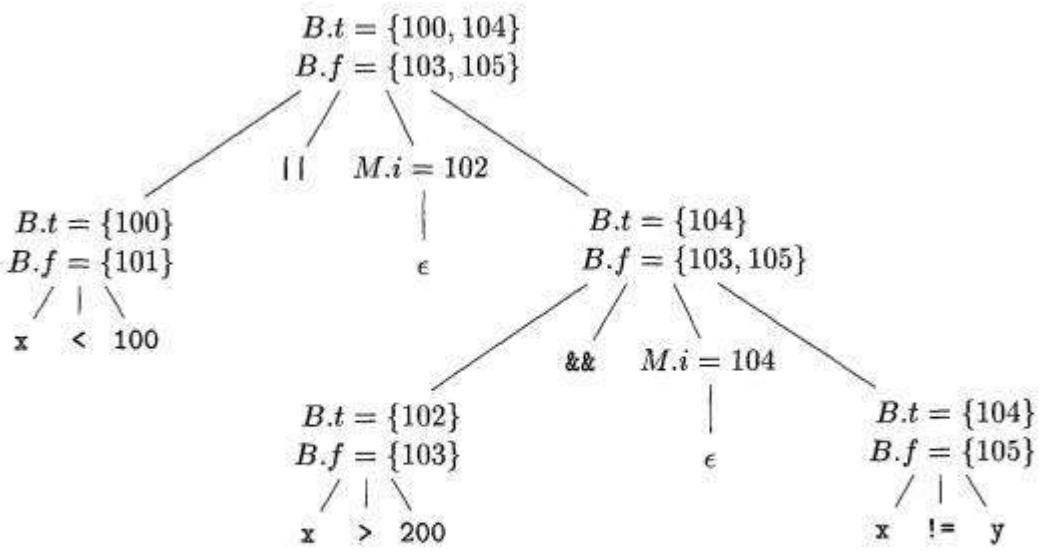


Figure 6.44: Annotated parse tree for $x < 100 \;||\; x > 200 \;\&\&\; x \;!= \; y$

An annotated parse tree is shown in Fig. 6.44; attributes *truelist, falselist,* and *instr* are represented by their initial letters. The actions are performed during a depth-first traversal of the tree. Since all actions appear at the ends of right sides, they can be performed in conjunction with reductions during a bottom-up parse. In response to the reduction of $x < 100$ to B by production (5), the two instructions

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

are generated. (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

Reducing *x ! = y* into *B* by production (5) generates

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

We now reduce by B —> B1 && M B2. The corresponding semantic action calls backpatch(B1.truelist,M.instr) to bind the true exit of B1 to the first instruction of B2. Since B1.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 —> B1 || MB2 calls backpatch({101},102) which leaves the instructions as in Fig

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

```
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.

```
100:  if x < 100 goto _
101:  goto 102
102:  if y > 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

of the expression. as

use backpatching to translate flow-of-control statements in one pass.

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

The translation scheme in Fig. 6.46 maintains lists of jumps that are filled in when their targets are found.

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('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('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

Backpatch the jumps when B is true to the instruction Mi.instr; the latter is the beginning of the code for Si. Similarly, we backpatch jumps when B is false to go to the beginning of the code for S 2 . The list S.nextlist includes all jumps out of Si and S 2 , as well as the jump generated by N. (Variable temp is a temporary that is used only for merging lists.

Semantic actions (8) and (9) handle sequences of statements. In

$$L \rightarrow L_1 \, M \, S$$

the instruction following the code for L± 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.

Note that 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.