COMPILER CONSTRUCTION (Syntax Directed Translation) **Fall 2019**

Lecture 24: Intermediate code generation

Instructor: K.R. Chowdhary *: Professor of CS*

24.1 Introduction

In the analysis-synthesis model of a compiler, the *front end* analyzes a source program and creates an intermediate representation, from this onwards the *back end* works and generates the target code. In ideal sense, details of the source language are confined to the front end, and details of the target machine[1] to the back end. With a properly defined intermediate representation, a compiler for any language i and any machine j can then be built by combining the front end for language i with the back end for machine j. This approach to create suite of compilers can save a considerable amount of effort: $m * n$ compilers can be built by writing just m front ends and n back ends. If $m = n$, then by writing just n front-ends, and n back-ends (total $2n$ code number of codes), we can construct $n \times n = n^2$ number of compilers, or by just 20 number of codes, it is possible to create 100 compilers – a very good saving in efforts for coding.

In the following, we will discuss the intermediate representations, static type checking, and intermediate code generation. For simplicity, we assume that a compiler front end is organized as shown in Fig. 24.1, 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 studied earlier 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 syntax directed translation. All schemes can be implemented by creating a syntax tree and then walking the tree.

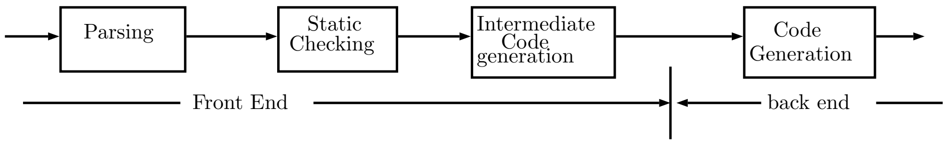

Figure 24.1: Logical structure of a compiler front end.

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-*,

[1]In case of language like Fortran, C and C++, the machine is hardware, and in case of Java it is JVM(java virtual machine – a software)

for-, or *switch-statement*; an error is reported if such statement exists outside the scope of *for, while, switch.*

There can be a wide range of intermediate representations, including syntax trees and three-address code. The term "three-address code" comes from instructions of the 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 shown in Fig. 24.2. 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. It is suited for static checking because the hierarchical structure of a language is fixed in structure.

$$\begin{tikzcd}
\text{Source Program} \arrow[r] & \text{High Level Intermediate Representation} \arrow[r] & \cdots \arrow[r] & \text{Low level Intermediate Representation} \arrow[r] & \text{Target Code}
\end{tikzcd}$$

Figure 24.2: 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. The three-address code (TAC) can range from high-level to low-level, depending on the choice of operators. For expressions, the differences between syntax trees and three-address code are superficial. 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. For example the following sequence of statements are close to the assembly language.

$$\begin{aligned}
label1: \; & x_i = y_i + z_i \\
& i = i + 1 \\
& if \; x < 100 \; goto \; label1 \\
& next \; statement
\end{aligned}$$

The choice or design of an intermediate representation varies from compiler to compiler. An intermediate representation may either be an actual target 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 code, treating a C compiler as a back end.

24.2 Applying Syntax-Directed Translation

The syntax-directed translation (SDT) techniques discussed here will be applied later in type checking and intermediate-code generation. The main application of syntax directed definition (SDD) is in construction of syntax trees. Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree.

To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree. Later, we will discuss approaches to intermediate-code generation that apply an SDD without ever constructing a tree explicitly.

We consider two SDD's for constructing syntax trees for expressions. The first, an S-attributed definition, which is suitable for use during bottom-up parsing. The second, L-attributed, is suitable for use during top-down parsing.

Construction of Syntax Trees. We know that each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression "$E_1 + E_2$" has label + and two children representing the subexpressions E_1 and E_2.

We can implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an *op* (operation code) field that is the label of the node. The objects will have additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function $Leaf(op, val)$ creates a leaf object. Alternatively, if nodes are viewed as records, then $Leaf$ returns a pointer to a new record for a leaf.
- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function $Node$ takes two or more arguments: $Node(op, c_1, c_2, ..., c_k)$ creates an object with first field *op* and k additional fields for the k children $c_1, ..., c_k$.

The table 24.1 sows the syntax directed definition to produce syntax Tree for a specific grammar production.

Table 24.1: Syntax-directed definition to produce syntax trees and DAG's

S.No.	Production	Semantic Rules
1)	$E \to E_1 + T$	$E.node = new\ Node('+', E.node, T.node)$
2)	$E \to E_1 - T$	$E.node = new\ Node('-', E.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to id$	$T.node = new\ Leaf(id, id.entry)$
6)	$T \to num$	$T.node = new\ Leaf(num, num.val)$

24.3 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 will see that DAGs can be constructed by using the same techniques that construct syntax trees.

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 24.1 *Construct DAG for the expression:*

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

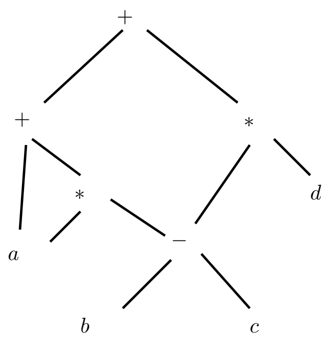

Figure 24.3: Dag for the expression $a + a * (b - c) + (b - c) * d$.

Figure 24.3 shows the DAG for above expression. 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 labelled $-$. 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 $b - c$.

It is left as an exercise for the student to construct the syntax-tree for the same expression, and verify that for two subexpressions $b - c$, two different subtrees are constructed. □

The SDD of Table 24.1 can be used to construct either syntax trees or a DAG's. The functions $Leaf$ and $Node$, create a fresh node each time they are 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, the function *Node* returns the existing node; otherwise, it creates a new node.

Example 24.2 *Constructing DAG of Fig. 24.3 for the expression:* $a+a*(b-c)+(b-c)*d$

The sequence of steps shown in Table 24.2 constructs the DAG of Fig. 24.3, provided *Node* and $Leaf$ return an existing node, if it is already created. We assume that *entry-a* points to the symbol-table entry for a, and similarly for the other identifiers. For creating nodes of DAG we scan the given expression from left to right, and return a pointer p_i, every time pointing either to a node or to a leaf.

Table 24.2: Steps for constructing the DAG for Fig. 24.3

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

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 $p_8 = p_4$). Hence the node returned at step 10 must be the same at that returned at step 5; i.e., $p_{10} = p_5$. □

24.4 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

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

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. The use of names for the intermediate values computed by a program allows three-address code to be rearranged easily. Consider the next example.

Example 24.3 *Construct a three address code for DAG shown in Fig. 24.4 (The Fig. 24.3) is repeated here for convenience).*

Three-address code is a *linearised* representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. The following is the three address for the DAG shown in Fig. 24.4.

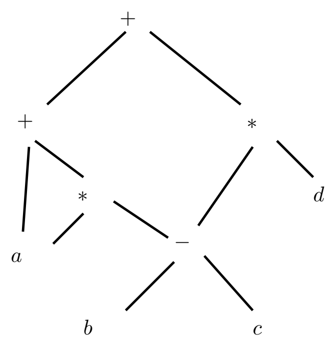

Figure 24.4: DAG for the expression $a + a * (b - c) + (b - c) * d$.

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

24.4.1 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 next section.

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 next section.
- *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 text. 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 later in this text. 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 "$if \ False \ x \ goto \ L$." These instructions execute the instruction with label L next if x is true and when x is false, respectively. Otherwise, the next 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,

   $$\begin{array}{l} param \ x_1 \\ param \ x_2 \\ .... \\ param \ x_n \\ call \ p, n \end{array}$$

   generated as part of a call of the procedure: $p(x_1, x_2, ..., x_n)$. The integer n, which indicates 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 later on.

8. Indexed copy instructions are 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 assignments is of the form $x = \&y$, and pointer assignment is of the form, $x = *y$, and $*x = y$. The instruction $x = \&y$ sets the r-value of x to be the location (l-value) 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. 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 24.4 *Find out the translation of the following statement:*

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

One possible translation of this statement is shown below with symbolic labels. Note that multiplier 4 is used as each memory location of integer is considered as of 4-byte integer data.

$$\begin{aligned} L\ :\ & t_1 = i + 1 \\ & i = t_1 \\ & t_2 = i * 4 \\ & t_3 = a[t_2] \\ & if\ t_3 < v\ goto\ L \end{aligned}$$

The other translation with position number is shown below:

$$\begin{aligned} 100\ \ & t_1 = i + 1 \\ 101\ \ & i = t_1 \\ 102\ \ & t_2 = i * 4 \\ 103\ \ & t_3 = a[t_2] \\ 104\ \ & if\ t_3 < v\ goto\ 100 \end{aligned}$$

The choice of permissible operators is an important issue in the design of an intermediate code. The operator set must be rich enough to implement the all operations in the source language, and efficiently also. Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine. However, if the front-end is required to 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. □

24.5 Quadruples

The three-address instructions discussed above specifies the components of each type of instruction, but it does not specify the representation of these instructions in some 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 *quadruples*, *triples*, and *indirect triples*.

A quadruple (or just "quad") has four fields: *op*, arg_1, arg_2, and *result*. The op field contains an internal code for the operator. For instance, the three-address instruction $x = y + x$ 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 24.5 *Find out the three-address code for the assignment* $a = b * -c + b * -c$;

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 following is three address code.

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

The following is quadruples representation.

Table 24.3: Quadruples.

S.no.	*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
	...	...	...	...

For readability, we use actual identifiers like a, b, and c in the fields arg_1, arg_2 and *result* in Table 24.3, instead of pointers to their symbol-table entries. Temporary names can either be entered into the symbol table like programmer-defined names, or they can be implemented as objects of a class *Temp* with its own methods.

24.6 Triples

A triple has only three fields, which we call *op*, arg_1, and arg_2. Note that the result field in Table 24.3 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 Table 24.3, a triple representation would refer to position (0). Parenthesized numbers represent pointers into the triple structure itself. Earlier, we called positions or pointers to positions as value numbers.

Triples are equivalent to signatures. 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 24.6 *Represent the triples corresponding to three-address code (TAC) and quadruples in example 24.5.*

Corresponding to the three-address code and quadruples in example 24.5, we represent the syntax tree and triples below. The Fig. 24.5 represent the syntax tree and Table 24.4 as Triples. In the triple representation in Table 24.4, 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.

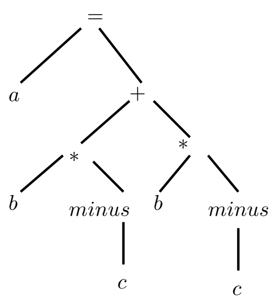

Figure 24.5: Syntax tree for $a + a * (b - c) + (b - c) * d$.

Table 24.4: Triples.

S.no.	*op*	arg_1	arg_2
0	*minus*	c	
1	$*$	b	(0)
2	*minus*	c	
3	$*$	b	(2)
4	$+$	(1)	(3)
5	$=$	a	(4)
	...	...	...

□

24.7 Static Single-Assignment Form

Static single-assignment (SSA) form 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-assignment. The following representation shows a sample intermediate program in *three-address code* and, then in SSA form. Note that subscripts in SSA distinguish each definition of variables p and q in the SSA representation.

Following is three-address code:

$$\begin{aligned}
p &= a + b \\
q &= p - c \\
p &= q * d \\
p &= e - p \\
q &= p + q
\end{aligned}$$

Following is SSA form:

$$\begin{aligned}
p_1 &= a + b \\
q_1 &= p_1 - c \\
p_2 &= q_1 * d \\
p_3 &= e - p_2 \\
q_2 &= p_3 + q_1
\end{aligned}$$

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 4-function to combine the two definitions of x:

```
if (flag)
    $x_1 = -1$;
else
    $x_2 = 1$;
$x_3 = \phi(x_1, x_2)$;
```

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.

Review Questions

1. By writing n front and n back end, how you are able to construct n^2 compiler?
2. Give an example, other than type-check, that is performed through static checking.
3. What are the forms of IRs (Intermediate Representations)? Give examples.
4. Elaborate the following statement: "Parsing, static type-checking, and intermediate code generation, are folded together, in the parsing."
5. What you understand by high-level and low-level intermediate representations? Give examples.
6. Give your own example, how a syntax-tree can be used for static-checking?
7. justify the statement: "Difference between syntax-tree and TAC is superficial."
8. What are the advantages of representing the code in high level intermediate code, like the syntax-tree?
9. Can the C language be used as an intermediate representation? Justify.
10. What are the functions performed by low-level intermediate representation? Explain.
11. The S-attributed SDD is suitable for top-down parsing (True/False).
12. The L-attributed SDD is suitable for bottom-up parsing (True/False)

Exercises

1. Construct DAG for the expression:
$$((x + y) - ((x + y))) + ((x + y) * (x - y))$$
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) $a + a + ((a + a + a + (a + a + a + a))$.
3. For expressions, the differences between syntax trees and three-address code are superficial. Justify this statement.
4. Assume there are M number of total hardware (machines) available, and there are total L number of high level languages.
   (a) How many maximum number of compilers are required so that every language $l \in L$ can run on every machine $m \in M$? Assume that the front-end and back-end of compilers are not separately constructed.
   (b) Let there is concept of front-end and back-end parts of compilers, such that for many front-ends there can be single back-end and vice-versa. How many minimum number of compilers are required in the above case?

   (c) What are the possible advantages and disadvantages of the above configurations?

5. How the hierarchical structure of high level languages is helpful in static type-checking? Give examples.

6. Represent the following expressions using DAG.

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

   (b) $a * a * a$

   (c) $(b + c)^3$

   (d) $(a + b)^2(c + d)^3$

7. "Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine." Justify.

8. Construct Label based and position number based translations for the following sections of codes:

   (a) $for(i = 0; i < 10; i++)j = i^2; k = i^3;$

   (b) $for(i = 0; i < 10; i++)while(i++ < 10);$

   (c) $while(i++ < 10);$

9. Translate the arithmetic expression $a + -(b + c)$ into:

   (a) A syntax tree.

   (b) Quadruples.

   (c) Triples.

   (d) Indirect triples.

10. Repeat above exercise for the following assignment statements:

    (a) $a = b[i] + c[j]$

    (b) $a[i] = b * c - b * d$

    (c) $x = f(y + 1) + 2$

    (d) $x = *p + \&y$

11. What are all the three-address instructions? Explain each in brief.

12. How do you justify the following as three-address instructions?

    (a) $x = op\ y$

    (b) unary minus

    (c) logical relation

    (d) Shift operation

    (e) Conversion of an integer into float

    (f) GOTO L

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, et al., Sep 10, 2006.

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990.

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.

[4] Tools for Large-scale Parser Development, Proceedings of the COLING-2000 Workshop on Efficiency In Large-Scale Parsing Systems, 2000, pp. 54-54, `http://dl.acm.org/citation.cfm?id=2387596.2387604`.