Chapter 6

Intermediate-Code Generation

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. Ideally, details of the source language are confined to the front end, and details of the target machine to the back end. With a suitably defined intermediate representation, a compiler for language i and machine j can then be built by combining the front end for language i with the back end for machine j. This approach to creating suite of compilers can save a considerable amount of effort: $m \times n$ compilers can be built by writing just m front ends and n back ends.

This chapter deals with intermediate representations, static type checking, and intermediate code generation. For simplicity, we assume that a compiler front end is organized as in Fig. 6.1, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. We shall use the syntax-directed formalisms of Chapters 2 and 5 to specify checking and translation. Many of the translation schemes can be implemented during either bottom-up or top-down parsing, using the techniques of Chapter 5. All schemes can be implemented by creating a syntax tree and then walking the tree.

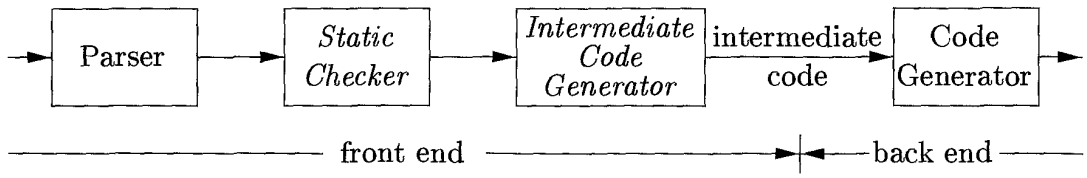


Figure 6.1: Logical structure of a compiler front end

Static checking includes *type checking*, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain

after parsing. For example, static checking assures that a break-statement in C is enclosed within a while-, for-, or switch-statement; an error is reported if such an enclosing statement does not exist.

The approach in this chapter can be used for a wide range of intermediate representations, including syntax trees and three-address code, both of which were introduced in Section 2.8. The term "three-address code" comes from instructions of the general form $x = y\ op\ z$ with three addresses: two for the operands y and z and one for the result x.

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations, as in Fig. 6.2. High-level representations are close to the source language and low-level representations are close to the target machine. Syntax trees are high level; they depict the natural hierarchical structure of the source program and are well suited to tasks like static type checking.

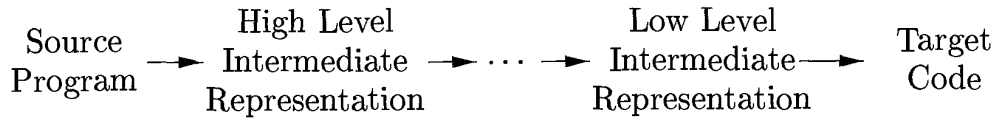


Figure 6.2: A compiler might use a sequence of intermediate representations

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. For expressions, the differences between syntax trees and three-address code are superficial, as we shall see in Section 6.2.3. For looping statements, for example, a syntax tree represents the components of a statement, whereas three-address code contains labels and jump instructions to represent the flow of control, as in machine language.

The choice or design of an intermediate representation varies from compiler to compiler. An intermediate representation may either be an actual language or it may consist of internal data structures that are shared by phases of the compiler. C is a programming language, yet it is often used as an intermediate form because it is flexible, it compiles into efficient machine code, and its compilers are widely available. The original C++ compiler consisted of a front end that generated C, treating a C compiler as a back end.

6.1 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.

6.1.1 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 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 gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Example 6.1: Figure 6.3 shows the DAG for the expression

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

The leaf for `a` has two parents, because `a` appears twice in the expression. More interestingly, the two occurrences of the common subexpression `b-c` are represented by one node, the node labeled $-$. That node has two parents, representing its two uses in the subexpressions `a*(b-c)` and `(b-c)*d`. Even though `b` and `c` appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression `b-c`. □

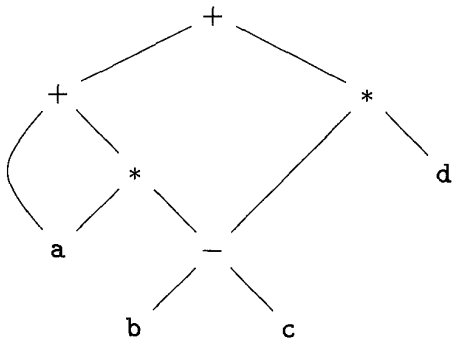


Figure 6.3: Dag for the expression `a + a * (b - c) + (b - c) * d`

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 is returned. For instance, before constructing a new node, *Node*(*op*, *left*, *right*) we check whether there is already a node with label *op*, and children *left* and *right*, in that order. If so, *Node* returns the existing node; otherwise, it creates a new node.

Example 6.2: The sequence of steps shown in Fig. 6.5 constructs the DAG in Fig. 6.3, provided *Node* and *Leaf* return an existing node, if possible, as

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

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

1) $p_1 = Leaf($**id**, *entry-a*$)$
2) $p_2 = Leaf($**id**, *entry-a*$) = p_1$
3) $p_3 = Leaf($**id**, *entry-b*$)$
4) $p_4 = Leaf($**id**, *entry-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) $p_8 = Leaf($**id**, *entry-b*$) = p_3$
9) $p_9 = Leaf($**id**, *entry-c*$) = p_4$
10) $p_{10} = Node('-', p_3, p_4) = p_5$
11) $p_{11} = Leaf($**id**, *entry-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

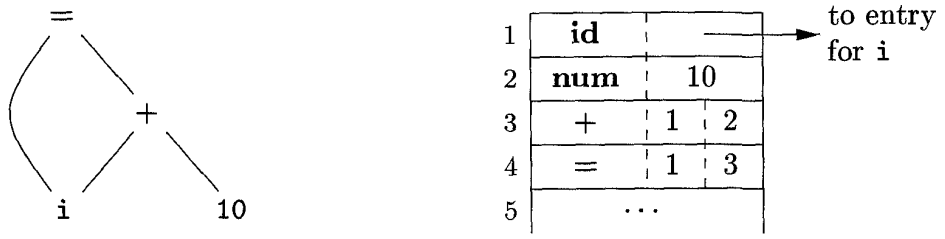
discussed above. We assume that *entry-a* points to the symbol-table entry for a, and similarly for the other identifiers.

When the call to $Leaf($**id**, *entry-a*$)$ is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$. Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., $p_8 = p_3$ and $p_9 = p_4$). Hence the node returned at step 10 must be the same at that returned at step 5; i.e., $p_{10} = p_5$. □

6.1.2 The Value-Number Method for Constructing DAG's

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.



(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 historically has been called the *value number* for the node or for the expression represented by the node. For instance, in Fig. 6.6, the node labeled + has value number 3, and its left and right children have value numbers 1 and 2, respectively. In practice, we could use pointers to records or references to objects instead of integer indexes, but we shall still refer to the reference to a node as its "value number." If stored in an appropriate data structure, value numbers help us construct expression DAG's efficiently; the next algorithm shows how.

Suppose that nodes are stored in an array, as in Fig. 6.6, and each node is referred to by its value number. Let the *signature* of an interior node be the triple $\langle op, l, r \rangle$, where op is the label, l its left child's value number, and r its right child's value number. A unary operator may be assumed to have $r = 0$.

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. □

While Algorithm 6.3 yields the desired output, searching the entire array every time we are asked to locate one node is expensive, especially if the array holds expressions from an entire program. A more efficient approach is to use a hash table, in which the nodes are put into "buckets," each of which typically will have only a few nodes. The hash table is one of several data structures that support *dictionaries* efficiently.[1] A dictionary is an abstract data type that

[1]See Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, for a discussion of data structures supporting dictionaries.

allows us to insert and delete elements of a set, and to determine whether a given element is currently in the set. A good data structure for dictionaries, such as a hash table, performs each of these operations in time that is constant or close to constant, independent of the size of the set.

To construct a hash table for the nodes of a DAG, we need a *hash function* h that computes the index of the bucket for a signature $\langle op, l, r \rangle$, in a way that distributes the signatures across buckets, so that it is unlikely that any one bucket will get much more than a fair share of the nodes. The bucket index $h(op, l, r)$ is computed deterministically from op, l, and r, so that we may repeat the calculation and always get to the same bucket index for node $\langle op, l, r \rangle$.

The buckets can be implemented as linked lists, as in Fig. 6.7. An array, indexed by hash value, holds the *bucket headers*, each of which points to the first cell of a list. Within the linked list for a bucket, each cell holds the value number of one of the nodes that hash to that bucket. That is, node $\langle op, l, r \rangle$ can be found on the list whose header is at index $h(op, l, r)$ of the array.

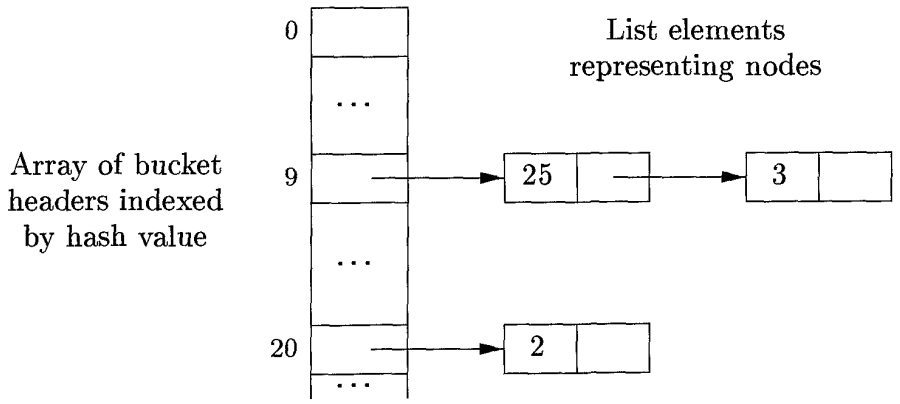


Figure 6.7: Data structure for searching buckets

Thus, given the input node op, l, and r, we compute the bucket index $h(op, l, r)$ and search the list of cells in this bucket for the given input node. Typically, there are enough buckets so that no list has more than a few cells. We may need to look at all the cells within a bucket, however, and for each value number v found in a cell, we must check whether the signature $\langle op, l, r \rangle$ of the input node matches the node with value number v in the list of cells (as in Fig. 6.7). If we find a match, we return v. If we find no match, we know no such node can exist in any other bucket, so we create a new cell, add it to the list of cells for bucket index $h(op, l, r)$, and return the value number in that new cell.

6.1.3 Exercises for Section 6.1

Exercise 6.1.1: Construct the DAG for the expression

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

Exercise 6.1.2: Construct the DAG and identify the value numbers for the subexpressions of the following expressions, assuming + associates from the left.

a) $a + b + (a + b)$.

b) $a + b + a + b$.

c) $a + a + \big((a + a + a + (a + a + a + a)\big)$.

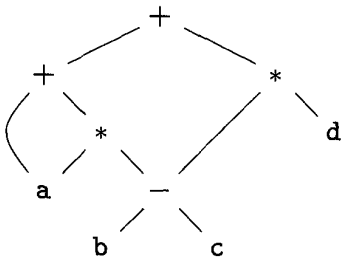
6.2 Three-Address Code

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

$$\begin{aligned} t_1 &= y * z \\ t_2 &= x + t_1 \end{aligned}$$

where t_1 and t_2 are compiler-generated temporary names. This unraveling of multi-operator arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target-code generation and optimization, as discussed in Chapters 8 and 9. The use of names for the intermediate values computed by a program allows three-address code to be rearranged easily.

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. □



(a) DAG

$$\begin{aligned} t_1 &= b - c \\ t_2 &= a * t_1 \\ t_3 &= a + t_2 \\ t_4 &= t_1 * d \\ t_5 &= t_3 + t_4 \end{aligned}$$

(b) Three-address code

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

6.2.1 Addresses and Instructions

Three-address code is built from two concepts: addresses and instructions. In object-oriented terms, these concepts correspond to classes, and the various kinds of addresses and instructions correspond to appropriate subclasses. Alternatively, three-address code can be implemented using records with fields for the addresses; records called quadruples and triples are discussed briefly in Section 6.2.2.

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.

We now consider the common three-address instructions used in the rest of this book. Symbolic labels will be used by instructions that alter the flow of control. A symbolic label represents the index of a three-address instruction in the sequence of instructions. Actual indexes can be substituted for the labels, either by making a separate pass or by "backpatching," discussed in Section 6.7. Here is a list of the common three-address instruction forms:

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, shift operators, 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.

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 $p(x_1, x_2, \ldots, x_n)$. The integer n, indicating the number of actual parameters in "`call` p, n," is not redundant because calls can be nested. That is, some of the first `param` statements could be parameters of a call that comes after p returns its value; that value becomes another parameter of the later call. The implementation of procedure calls is outlined in Section 6.9.

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.

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 uses a symbolic label `L`, attached to the first instruction. The

[2] From Section 2.8.3, l- and r-values are appropriate on the left and right sides of assignments, respectively.

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. □

```
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

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set clearly must be rich enough to implement the operations in the source language. Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine. However, if the front end must generate long sequences of instructions for some source-language operations, then the optimizer and code generator may have to work harder to rediscover the structure and generate good code for these operations.

6.2.2 Quadruples

The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure. In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called "quadruples," "triples," and "indirect triples."

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:

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*.

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 $a = t_5$.

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

```
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

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.

6.2.3 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 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.

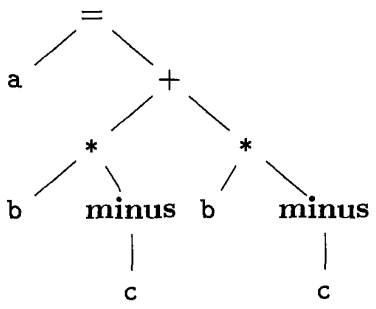
Triples are equivalent to signatures in Algorithm 6.3. Hence, the DAG and triple representations of expressions are equivalent. The equivalence ends with expressions, since syntax-tree variants and three-address code represent control flow quite differently.

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. □

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

Why Do We Need Copy Instructions?

A simple algorithm for translating expressions generates copy instructions for assignments, as in Fig. 6.10(a), where we copy $\texttt{t}_5$ into a rather than assigning $\texttt{t}_2 + \texttt{t}_4$ to a directly. Each subexpression typically gets its own, new temporary to hold its result, and only when the assignment operator = is processed do we learn where to put the value of the complete expression. A code-optimization pass, perhaps using the DAG of Section 6.1.1 as an intermediate form, can discover that $\texttt{t}_5$ can be replaced by a.



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

t = $y[i]$ and x = t, where t is a compiler-generated temporary. Note that the temporary t does not actually appear in a triple, since temporary values are referred to by their position in the triple structure.

A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around. With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change. With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result. This problem does not occur with indirect triples, which we consider next.

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.

With indirect triples, an optimizing compiler can move an instruction by reordering the *instruction* list, without affecting the triples themselves. When implemented in Java, an array of instruction objects is analogous to an indirect triple representation, since Java treats the array elements as references to objects.

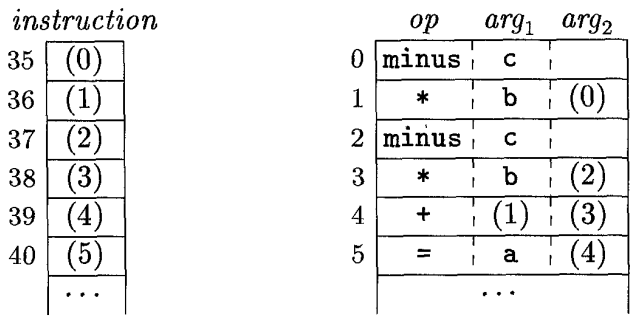


Figure 6.12: Indirect triples representation of three-address code

6.2.4 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.

```
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

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;
```

has two control-flow paths in which the variable x gets defined. If we use different names for x in the true part and the false part of the conditional statement, then which name should we use in the assignment y = x * a? Here is where the second distinctive aspect of SSA comes into play. SSA uses a notational convention called the ϕ-function to combine the two definitions of x:

```
if ( flag ) x1 = -1; else x2 = 1;
x3 = φ(x1, x2);
```

Here, $\phi(x_1, x_2)$ has the value x_1 if the control flow passes through the true part of the conditional and the value x_2 if the control flow passes through the false part. That is to say, the ϕ-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the ϕ-function.

6.2.5 Exercises for Section 6.2

Exercise 6.2.1: Translate the arithmetic expression $a + -(b + c)$ into:

a) A syntax tree.

b) Quadruples.

c) Triples.

d) Indirect triples.

Exercise 6.2.2: Repeat Exercise 6.2.1 for the following assignment statements:

i. `a = b[i] + c[j].`

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

iii. `x = f(y+1) + 2.`

iv. `x = *p + &y.`

Exercise 6.2.3: Show how to transform a three-address code sequence into one in which each defined variable gets a unique variable name.

6.3 Types and Declarations

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

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

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