CHAPTER SIX
6. INTERMEDIATE CODE GENERATION

6.1. Introduction

In a compiler, the front end translates a source program into an intermediate representation, and the back end generates the target code from this intermediate representation. Although a source program can be translated directly into the target language, some benefits of using a machine-independent intermediate form are:

- Retargeting to another machine is facilitated; a complier for a different machine can be created by attaching a back end for the new machine to an existing front end.
- A machine-independent code optimizer can be applied to the intermediate representation
- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.

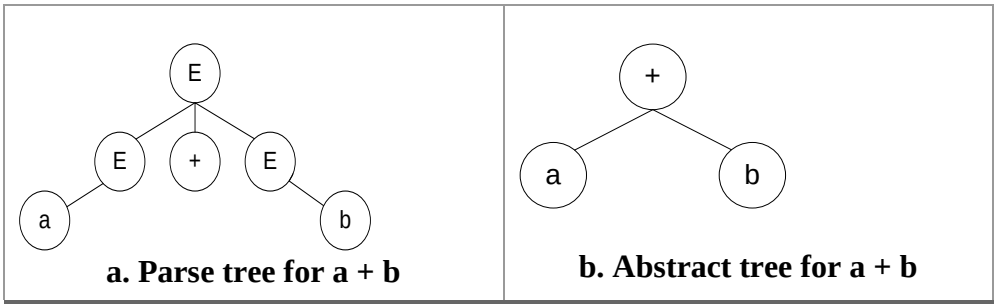
6.2. Intermediate Languages

Intermediate codes can be represented in a variety of ways. The intermediate code generator receives syntax directed translated syntactical constructs and represent them in any one of the following intermediate codes: Syntax trees, postfix notations, and three-address code are intermediate representations.

❖ Syntax Tree

During parsing, syntax-tree nodes are created to represent significant programming constructs. As analysis proceeds, information is added to the nodes in the form of attributes associated with the nodes. The choice of attributes depends on the translation to be performed. It is the natural hierarchical structure of a source program.

A syntax tree (abstract tree) is a condensed form of parse tree useful for representing language constructs. For example, for the string **a + b**, the parse tree in (a) below will be represented by the syntax tree shown in (b); in a syntax tree, operators and keywords do not appear as leaves, but rather are associated with the interior node that would be the parent of those leaves in the parse tree.



a. Parse tree for a + b **b. Abstract tree for a + b**

The syntax trees will have operators as nodes but the parse trees will have the non-terminals as their nodes. A CFG is essential for drawing a parse tree, but for a syntax tree an expression is

essential (input string). This type of intermediate code representation is not also widely used in the compilers.

❖ Postfix Notation

We are familiar with postfix notations. This is also one type of intermediate code representation. It is called reverse polish notation. The postfix notation is practical for an intermediate representation as the operands are found just before the operator. In fact, the postfix notation is a linearized representation of a syntax tree.

- e.g., 1 + 2 * 3 (this is infix expression) will be represented in the postfix notation as 1 2 3 * +.

This type of intermediate code representation is not generally used in the compilers.

❖ Three-Address code

The three-address code is a type of intermediate code, which are popularly used in compilers. Mostly these are used in optimizing compilers.

The three address code is a sequence of statements of the form:

X := Y op Z

where: X, Y, and Z are names, constants or compiler-generated temporaries; and ***op*** is an operator such as integer or floating point arithmetic operator, or a logical operator on boolean-valued data.

Note that:

- ✓ No built-up arithmetic operator is permitted
- ✓ Only one operator at the right side of the assignment is possible, i.e., **x + y + z** is not possible
- ✓ Similarly to postfix notation, the three-address code is a linearized representation of a syntax tree. It has been given the name "three-address code" because each instruction usually contains three-addresses, two for the operands and one for the result.

A source language expression like **x+ y * z** might be translated into a sequence

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

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. The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged - unlike postfix notation.

For the assignment statement **a:= b* -c + b* -c**, the corresponding three-address code is the following.

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

❖ Types of Three-Address Statements

Three-address statements are similar 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.

Here are the common three-address code statements:

1. Assignment statements of the form *x := y op z*, where *op* is a binary arithmetic or logical operator.
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 integer 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 reiop 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 x_1
   param x_2
   …
   param x_n
   call p, n

   generated as part of a call of the procedure p(x_1 , x_2, . . . , x_n). The integer *n* indicating the number of actual-parameters in *"call p , n"* is not redundant because calls can be nested.
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 god code is to be generated.

❖ Implementation of Three-Address Statements

A three address statement is an abstract form of intermediate ale. 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.

1) Quadruples

A quadruple is a record structure with four fields, which we call *op*, *arg1*, *arg2*, and *result*. The *op* field contains an internal code for the operator. Note: *op* is operator, *arg1* is argument1 and *arg2* is argument2. For instance, the three-address instruction $x = y + z$ is represented by placing + in *op*, y in *arg1*, z in *arg2*, and x in *result*. The following are some exceptions to this rule:

- ✓ Instructions with unary operators like $x = - y$ or $x = y$ do not use *arg2*.
- ✓ Operators like *param* use neither *arg2* nor result.
- ✓ Conditional and unconditional jumps put the target label in *result*.

Example: Three-address code for the assignment **a = b * - c + b * - c**; appears in the following. 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.

t_1 = minus c
t_2 = b * t_1
t_3 = minus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

The quadruple representation for this three-address code is the following.

	op	*arg1*	*arg2*	*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

The contents of fields *arg1*, *arg2*, 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.

2) Triples

Triple is also a record structure used to represent the three address codes, which has only three fields: *op, arg1* and *arg2*.

The fields *arg1* and *arg2*, 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).

Note that the *result* field in quadruple representation 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 quadruple representation, a triple representation would refer to position (0). Parenthesized numbers represent pointers into the triple structure itself, while symbol-table pointers are represented by the names themselves. In practice, the information needed to interpret the different kinds of entries in the *arg1* and *arg2* fields can be encoded into the *op* field or some additional fields.

The triples representation corresponding to the above quadruples is the following. Note that the copy statement a = t_5 is encoded in the triple representation by placing *a* in the *arg1* field and using the operator *assign*.

	op	*arg1*	*arg2*
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

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

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

(a) x [i] = y

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

(b) y = x [i]

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

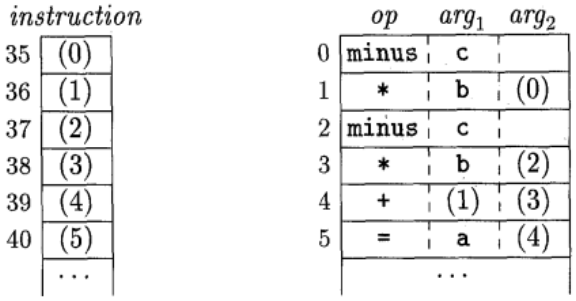
referred to by its position, so moving an instruction may require us to change all references to that result. This problem does not occur with indirect triples, which we consider next.

3) 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. With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the subexpression to produce an optimized code.

For example, let us use an array *instruction* to list pointers to triples in the desired order. Then, the triples for the above example might be represented in indirect triples as follows.



6.3. Declarations

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later should be allocated memory next to the first one.

Example:
We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

```
int a; float b;
Allocation process:
{offset = 0}
int a;
id.type = int
```

```
id.width = 2
offset = offset + id.width
{offset = 2}
float b;
    id.type = float
    id.width = 4
    offset = offset + id.width
{offset = 6}
```

To enter this detail in a symbol table, a procedure enter can be used. This method may have the following structure: *enter(name, type, offset).*

This procedure should create an entry in the symbol table, for variable name, having its type set to *type* and relative address *offset* in its data area.

The declaration is used by the compiler as a source of type-information that it will store in the symbol table. While processing the declaration, the compiler reserves memory area for the variables and stores the relative address of each variable in the symbol table. The relative address consists of an address from the static data area.

The compiler maintains a global *offset* variable that indicates the first address not yet allocated. Initially, offset is assigned 0. Each time an address is allocated to a variable, the offset is incremented by the *width* of the data object denoted by the name.

In the following translation scheme, 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 abject denoted by that name.

The procedure *enter (name, type, offset)* creates a symbol table entry for *name*, give it the type *type* and the relative address *offset* in its data area. We use synthesized attributes *name* and *width* for non-terminal T to indicate the type and width, or number of memory units taken by objects of that type.

Example: Semantic actions for the declaration part. We consider that an integer and a pointer occupy four bytes and a real number occupies 8 bytes of memory.

Production	**Semantic Rules**
P → D	{offset := 0}
D → D; D	
D → **id** :T	{enter(**id**.name, T.*type*, *offset*); *offset* := *offset* + T.*width*}
T → integer	{T.type := integer; T.width := 4}
T → real	{T.*type* := real; T.*width* := 8}
T → array[num] of T1	{T.type := array (num.val, T_1.type); T.width := num.val * T_1.width}
T → ^T1	{T.type := pointer (T_1.type); T.width := 4}

- Note: In languages where nested procedures are possible, we must have several symbol tables, one for each procedure.