CHAPTER 8

Intermediate Code Generation

In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back end generates target code. Details of the target language are confined to the back end, as far as possible. Although a source program can be translated directly into the target language, some benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated; a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation. Such optimizers are discussed in detail in Chapter 10.

This chapter shows how the syntax-directed methods of Chapters 2 and 5 can be used to translate into an intermediate form programming language constructs such as declarations, assignments, and flow-of-control statements. For simplicity, we assume that the source program has already been parsed and statically checked, as in the organization of Fig. 8.1. Most of the syntax-directed definitions in this chapter can be implemented during either bottom-up or top-down parsing using the techniques of Chapter 5, so intermediate code generation can be folded into parsing, if desired.

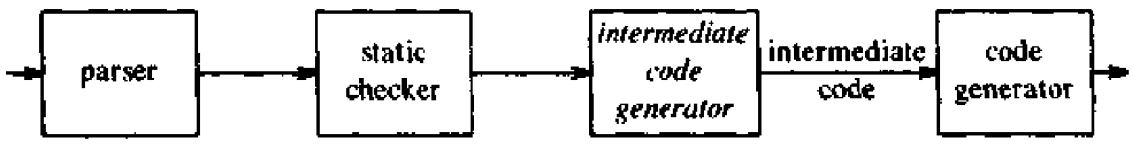


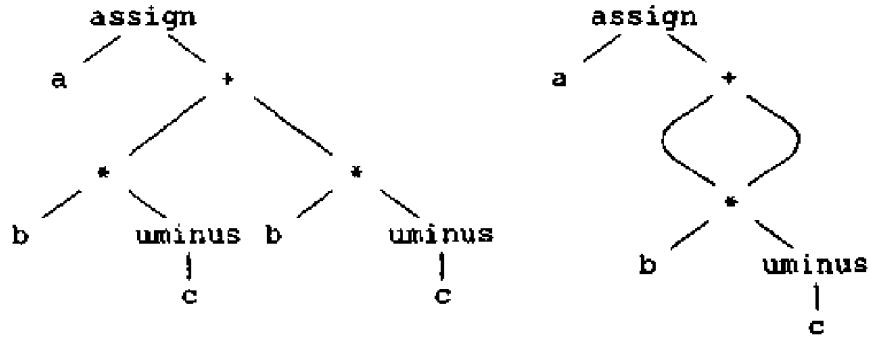
Fig. 8.1. Position of intermediate code generator.

8.1 INTERMEDIATE LANGUAGES

Syntax trees and postfix notation, introduced in Sections 5.2 and 2.3, respectively, are two kinds of intermediate representations. A third, called three-address code, will be used in this chapter. The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

Graphical Representations

A syntax tree depicts the natural hierarchical structure of a source program. A dag gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement `a := b * - c + b * - c` appear in Fig. 8.2.



(a) Syntax tree. (b) Dag.

Fig. 8.2. Graphical representations of `a := b * - c + b * - c`.

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree in Fig. 8.2(a) is

```
a b c uminus * b c uminus * + assign          (8.1)
```

The edges in a syntax tree do not appear explicitly in postfix notation. They can be recovered from the order in which the nodes appear and the number of operands that the operator at a node expects. The recovery of edges is similar to the evaluation, using a stack, of an expression in postfix notation. See Section 2.8 for more details and the relationship between postfix notation and code for a stack machine.

Syntax trees for assignment statements are produced by the syntax-directed definition in Fig. 8.3; it is an extension of one in Section 5.2. Nonterminal S generates an assignment statement. The two binary operators `+` and `*` are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree of Fig. 8.2(a) from the input `a := b * - c + b * - c`.

PRODUCTION	SEMANTIC RULE
$S \rightarrow$ **id** := E	$S.nptr$:= *mknode*('assign', *mkleaf*(**id**, **id**.*place*), $E.nptr$)
$E \rightarrow E_1 + E_2$	$E.nptr$:= *mknode*('+', $E_1.nptr$, $E_2.nptr$)
$E \rightarrow E_1 * E_2$	$E.nptr$:= *mknode*('*', $E_1.nptr$, $E_2.nptr$)
$E \rightarrow - E_1$	$E.nptr$:= *mkunode*('uminus', $E_1.nptr$)
$E \rightarrow (E_1)$	$E.nptr$:= $E_1.nptr$
$E \rightarrow$ **id**	$E.nptr$:= *mkleaf*(**id**, **id**.*place*)

Fig. 8.3. Syntax-directed definition to produce syntax trees for assignment statements.

This same syntax-directed definition will produce the dag in Fig. 8.2(b) if the functions *mkunode*(*op*, *child*) and *mknode*(*op*, *left*, *right*) return a pointer to an existing node whenever possible, instead of constructing new nodes. The token **id** has an attribute *place* that points to the symbol-table entry for the identifier. In Section 8.3, we show how a symbol-table entry can be found from an attribute **id**.*name*, representing the lexeme associated with that occurrence of **id**. If the lexical analyzer holds all lexemes in a single array of characters, then attribute *name* might be the index of the first character of the lexeme.

Two representations of the syntax tree in Fig. 8.2(a) appear in Fig. 8.4. Each node is represented as a record with a field for its operator and additional fields for pointers to its children. In Fig 8.4(b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.

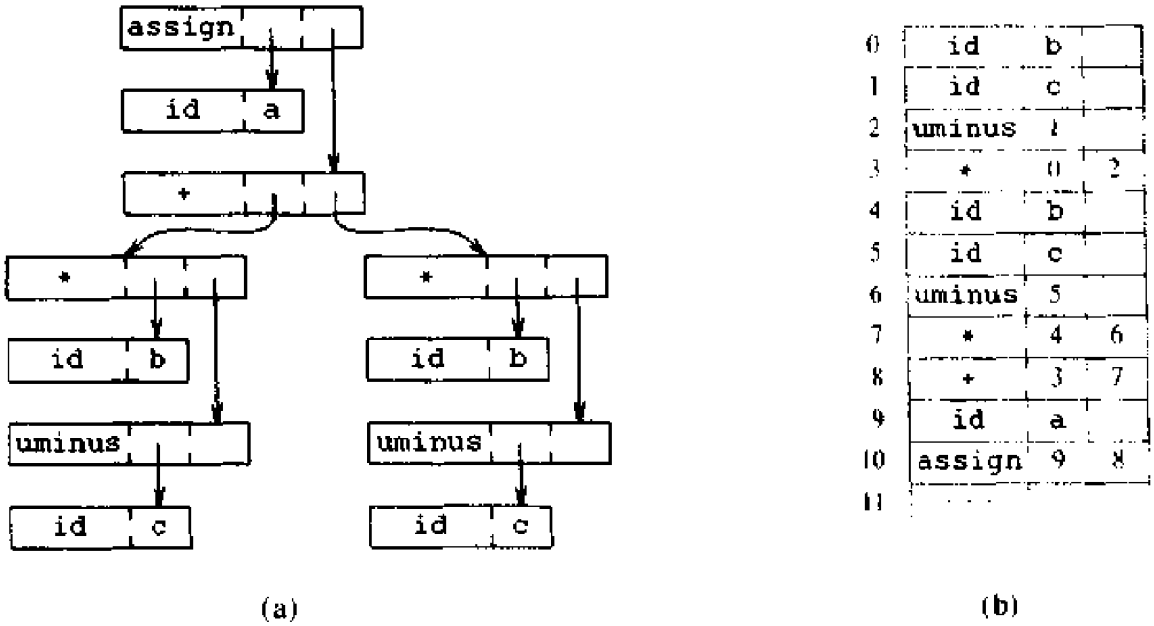


Fig. 8.4. Two representations of the syntax tree in Fig. 8.2(a).

Three-Address Code

Three-address code is a sequence of statements of the general form

```
x := y op z
```

where x, y, and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Note that no built-up arithmetic expressions are permitted, as there is only one operator on the right side of a statement. Thus a source language expression like x + y * z might be translated into a sequence

```
t1 := y * z
t2 := x + t1
```

where t_1 and t_2 are compiler-generated temporary names. This unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization. (See Chapters 10 and 12.) The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged — unlike postfix notation.

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 syntax tree and dag in Fig. 8.2 are represented by the three-address code sequences in Fig. 8.5. Variable names can appear directly in three-address statements, so Fig. 8.5(a) has no statements corresponding to the leaves in Fig. 8.4.

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

(a) Code for the syntax tree.

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

(b) Code for the dag.

Fig. 8.5. Three-address code corresponding to the tree and dag in Fig. 8.2.

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result. In the implementations of three-address code given later in this section, a programmer-defined name is replaced by a pointer to a symbol-table entry for that name.

Types of Three-Address Statements

Three-address statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding intermediate code. Actual indices can be substituted for the labels either by making a separate pass, or by using "backpatching," discussed in Section 8.6.

Here are the common three-address statements used in the remainder of this book:

1. Assignment statements of the form `x := y` *op* `z`, where *op* is a binary arithmetic or logical operation.

2. Assignment instructions of the form `x :=` *op* `y`, where *op* is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.

3. *Copy statements* of the form `x := y` where the value of `y` is assigned to `x`.

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

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

6. `param x` and `call p,` *n* for procedure calls and `return y`, where `y` representing a returned value is optional. Their typical use is as the sequence of three-address statements

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

   generated as part of a call of the procedure $p(x_1, x_2, \ldots, x_n)$. The integer *n* indicating the number of actual-parameters in "`call p,` *n*" is not redundant because calls can be nested. The implementation of procedure calls is outlined in Section 8.7.

7. Indexed assignments of the form `x := y[i]` and `x[i] := y`. The first of these sets `x` to the value in the location `i` memory units beyond location `y`. The statement `x[i] := y` sets the contents of the location `i` units beyond `x` to the value of `y`. In both these instructions, `x`, `y`, and `i` refer to data objects.

8. Address and pointer assignments of the form `x := &y`, `x := *y`, and

`*x := y`. The first of these sets the value of `x` to be the location of `y`. 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. That is, the *r*-value of `x` is the *l*-value (location) of some object. In the statement `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`.

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set must clearly be rich enough to implement the operations in the source language. A small operator set is easier to implement on a new target machine. However, a restricted instruction set may force the front end to generate long sequences of statements for some source language operations. The optimizer and code generator may then have to work harder if good code is to be generated.

Syntax-Directed Translation into Three-Address Code

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. The value of nonterminal E on the left side of $E \rightarrow E_1 + E_2$ will be computed into a new temporary `t`. In general, the three-address code for **id** `:=` E consists of code to evaluate E into some temporary `t`, followed by the assignment **id**.*place* `:= t`. If an expression is a single identifier, say `y`, then `y` itself holds the value of the expression. For the moment, we create a new name every time a temporary is needed; techniques for reusing temporaries are given in Section 8.3.

The S-attributed definition in Fig. 8.6 generates three-address code for assignment statements. Given input `a := b*-c + b*-c`, it produces the code in Fig. 8.5(a). The synthesized attribute *S.code* represents the three-address code for the assignment S. The nonterminal E has two attributes:

1. *E.place*, the name that will hold the value of E, and
2. *E.code*, the sequence of three-address statements evaluating E.

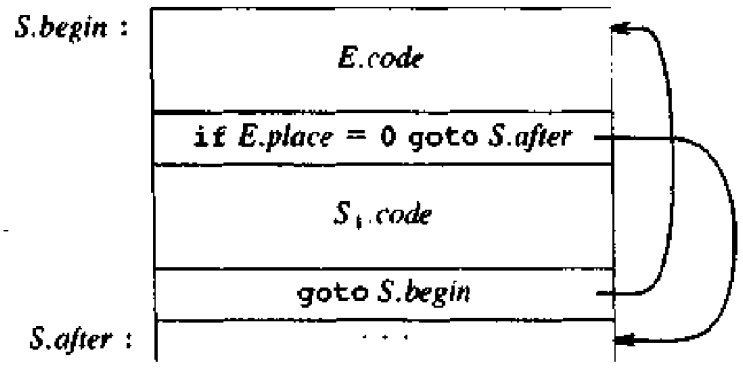
The function *newtemp* returns a sequence of distinct names $t_1, t_2, \ldots$ in response to successive calls.

For convenience, we use the notation *gen*(x ':=' y '+' z) in Fig. 8.6 to represent the three-address statement $x := y + z$. Expressions appearing instead of variables like x, y, and z are evaluated when passed to *gen*, and quoted operators or operands, like '+', are taken literally. In practice, three-address statements might be sent to an output file, rather than built up into the *code* attributes.

Flow-of-control statements can be added to the language of assignments in Fig. 8.6 by productions and semantic rules like the ones for while statements in Fig. 8.7. In the figure, the code for $S \rightarrow$ **while** E **do** S_1 is generated using new attributes *S.begin* and *S.after* to mark the first statement in the code for E

PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **id** := E	$S.code := E.code \parallel gen(\textbf{id}.place\ ':='\ E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place\ ':='\ E_1.place\ '+'\ E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$ $E.code := E_1.code \parallel E_2.code \parallel gen(E.place\ ':='\ E_1.place\ '*'\ E_2.place)$
$E \rightarrow - E_1$	$E.place := newtemp;$ $E.code := E_1.code \parallel gen(E.place\ ':='\ \texttt{'uminus'}\ E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow$ **id**	$E.place :=$ **id**$.place;$ $E.code := ''$

Fig. 8.6. Syntax-directed definition to produce three-address code for assignments.



PRODUCTION	SEMANTIC RULES
$S \rightarrow$ **while** E **do** S_1	$S.begin := newlabel;$ $S.after := newlabel;$ $S.code := gen(S.begin\ ':')\ \parallel$ $E.code \parallel$ $gen(\texttt{'if'}\ E.place\ \texttt{'='}\ \texttt{'0'}\ \texttt{'goto'}\ S.after) \parallel$ $S_1.code \parallel$ $gen(\texttt{'goto'}\ S.begin) \parallel$ $gen(S.after\ ':')$

Fig. 8.7. Semantic rules generating code for a while statement.

and the statement following the code for S, respectively. These attributes represent labels created by a function *newlabel* that returns a new label every time it is called. Note that *S.after* becomes the label of the statement that comes after the code for the while statement. We assume that a non-zero expression represents true; that is, when the value of E becomes zero, control leaves the while statement.

Expressions that govern the flow of control may in general be boolean expressions containing relational and logical operators. The semantic rules for while statements in Section 8.6 differ from those in Fig. 8.7 to allow for flow of control within boolean expressions.

Postfix notation can be obtained by adapting the semantic rules in Fig. 8.6 (or see Fig. 2.5). The postfix notation for an identifier is the identifier itself. The rules for the other productions concatenate only the operator after the code for the operands. For example, associated with the production $E \rightarrow -E_1$ is the semantic rule

$$E.code \;:=\; E_1.code \;\|\; \texttt{'uminus'}$$

In general, the intermediate form produced by the syntax-directed translations in this chapter can be changed by making similar modifications to the semantic rules.

Implementations of Three-Address Statements

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are quadruples, triples, and indirect triples.

Quadruples

A quadruple is a record structure with four fields, which we call *op*, *arg* 1, *arg* 2, and *result*. The *op* field contains an internal code for the operator. The three-address statement `x := y op z` is represented by placing `y` in *arg* 1, `z` in *arg* 2, and `x` in *result*. Statements with unary operators like `x := - y` or `x := y` do not use *arg* 2. Operators like `param` use neither *arg* 2 nor *result*. Conditional and unconditional jumps put the target label in *result*. The quadruples in Fig. 8.8(a) are for the assignment `a := b * - c + b * - c`. They are obtained from the three-address code in Fig. 8.5(a).

The contents of fields *arg* 1, *arg* 2, and *result* are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples

To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it. If we do

	op	*arg* 1	*arg* 2	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Quadruples

	op	*arg* 1	*arg* 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

Fig. 8.8. Quadruple and triple representations of three-address statements.

so, three-address statements can be represented by records with only three fields: *op*, *arg* 1 and *arg* 2, as in Fig. 8.8(b). The fields *arg* 1 and *arg* 2, for the arguments of *op*, are either pointers to the symbol table (for programmer-defined names or constants) or pointers into the triple structure (for temporary values). Since three fields are used, this intermediate code format is known as triples.[1] Except for the treatment of programmer-defined names, triples correspond to the representation of a syntax tree or dag by an array of nodes, as in Fig. 8.4.

Parenthesized numbers represent pointers into the triple structure, while symbol-table pointers are represented by the names themselves. In practice, the information needed to interpret the different kinds of entries in the *arg* 1 and *arg* 2 fields can be encoded into the *op* field or some additional fields. The triples in Fig. 8.8(b) correspond to the quadruples in Fig. 8.8(a). Note that the copy statement `a := t5` is encoded in the triple representation by placing `a` in the *arg* 1 field and using the operator `assign`.

A ternary operation like `x[i] := y` requires two entries in the triple structure, as shown in Fig. 8.9(a), while `x := y[i]` is naturally represented as two operations in Fig. 8.9(b).

	op	*arg* 1	*arg* 2
(0)	[]=	x	i
(1)	assign	(0)	y

(a) `x[i] := y`

	op	*arg* 1	*arg* 2
(0)	=[]	y	i
(1)	assign	x	(0)

(b) `x := y[i]`

Fig. 8.9. More triple representations.

[1] Some refer to triples as "two-address code," preferring to identify "quadruples" with the term "three-address code." We shall, however, treat "three-address code" as an abstract notion with various implementations, triples and quadruples being the principal ones.

Indirect Triples

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples.

For example, let us use an array *statement* to list pointers to triples in the desired order. Then the triples in Fig. 8.8(b) might be represented as in Fig. 8.10.

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg* 1	*arg* 2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Fig. 8.10. Indirect triples representation of three-address statements.

Comparison of Representations: The Use of Indirection

The difference between triples and quadruples may be regarded as a matter of how much indirection is present in the representation. When we ultimately produce target code, each name, temporary or programmer-defined, will be assigned some run-time memory location. This location will be placed in the symbol-table entry for the datum. Using the quadruple notation, a three-address statement defining or using a temporary can immediately access the location for that temporary via the symbol table.

A more important benefit of quadruples appears in an optimizing compiler, where statements are often moved around. Using the quadruple notation, the symbol table interposes an extra degree of indirection between the computation of a value and its use. If we move a statement computing x, the statements using x require no change. However, in the triples notation, moving a statement that defines a temporary value requires us to change all references to that statement in the *arg* 1 and *arg* 2 arrays. This problem makes triples difficult to use in an optimizing compiler.

Indirect triples present no such problem. A statement can be moved by reordering the *statement* list. Since pointers to temporary values refer to the *op-arg* 1-*arg* 2 array(s), which are not changed, none of those pointers need be changed. Thus, indirect triples look very much like quadruples as far as their utility is concerned. The two notations require about the same amount of space and they are equally efficient for reordering of code. As with ordinary triples, allocation of storage to those temporaries needing it must be deferred to the code generation phase. However, indirect triples can save some space

compared with quadruples if the same temporary value is used more than once. The reason is that two or more entries in the *statement* array can point to the same line of the *op-arg* 1-*arg* 2 structure. For example, lines (14) and (16) of Fig. 8.10 could be combined and we could then combine (15) and (17).

8.2 DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

When the front end generates addresses, it may have a target machine in mind. Suppose that addresses of consecutive integers differ by 4 on a byte-addressable machine. The address calculations generated by the front end may therefore include multiplications by 4. The instruction set of the target machine may also favor certain layouts of data objects, and hence their addresses. We ignore alignment of data objects here; Example 7.3 shows how data objects are aligned by two compilers.

Declarations in a Procedure

The syntax of languages such as C, Pascal, and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say *offset*, can keep track of the next available relative address.

In the translation scheme of Fig. 8.11 nonterminal P generates a sequence of declarations of the form **id** : T. Before the first declaration is considered, *offset* is set to 0. As each new name is seen, that name is entered in the symbol table with offset equal to the current value of *offset*, and *offset* is incremented by the width of the data object denoted by that name.

The procedure *enter*(*name*, *type*, *offset*) creates a symbol-table entry for *name*, gives it type *type* and relative address *offset* in its data area. We use synthesized attributes *type* and *width* for nonterminal T to indicate the type and width, or number of memory units taken by objects of that type. Attribute *type* represents a type expression constructed from the basic types *integer* and *real* by applying the type constructors *pointer* and *array*, as in Section 6.1. If type expressions are represented by graphs, then attribute *type* might be a pointer to the node representing a type expression.

In Fig. 8.11, integers have width 4 and reals have width 8. The width of an array is obtained by multiplying the width of each element by the number of elements in the array.[2] The width of each pointer is assumed to be 4. In

[2] For arrays whose lower bound is not 0, the calculation of addresses for array elements is simplified if the offset entered into the symbol table is adjusted as discussed in Section 8.3.