# Unit 4

# Chapter 5

# Intermediate Code Generation

## Structure of the Unit

1. Intermediate Languages
2. Declarations
3. Assignment Statements
4. Boolean Expressions
5. Case Statements
6. Backpatching
7. Procedure Calls

## 5.1 Course Outcomes

After the successful completion of this unit, the student will be able to:

1. Define the concept of intermediate languages and List common intermediate languages used in the compilation process.
2. Explain the scope and lifetime of variables based on declaration rules.
3. Develop a code generator that translates assignment statements into intermediate code.
4. Generate intermediate code for Boolean expressions in various contexts within a program.

## 5.2 Introduction

In the analysis-synthesis model of the compiler, the front-end translates a source program into an intermediate representation. The back-end generates the target-code from the intermediate representation. Benefits of using a machine independent intermediate form are

1. *Retargeting* is facilitated, i.e., a compiler for a different target machine can be designed by attaching a new back-end to the existing front-end.
2. A machine-independent code optimizer can be used to optimize the intermediate representation.

The position of an intermediate code generator is depicted in Fig 5.1.

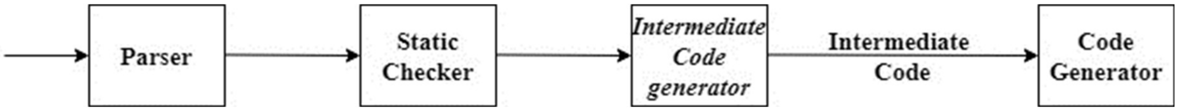

**Fig 5.1** Position of the intermediate code generator

Intermediate code generation is a crucial phase in the compilation process of programming languages, serving as a bridge between the high-level source code and the low-level machine code. In the context of compilers, this phase plays a pivotal role in simplifying the complexities of the source code and facilitating the generation of efficient and target-independent machine code.

During intermediate code generation, the compiler translates the source code into an intermediate representation that is both closer to the source language than machine code and abstract enough to allow for subsequent optimization and code generation stages. This intermediate code serves as an intermediate representation of the program's logic and structure.

The primary objectives of intermediate code generation include facilitating easier analysis of the program, aiding in the detection of errors and optimization opportunities, and providing a foundation for generating efficient target code for different architectures. By creating an intermediate representation, compilers gain a level of abstraction that allows for cleaner separation of concerns between various phases of compilation.

Intermediate code is typically simpler and more uniform than the original source code, making subsequent optimization stages more manageable. It serves as a standardized form that enables the compiler to apply a range of optimization techniques, including peephole optimization, constant folding, and dead code elimination. Additionally, the intermediate code is language-agnostic, enabling the compiler to handle diverse source languages while targeting a specific machine architecture.

In summary, intermediate code generation in the context of compilers is a crucial step that contributes to the overall efficiency, portability, and maintainability of the compiled code. It acts as a pivotal intermediary, allowing the compiler to navigate the complexities of diverse source languages while preparing the program for subsequent phases of optimization and target code generation.

## 5.3 Intermediate Languages

Commonly used intermediate representations are:

1. Syntax Trees
2. Directed Acyclic Graphs(DAG)
3. Three-Adress Codes

The syntax directed definition(semantic rules in section 3.4, table 3.3) to generate directed acyclic graphs are similar to that of syntax trees.

**Graphical Representation:**

Syntax tree represents the natural hierarchical structure of a source program. DAG gives the same information in a compact way. The syntax tree and Dag for the statement $a = b * -c + b * -c$ is depicted in Fig 5.2.

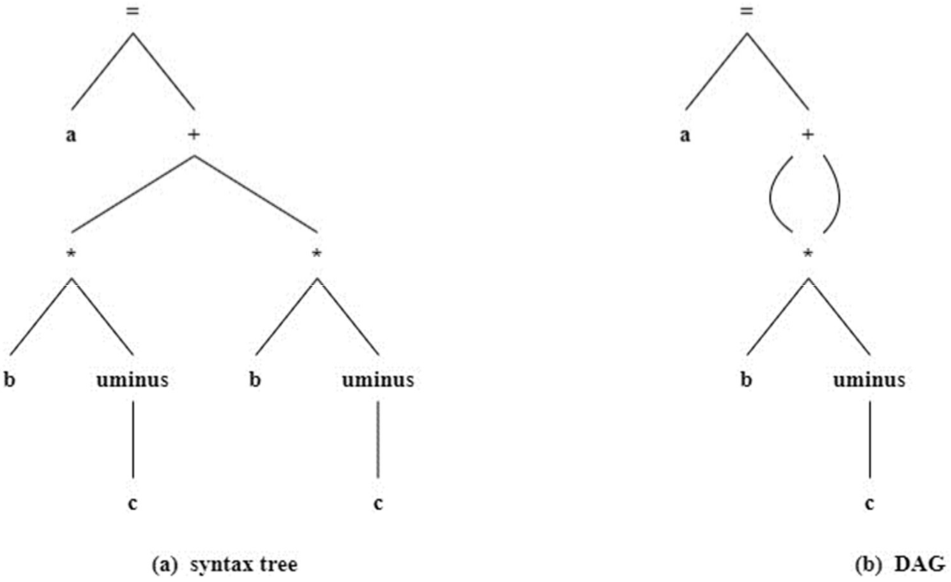

**Fig 5.2** Graphical representation of $a = b * -c + b * -c$

### 5.3.1 Three Address Codes

The three-address code is a linearized representation of the syntax tree. A three-address code consists of a sequence of instructions which are very close to assembly language instructions. Each three-address code has at most

1. Three operands and
2. One operator other than the assignment operator

Each operand may be a *name*, a *constant* or a *compiler generated temporary*.

The statement $a = b + c * 10$ may be translated in to the following sequence

$t_1 = c * 10$

$t_2 = b + t_1$

$a = t_2$

In the above sequence $t_1$ and $t_2$ are compiler generated temporary names.

The syntax tree in Fig 5.2(a) is represented by the following three-address code:

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

$a = t_5$

The DAG in Fig 5.3(a) is represented by the following three-address code:

$t_1 = -c$

$t_2 = b * t_1$

$t_5 = t_1 + t_2$

$a = t_5$

**Types of Three-Address Statements**

1. Assignment statement of the form $a = b\ op\ c$, where $op$ is a binary operator.
2. Statements of the form $a = op\ b$ where $op$ is a unary operator. For example, unary minus, logical $not$ operator etc.
3. Copy statements like $a = b$.
4. The unconditional jump `goto L`. Control is transferred to the statement with the label `L`.
5. Conditional branches like `if` $a\ relop\ b$ `goto L`. $relop$ is a relational operator. If the condition is *true* control is transferred to the instruction with the label `L`. if the condition is *false* the control flows to the next statement.

6. Procedure calls and return statements are implemented using `param x`(parameters), `call p,n`(procedure call) and `return`. For example, the procedure call `p(a, b, c)` is represented using the following instructions

   `param a`

   `param b`

   `param c`

   `call p, 3` (3 is the number of parameters)

7. Assignments of the form `a = b[i]` and `a[i] = b`.

8. Address and pointer assignments of the form `a = &b` and `a = *b`, `*a = *b`.

### 5.3.2 Syntax Directed Translation to generate Three-Address Code

To generate the three-address code temporary names need to be generated for the interior nodes of the syntax tree/DAG. The three-address code for the assignment statement of the form $id = E$ consists of code to evaluate the expression $E$ followed by the assignment $id = temporay$. The value of the nonterminal $E$ in the production $E \rightarrow E_1 + E_2$ is computed and stored in the compiler generated temporary `t.`

The following attributes are used with the non-terminals to generate the three-address code:

1. $S.code$ : This attribute holds the three-address code for the assignment statement $S$.
2. $E.place$ : It is the name that holds the value of $E$.
3. $E.code$ : It holds the sequence of three-address instructions to evaluate the expression $E$.

The following functions are used in the semantic rules:

1. Function *newtemp* : Generates a sequence of distinct names $t_1, t_2, \cdots, t_n$ to hold the intermediate results.
2. Function *gen* : Generates the three-address instruction in the given format.

Table 5.1 shows the syntax directed definition to generate the three-address code for the assignment statements. The operator ‖ represents the string concatenation. Quoted operators like $'+'$ are taken literally.

| **Production** | **Semantic Rule** |
|---|---|
| $S \rightarrow id = E$ | $S.code = E.code \parallel gen(\boldsymbol{id}.place\ '='\ E.place)$ |
| $E \rightarrow E_1 + E_2$ | $E.place = newtemp;$<br>$E.code = E_1.code \parallel E_2.code \parallel gen(E.place\ '='\ E_1.place\ '+'\ E_2.place)$ |
| $E \rightarrow E_1 * E_2$ | $E.place = newtemp;$<br>$E.code = E_1.code \parallel E_2.code \parallel gen(E.place\ '='\ E_1.place\ '*'\ E_2.place)$ |
| $E \rightarrow -E_1$ | $E.place = newtemp;$<br>$E.code = E_1.code \parallel gen(E.place\ '='\ 'uminus'\ E_1.place\ )$ |
| $E \rightarrow (E_1)$ | $E.place = E_1.place;$<br>$E.code = E_1.code$ |
| $E \rightarrow \boldsymbol{id}$ | $E.place = \boldsymbol{id}.place;$<br>$E.code = '\ '$ |

**Table 5.1** SDD to generate three-address code for the assignment statement

The skeleton for the execution of while loop is shown in Fig 5.3.

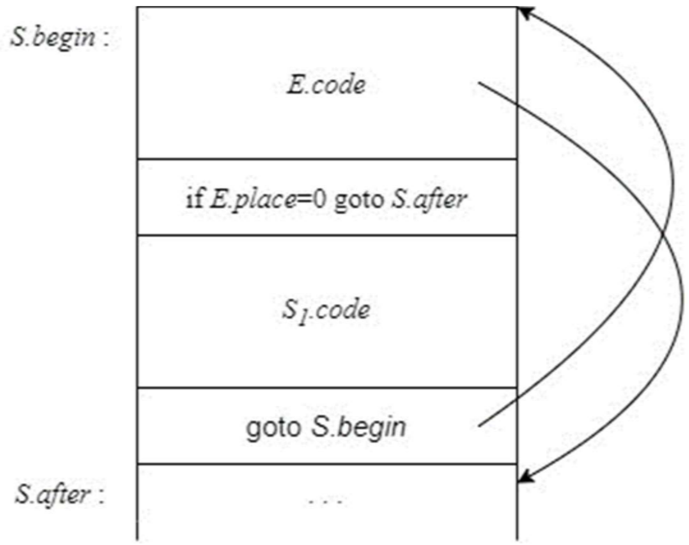

**Fig 5.3** while loop execution

SDD to generate three-address code for while loop is shown in Table 5.2.

| Production | Semantic Rule |
|---|---|
| $S \rightarrow \boldsymbol{while}\ E\ \boldsymbol{do}\ S_1$ | $S.begin = newlabel;$<br>$S.after = newlabel;$<br>$S.code = gen(s.begin':') \parallel E.code$<br>$\parallel gen('if'\ E.place' ='' 0'\ 'goto'S.after) \parallel S_1.code$<br>$\parallel gen('goto'S.begin) \parallel gen(S.after':')$ |

**Table 5.2** SDD to generate three-address code for while loop

The attributes used in the SDD are

- $S.begin$ : Label that marks the first statement in the code for $E$.
- $S.after$ : Label that marks the first statement following the code for $S$.
- $S.code$ : Three-address code for the statement $S(while\ loop)$.

The function *newlabel* returns a new label every time it is called.

### 5.3.3 Implementation of Three Address Statements

Three-address statements can be implemented as records. The record has the fields for the operator and operands. Following are the representations for the three-address statements:

1. Quadruples
2. Triples
3. Indirect Triples

**Quadruples:**

Each three-address statement is represented by a record containing four fields, $op, arg1, arg2, result$. The $op$ field contains the internal representation of the operator. $arg1$ and $arg2$ are the two operands. The field $result$ holds the result of the operation.

Table 5.3 is the quadruple representation for the below three-address code:

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

| | *op* | *arg1* | *arg2* | *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) | $=$ | $t_5$ | | $a$ |

**Table 5.3** quadruple representation for $a = b * -c + b * -c$

**Triples:**

Entering the temporary names in the symbol table may be avoided by referring to the temporary by a position. This position is the position of the statement that computes it. Hence three-address statements can be represented by records consisting of only three fields. The first field is the operator field $op$, the second and third fields are for the operands $arg1, arg2$. The fields $arg1, arg2$ may be operands or pointers to the symbol table(programmer defined names or constants) or pointers into the triple structures(for compiler generated temporaries).

Table 5.4 is the triple representation for the below code

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

| | *op* | *arg1* | *arg2* |
|---|---|---|---|
| (0) | *uminus* | *c* | |
| (1) | $*$ | *b* | (0) |
| (2) | *uminus* | *c* | |
| (3) | $*$ | *b* | (2) |
| (4) | $+$ | (1) | (3) |
| (5) | $=$ | *a* | (4) |

**Table 5.4** Triple representation for $a = b * -c + b * -c$

Parenthesized numbers represent pointers to the entries in the triple structure.

**Indirect Triples:**

In this representation instead of listing the triples themselves, a listing of pointers to triples is used. These pointers are stored in an array. The triple representation in table 5.4 may be represented by indirect triples as shown in tables 5.5(a) and (b).

| | *Statement* |
|---|---|
| (0) | (20) |
| (1) | (21) |
| (2) | (22) |
| (3) | (23) |
| (4) | (24) |
| (5) | (25) |

| | *op* | *arg1* | *arg2* |
|---|---|---|---|
| (20) | *uminus* | *c* | |
| (21) | $*$ | *b* | (0) |
| (22) | *uminus* | *c* | |
| (23) | $*$ | *b* | (2) |
| (24) | $+$ | (1) | (3) |
| (25) | $=$ | *a* | (4) |

**Table 5.5(a)** List of pointers **Table 5.5(b)** List of triples

The indirect triples facilitate the movement of three address instructions during code optimization.

# 5.4 Declarations

As the compiler processes the sequence of declarations in a procedure or block, it has to lay out storage for names that are local to the procedure or block. A symbol table entry is created for each name local to the procedure or block. This entry consists of the *symbol name*, *type* and the *relative address* of the storage for the name.

## 5.4.1 Declarations in a Procedure

Languages such as C, PASCAL and FORTRAN allow all the declarations in a block to be processed as a group. In such cases the compiler maintains a global variable *offset*, to keep track of the next available relative address. The *offset* is set to 0 initially.
Each time a new name is seen the compiler

1. Enters the name in the symbol table with the current value of the *offset* and
2. Increments the *offset* by the width of the data object in the declaration.

The following attributes and procedures are used in the SDD to process the declaration statements.

1. $T.type$: Indicates the type of the names.
2. $T.width$ : Indicates the width or number of memory units(normally bytes) taken by the object of that type.
3. *Procedure enter(name, type, offset)* : Creates a symbol table entry for *name*, gives it the type *type* and the relative address *offset*.
4. *Array(num, type)*: It is a constructor to create the *array* type, with *num* as the size of the array and *type* as the type of the array.

We assume integers have width 4 and reals have width 8. The width of an array is obtained by multiplying the width each element by the number of elements in the array.

Table 5.6 illustrates the translation scheme used to generate relative addresses for the declared names.

| |
|---|
| $P \rightarrow \{offset = 0\}\ D$ |
| $D \rightarrow D; D$ |
| $D \rightarrow \boldsymbol{id} : T\ \{\ enter(\boldsymbol{id}.name, T.type, offset);$<br>$offset = offset + T.width\}$ |
| $T \rightarrow \boldsymbol{integer}\ \{T.type = integer; T.width = 4\}$ |
| $T \rightarrow \boldsymbol{real}\ \{T.type = real; T.width = 8\}$ |
| $T \rightarrow \boldsymbol{array}\ [\ num\ ]\boldsymbol{of}\ T_1\ \{T.type = array(\boldsymbol{num}.val, T_1.type);$<br>$T.width = \boldsymbol{num}.val \times T_1.width\}$ |

**Table 5.6** SDT for variable declarations

## 5.5 Assignment Statements

In this section we will discuss the process of translating assignment statements into three-address code. Additionally, we will explore methods for referencing names in the symbol table and accessing array elements. The translation scheme employed for this purpose involves utilizing the following attributes and methods.

1. $\boldsymbol{id}.name$ :This attribute stores the lexeme for the name represented by **id**.
2. $procedure\ lookup(\boldsymbol{id}$.name) :Checks if there is an entry for this occurrence of $\boldsymbol{id}.name$ in the symbol table.
3. $procedure\ emit$ : Writes three-address code to an output file.

The translation scheme is shown in table 5.7. The attributes $E.code$, $E.place$ have same meaning as in table 5.1.

```
$S \rightarrow \boldsymbol{id} = E; \{p = lookup(\boldsymbol{id}.name);$
                $\boldsymbol{if}\ p \neq null\ \boldsymbol{then}$
                    $emit(p' =' E.place)$
                $\boldsymbol{else}$
                    $error$
```

| |
|---|
| $E \rightarrow E_1 + E_2$ $\{ E.place = newtemp;$<br>$emit(E.place\ '='\ E_1.place'+'E_2.place)$ |
| $E \rightarrow E_1 * E_2$ $\{ E.place = newtemp;$<br>$emit(E.place\ '='\ E_1.place'*'\ E_2.place)$ |
| $E \rightarrow -E_1$ $\{ E.place = newtemp;$<br>$emit(E.place\ '='\ 'uminus'E_1.place)$ |
| $E \rightarrow ( E_1 )$ $\{E.place = E_1.place \}$ |
| $E \rightarrow \boldsymbol{id}$ $\{ p = lookup(\boldsymbol{id}.name);$<br>$\boldsymbol{if}\ p \neq null\ \boldsymbol{then}$<br>$E.place = p$<br>$\boldsymbol{else}$<br>$error \}$ |

**Table 5.6** Translation scheme to generate three-address code for the assignments

**Reusing Temporary Names:**

The method $newtemp$ used in the earlier SDD/SDTs generates a new temporary every time it is invoked. It is useful in optimizing compilers. However, this messes up the symbol table as the number of temporaries generated is huge. Temporaries can be reused by changing the $newtemp$ method.

**Addressing array elements:**

Array elements can be accessed quickly if they are stored in contiguous memory locations. If the width of an element in an array is $w$ then,

The address of the $i^{th}$ element in a single dimensional array is given by:

$base + (i - low) \times w$, where

- $base$ is the start address of the array,
- $low$ is the index of the first element in the array(In C language $low$ is 0).

The address of the element $A[i][j]$ in a two-dimensional array is

$base + ((i - low_1) \times n_2 + j - low_2) \times w$, (array represented in the row-major order) where

- $base$ is the start address of the array,
- $low_1$ is the index of the first element in the first dimension i.e., row(In C language it is 0).
- $low_2$ is the index of the first element in the second dimension i.e., column(In C language it is 0).
- $n_2$ is the number of elements in the second dimension, i.e., number of columns.

The elements in a two-dimensional array can be stored either in the row major order or column major order. Fig 5.4 depicts these layouts for a two-dimensional array $A[2][3]$, i.e., an array with 2 rows and 2 columns.

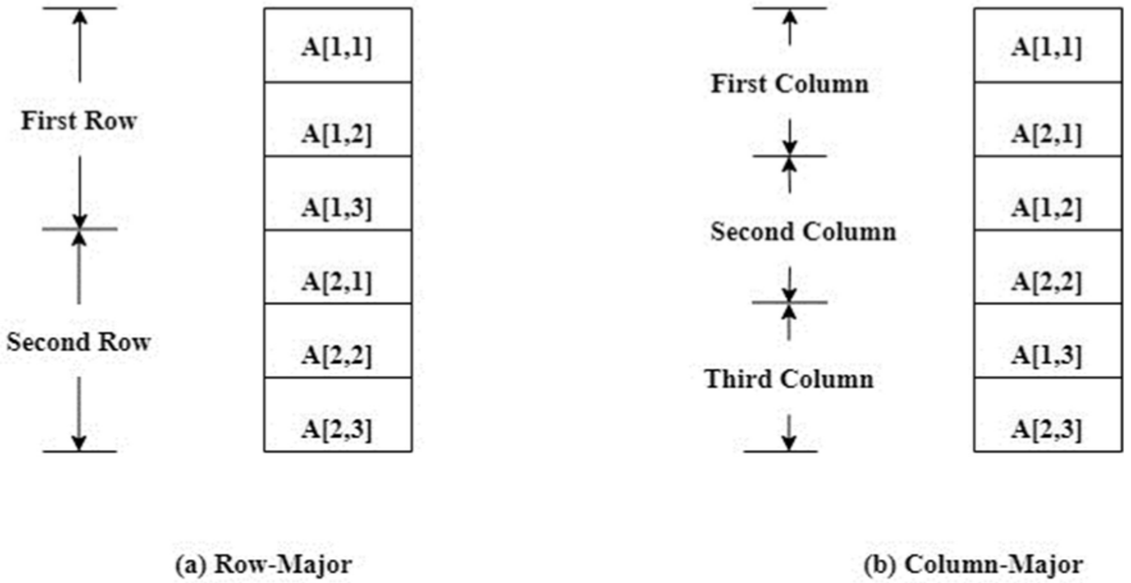

**Fig 5.4** Lay-outs of a two-dimensional array

If an array element is to be accessed by a statement, its address needs to be computed. Hence the compiler needs to generate the code to compute the address. The discussions in this section guide us to design the translation scheme to generate the three-address code, that computes the address of an array element.

## 5.6 Boolean Expressions

Statements that alter the flow-of-control or looping statements use conditional expressions. The purpose of Boolean expressions in a programming language is to compute the logical values, which are often used in conditional expressions.

Boolean expressions use the operators(Boolean) $\boldsymbol{and}, \boldsymbol{or}$, and $\boldsymbol{not}$. These operators are applied to Boolean variables or relational expressions. The relational expressions are of the form $E_1\ \boldsymbol{relop}\ E_2$. $E_1$ and $E_2$ are arithmetic expressions. $\boldsymbol{relop}$ may be $<, \leq, >, \geq, =, \neq$. In this discussion we consider the Boolean expressions represented by the below grammar.

$$E \rightarrow E\ \boldsymbol{or}\ E\ |E\ \boldsymbol{and}\ E|\boldsymbol{not}\ E|\ (E)|\ E\ \boldsymbol{relop}\ E\ |\ true\ |\ false$$

The operator $\boldsymbol{not}$ has the highest precedence, followed by the operator $\boldsymbol{and}$, whereas $\boldsymbol{or}$ has the least priority. The operator $\boldsymbol{not}$ is evaluated from right to left. The operators $\boldsymbol{and}$ and $\boldsymbol{or}$ are evaluated from left to right.

**Methods for Translating the Boolean Expressions**

There are two methods of representing a value of a Boolean expression.

The ***first method*** encodes $true$ and $false$ numerically.

- Often 1 denotes $true$ and 0 denotes $false$.
- In other encodings any $non - zero\ value$ denotes $true$ and $zero$ denotes $false$(as in the programming language C).
- We may let $non - negative$ quantity denote $true$ and $negative$ quantity represent $false$.

The ***second method*** implements Boolean expressions by flow control.

- The value of a Boolean expression is represented by a position reached in a program. This method is useful in implementing flow-of-control statements or loops. For example, in the expression $E_1$ $\boldsymbol{or}$ $E_2$ if $E_1$ is $true$ then $E_2$ we conclude the expression $E_1$ $\boldsymbol{or}$ $E_2$ is $true$ without evaluating $E_2$. This helps the compiler to optimize the code.

Numerical Representation

Consider implementation of Boolean expressions using 1 to denote $true$ and 0 to denote $false$. Expressions will be evaluated completely like arithmetic expressions. Consider the expression

$$a\ or\ b\ and\ not\ c$$

The translation for the above expression is the three-address code

$$t_1 = not\ c$$

$$t_2 = b\ and\ t_1$$

$$t_3 = a\ or\ t_2$$

A relational expression $x > y$ is equivalent to the logical statement if $x > y$ then 1 else 0, which is translated to the following three-address code:

```
100: if x > y goto 103
101: t = 0
102: goto 104
103: t = 1
104:
```

The translation scheme to produce the three-address code uses a variable $nextstat$ that gives the index of the next three-address statement in the output sequence. Table 5.7 shows the translation scheme.

| Production | Semantic Rule |
|---|---|
| $E \rightarrow E_1 \; or \; E_2$ | $\{ E.place = newtemp;$<br>$emit(E.place' =' E_1.place'or'E_2.place)\}$ |
| $E \rightarrow E_1 \; and \; E_2$ | $\{ E.place = newtemp;$<br>$emit(E.place' =' E_1.place'and'E_2.place)\}$ |
| $E \rightarrow not \; E_1$ | $\{ E.place = newtemp;$<br>$emit(E.place' =' \; 'not'E_1.place)\}$ |
| $E \rightarrow ( \; E_1 \; )$ | $\{ E.place = E_1.place\}$ |
| $E \rightarrow \boldsymbol{id_1} \; \boldsymbol{relop} \; \boldsymbol{id_2}$ | $\{ E.place = newtemp;$<br>$emit('if' \; \boldsymbol{id_1}.place \; \boldsymbol{relop}.op \; \boldsymbol{id}_2.place \; 'goto' \; nextstat + 3);$<br>$emit(E.place' =' \; 0;);$<br>$emit('\boldsymbol{goto}' \; nextstat + 2);$<br>$emit(E.place \; ' ='' \; 1')\}$ |
| $E \rightarrow \boldsymbol{true}$ | $\{ E.place = newtemp;$<br>$emit(E.place' ='' \; 1')\}$ |
| $E \rightarrow \boldsymbol{false}$ | $\{ E.place = newtemp;$<br>$emit(E.place' ='' \; 0')\}$ |

**Table 5.7** Translation scheme for Boolean expressions using numerical representation

**Example 1:** The three-address code generated using the translation scheme in table 5.7 for the expression $a < b \; or \; c < d \; or \; e < f$ is shown below.

100: if $a < b$ goto 103

101: $t_1 = 0$

102: goto 104

103: $t_1 = 1$

104: if $c < d$ goto 107

105: $t_2 = 0$

106: goto 108

107: $t_2 = 1$

108: if $e < f$ goto 111

109: $t_3 = 0$

110: goto 112

111: $t_3 = 1$

112: $t_4 = t_2 \; and \; t_3$

113: $t_5 = t_1 \; or \; t_4$

### Short Circuit Code

A Boolean expression may be translated to three-address code without generating code for any of the Boolean operators and without having the code to evaluate the entire expression. This type of evaluation is sometimes called “short-circuit” or “jumping” code.

### Flow-of-Control Statements:

In this section we shall discuss the translation of Boolean expressions inti three-address code in the context of if-then, if-then-else and while-do statements. The following grammar generates these statements:

$S \rightarrow \boldsymbol{if}\ E\ \boldsymbol{then}\ S_1$

$\quad |\ \boldsymbol{if}\ E\ \boldsymbol{then}\ S_1\ \boldsymbol{else}\ S_2$

$\quad |\ \boldsymbol{while}\ E\ \boldsymbol{do}\ S_1$

In the above grammar $E$ is the Boolean expression. Each three-address statement is labeled as in the above example.

- The function *newlabel* generates a new label each time it is called.
- The attributes $E.true$ and $E.false$ are the labels. $E.true$ is the label to which control is transferred when the condition $E$ is $true$. $E.false$ is the label to which control is transferred when the condition $E$ is $false$.
- The attribute $S.code$ is the three-address code for the statement $S$.
- The attribute $S.next$ is the label attached to the first three-address instruction to be executed after the statement $S$.
- $gen()$ is a method that generates the three-address code.

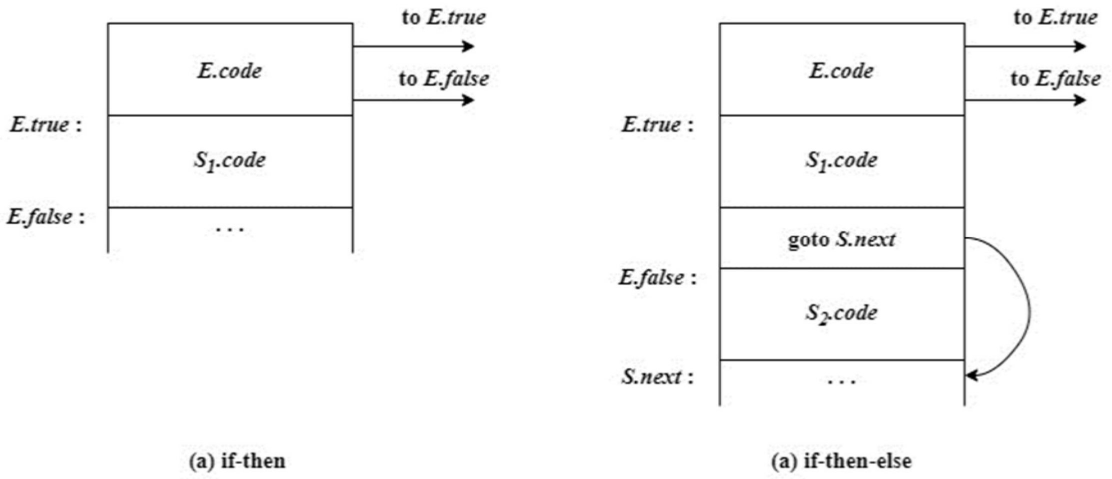

**Fig 5.5** Schematic representation of *if-then*, *if-then-else* statements

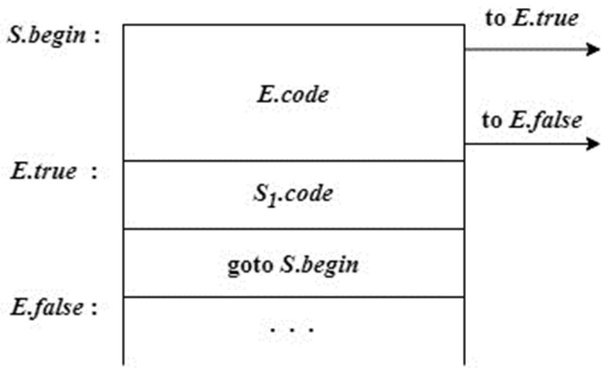

**Fig 5.5** Schematic representation of *while-do* statement

| Production | Semantic Rule |
|---|---|
| $E \rightarrow E_1 \ \boldsymbol{or} \ E_2$ | $E_1.true = E.true;$<br>$E_1.false = newlabel;$<br>$E_2.true = E.true;$<br>$E_2.false = E.false$<br>$E.code = E_1.code \parallel gen(E_1.false':') \parallel E_2.code$ |
| $E \rightarrow E_1 \ \boldsymbol{and} \ E_2$ | $E_1.true = newlabel;$<br>$E_1.false = E.false;$<br>$E_2.true = E.true;$<br>$E_2.false = E.false$<br>$E.code = E_1.code \parallel gen(E_1.true':') \parallel E_2.code$ |
| $E \rightarrow \boldsymbol{not} \ E_1$ | $E_1.true = E.false;$<br>$E_1.false = E.true;$<br>$E.code = E_1.code$ |
| $E \rightarrow ( E_1 )$ | $E_1.true = E.true;$<br>$E_1.false = E.false;$<br>$E.code = E_1.code$ |
| $E \rightarrow \boldsymbol{id_1} \ \boldsymbol{relop} \ \boldsymbol{id_2}$ | $E.code = gen('if' \ \boldsymbol{id_1}.place \ \boldsymbol{relop}.op \ \boldsymbol{id_2}.place \ 'goto' \ E.true) \parallel gen('goto' E.false)$ |
| $E \rightarrow \boldsymbol{true}$ | $E.code = gen('goto' E.true)$ |
| $E \rightarrow \boldsymbol{false}$ | $E.code = gen('goto' E.false)$ |

**Table 5.8** SDD to produce short circuit code for Booleans

**Example 2:** Consider the logical expression $a < b \; or \; c < d \; and \; e < f$ denoted by $L$. The short circuit code for the Boolean expression is shown below.

```
        if $a < b$ goto Ltrue
        goto L1
L1:     if $c < d$ goto L2
        goto Lfalse
L2:     if $e < f$ goto Ltrue
        goto Lfalse
```

- $Ltrue$: serves as the label for the statement that will be executed when the logical expression L evaluates to *true*.
- $Lfalse$: serves as the label for the statement that will be executed when the logical expression L evaluates to *false*.

**Example 3:** Consider the statement

```
while $a < b$ do
    if $c < d$ then
      $x = y + z$
    else
      $x = y - z$
```

The syntax directed definition in table 5.8 with the translation scheme for the assignment statements would produce the following code.

```
L1:     if $a < b$ goto L2
        goto Lnext
L2:     if $c < d$ goto L3
        goto L4
L3:     $t_1 = y + z$
        $x = t_1$
        goto L1
L4:     $t_2 = y - z$
        $x = t_2$
        goto L1
Lnext:
```

$Lnext$: serves as the label for the statement that will be executed when the while loop terminates, i.e., the condition(logical expression) L fails.

**Mixed Mode Boolean Expressions**

So far, we have discussed Boolean expressions containing only relational and logical operators. Typically, Boolean expressions contain arithmetic subexpressions. For example, $(a + b) < c)$ $and$ $(b - c) \neq 10)$. The short circuit or jumping code may be used to represent mixed mode Boolean expressions. For example, the below grammar indicates how mixed mode Boolean expressions might be represented.

$$E \rightarrow E + E \mid E\ \boldsymbol{and}\ E \mid E\ \boldsymbol{relop}\ E \mid \boldsymbol{id}$$

The production $E \rightarrow E + E$ produces a numeric result, whereas the productions $E \rightarrow E\ \boldsymbol{and}\ E$, $E \rightarrow E\ \boldsymbol{relop}\ E$ produce Boolean values. The expression $E\ \boldsymbol{and}\ E$ requires both operands to be Boolean whereas the expression $E\ \boldsymbol{relop}\ E$ take either type of operands. To generate code for this type of expressions, we may use a synthesized attribute $E.type$. The attribute will be either numeric or Boolean depending on the type of $E$. If $E$ is Boolean, then it will have the attributes $E.true$, $E.false$. If $E$ is numeric then it is associated with the attribute $E.place$. Part of the semantic rule for $E \rightarrow E_1 + E_2$ is shown below.

```
$E.type = arith;$
if $E_1.type = arith$ and $E_2.type = arith$ then {
    /* normal addition */
    $E.place = newtemp;$
    $E.code = E_1.code \parallel E_2.code \parallel gen(E.place\ '='\ E_1.place\ '+'\ E_2.place)$
}
else if $E_1.type = arith$ and $E_2.type = bool$ then {
    $E.place = newtemp;$
    $E_2.true = newlabel;$
    $E_2.false = newlabel;$
    $E.code = E_1.code \parallel E_2.code \parallel gen(E_2.true\ ':'\ E.place\ '='\ E_1.place + 1) \parallel$
                                  $gen('goto'\ nextstat + 1) \parallel$
                                  $gen(E_2.false\ ':'\ E.place\ '='\ E_1.place)$
    }
    else if ⋯
```

## 5.7 Case Statements

A “switch” or “case” statement is available in many programming languages. The syntax is shown below:

```
switch (expression) {
        case value: statement;
        case value: statement;
        case value: statement;
                …
        case value: statement;
}
```

The $expression$ is called a selector expression. The selector expression is evaluated first. The $result$ is compared with the constant values and an appropriate statement is executed. The translation of the switch statement must generate the code to:

1. Evaluate the $expression$.
2. Find the value in the list of cases that matches the $result$. The $default$ value matches the $result$ if none of the values mentioned in the cases match($n-way$ branch).
3. Execute the statement associated with the matched value.

**Syntax Directed Translation of Switch(Case) Statements**

Consider the statement

```
switch $E$
{
        case $V_1$: $S_1$
        case $V_2$: $S_2$
                …
        case $V_{n-1}$: $S_{n-1}$
        default: $S_n$


}
```

One translation of the case statement is shown below:

```
          code to evaluate $E$ into $t$
          goto test
$L_1$:    code for $S_1$
          goto next
$L_2$:    code for $S_2$
          goto next
          …
$L_{n-1}$:  code for $S_{n-1}$
          goto next
$L_n$:    code for $S_n$
          goto next
test:     if $t = V_1$ goto $L_1$
          if $t = V_2$ goto $L_2$
          …
          if $t = V_{n-1}$ goto $L_{n-1}$
          goto $L_n$
next:
```

In the above code tests appear at the end, so that the code generator can generate efficient code. Another translation, that generates more straightforward sequence is show below.

```
          code to evaluate $E$ into $t$
          if $t \neq V_1$ goto $L_1$
          code for $S_1$
          goto next
$L_1$:    if $t \neq V_2$ goto $L_2$
          code for $S_1$
          goto next
$L_2$:    …
$L_{n-2}$:  if $t \neq V_{n-1}$ goto $L_{n-1}$
          code for $S_1$
          goto next
$L_{n-1}$:  code for $S_n$
next:
```

The above code is straightforward, but the compiler has to extensive analysis to optimize it.

## 5.8 Backpatching

The most straightforward method for creating syntax-directed definitions involves a two-pass approach. During the initial pass, construct the syntax tree based on the input, and in the second pass, traverse the tree in a depth-first order to calculate the translations. However, a challenge arises with flow-of-control statements and Boolean expressions, as the compiler may not have knowledge of all the labels in jump (goto) statements during a single pass.

To address this issue, an effective solution is to temporarily leave the jump statements unspecified. Each such statement is placed on a list. As labels are determined, they are filled in, and the corresponding statement is removed from the list. This process of assigning labels subsequently is known as *backpatching*.

To handle the list of labels, following functions are employed:

1. $makelist(i)$: Creates a new list containing only $i$, an index into the array of quadruples. It returns a pointer to the newly created list.
2. $merge(p_1, p_2)$: Concatenates the lists pointed to by $p_1$and $p_2$. Returns a pointer to the concatenated list.
3. $backpatch(p, i)$: Inserts $i$ as target label for each of the statements on the list pointed to by $p$.

**Boolean Expressions**

We use the following grammar to produce quadruples(intermediate code) for the Boolean expressions. A marker non-terminal is used for convenience.

$$
\begin{aligned}
E \rightarrow\ & E_1\ \boldsymbol{or}\ M\ E_2 \\
& |\ E_1\ \boldsymbol{and}\ M\ E_2 \\
& |\ \boldsymbol{not}\ E_1 \\
& |\ (E_1) \\
& |\ \boldsymbol{id_1\ relop\ id_2} \\
& |\ \boldsymbol{true} \\
& |\ \boldsymbol{false} \\
M \rightarrow\ & \epsilon
\end{aligned}
$$

As code is generated for the non-terminal $E$, jumps to the $true$ and $false$ exits are left incomplete, i.e., the label field in the “goto” is unfilled. The following attributes are used to keep track of the unfilled jumps.

1. $E.truelist$: It is a pointer to the list containing the unfilled jumps corresponding to $E = true$.

2. $E.falselist$: It is a pointer to the list containing the unfilled jumps corresponding to $E = false$.

Consider the production $E \rightarrow E_1\ \boldsymbol{and}\ M\ E_2$. If $E_1$ is $false$, then $E$ is false. Hence, the incomplete jumps(gotos) on $E_1.falselist$ become part of $E.falselist$. If $E_1$ is $true$, then we must test $E_2$. Therefore, the target for the jumps on $E.truelist$ must be the beginning of the code generated by $E_2$. The marker non-terminal $M$ is used to obtain the target. The attribute $M.quad$ gives the index of the first statement of $E_2$, in the quadruples table. The production $M \rightarrow \epsilon$, is associated with the semantic rule $M.quad = nextquad$. The variable $nextquad$ holds the index of the next quadruple to follow. This value will be backpatched to the statements on the $E_1.truelist$. The translation scheme is shown below.

| Production | Semantic Rule |
|---|---|
| $E \rightarrow E_1\ \boldsymbol{or}\ ME_2$ | $\{\ backpatch(E_{1.}falselist, M.quad);$<br>$E.truelist = merge(E_1.truelist\ , E_2.truelist);$<br>$E.falselist = E_2.falselist\ \}$ |
| $E \rightarrow E_1\ \boldsymbol{and}\ ME_2$ | $\{\ backpatch(E_{1.}truelist, M.quad);$<br>$E.falselist = merge(E_1.falselist\ , E_2.falselist);$<br>$E.truelist = E_2.truelist\}$ |
| $E \rightarrow \boldsymbol{not}\ E_1$ | $\{\ E.truelist = E_1.falselist$<br>$E.falselist = E_1.truelist\ \}$ |
| $E \rightarrow (\ E_1\ )$ | $\{\ E.truelist = E_1.truelist$<br>$E.falselist = E_1.falselist\ \}$ |
| $E \rightarrow \boldsymbol{id_1}\ \boldsymbol{relop}\ \boldsymbol{id_2}$ | $\{\ E.truelist = makelist(nextquad);$<br>$E.falselist = makelist(nextquad + 1);$<br>$emit('if'\ \boldsymbol{id}_1.place\ \boldsymbol{relop}.op\ \boldsymbol{id_2}.place\ 'goto\ \_')$<br>$emit('goto\ \_')\ \}$ |
| $E \rightarrow \boldsymbol{true}$ | $\{\ E.truelist = makelist(nextquad);$<br>$emit('goto\ \_')\ \}$ |
| $E \rightarrow \boldsymbol{false}$ | $\{\ E.falselist = makelist(nextquad);$<br>$emit('goto\ \_')\ \}$ |
| $M \rightarrow \epsilon$ | $\{\ M.quad = nextquad\ \}$ |

**Table 5.9** Translation scheme Boolean expressions using backpatching

**Flow of Control Statements**

In this section, we will explore the application of backpatching in translating flow-of-control statements, with a specific emphasis on generating quadruples—one of the representations for three-address codes. To illustrate these concepts more comprehensively, we will develop the translation scheme for the statements generated by the grammar outlined below.

$S \rightarrow \boldsymbol{if}\ E\ \boldsymbol{then}\ S$
$\quad |\ \boldsymbol{if}\ E\ \ \boldsymbol{then}\ \ S\ \boldsymbol{else}\ S$
$\quad |\ \boldsymbol{while}\ E\ \ \boldsymbol{do}\ S$
$\quad |\ \boldsymbol{begin}\ L\ \boldsymbol{end}$
$\quad |\ A$
$L\ \rightarrow L\ \ ;S$
$\quad |\ S$

In the above grammar the non-terminals carry the following meanings:

- $S$ : denotes a statement
- $L$ : is a statement list
- $A$: is an assignment statement
- $E$ : is a Boolean expression

The above productions are sufficient enough to demonstrate the techniques to translate the flow-of-control statements. The general approach is to generate the jumps(goto statements) without labels and subsequently fill them as their targets are identified.

**Scheme to implement the Translation**

Following are the attributes used in the translation scheme.

- $E.truelist$ and $E.falselist$ and $M.quad$ as used in the previous section.
- $L.nextlist$ and $S.nextlist$ are pointers to the lists of all conditional and non-conditional jumps to the statements following $L$ and $S$ respectively in the quadruple list.
- $S.begin$ marks the beginning of the code for the complete statement $S$.
- $E.true$ marks the beginning of the code for the complete statement $S_1$.
- $N.nextlist$ is the list containing the quadruple number of the statement "`goto _`" that is generated by the semantic rule for $N$.

| Production | Semantic Rule |
|---|---|
| $S \rightarrow \boldsymbol{if}\ E\ \boldsymbol{then}\ M_1\ S_1\ N\ \boldsymbol{else}\ M_2 S_2$ | $\{\ backpatch(E.truelist, M_1, quad);$<br>$backpatch(E.falselist, M_2, quad);$<br>$S.nextlist =$<br>$merge\big(S_1.nextlist, merge(N.nextlist, S_1.nextlist)\big)\ \}$ |
| $N \rightarrow \epsilon$ | $\{\ N.nextlist = makelist(nextquad);$<br>$emit('goto\ \_'\ )\ \}$ |
| $M \rightarrow \epsilon$ | $\{\ M.quad = nextquad\ \}$ |
| $S \rightarrow \boldsymbol{if}\ E\ \boldsymbol{then}\ M\ S_1$ | $\{\ backpatch(E.truelist, M.quad);$<br>$S.nextlist = merge(E.falselist, S_1.nextlist)\ \}$ |
| $S \rightarrow \boldsymbol{while}\ M_1\ E\ \boldsymbol{do}\ M_2\ S_1$ | $\{\ backpatch(S_1.nextlist, M_1.quad);$<br>$backpatch(E.truelist, M_2.quad);$<br>$S.nextlist = E.falselist;$<br>$emit('goto'\ M_1.quad)\ \}$ |
| $S \rightarrow \boldsymbol{begin}\ L\ \boldsymbol{end}$ | $\{\ S.nextlist = L.nextlist\ \}$ |
| $S \rightarrow A$ | $\{\ S.nextlist = nil\ \}$ |
| $L \rightarrow L_1\ ;\ M\ S$ | $\{\ backpatch(L_1.nextlist, M.quad);$<br>$L.nextlist = S.nextlist\ \}$ |
| $L \rightarrow S$ | $\{\ L.nextlist = S.nextlist\ \}$ |

**Table 5.10** Translation scheme for flow-of-control statements

# 5.9 Procedure Calls

The procedure is an important and frequently used programming construct. In this section we discuss the code that is generated for procedure calls and returns. The grammar for a simple procedure call is shown below.

$S \rightarrow \boldsymbol{call\ id}\ (\ Elist\ )$

$Elist \rightarrow Elist, E$

$Elist \rightarrow E$

Where $E$ is an expression and $Elist$ is the list of expressions.

**Calling Sequences**

The calling sequence is the sequence of actions taken during an entry to and exit from the procedure. The compiler must generate the calling sequence while translating a procedure call statement. The following actions typically constitute the calling sequence.

**Procedure Call:**

- Allocate the space for the activation record of the called procedure.
- Evaluate the arguments of the called procedure and make them available to the called procedure in a known location.
- Establish the environment pointers enabling the called procedure to access the data in the enclosing blocks.
- Save the state of the calling procedure.
- Save the return address in a known location.
- Generate a jump to the beginning of the code for the called procedure.

**Procedure Return:**

- If the called procedure is a function store the result in a known location.
- Restore the activation record of the calling procedure.
- Generate a jump to the calling procedure’s return address.

A convenient data structure to store the activation record, the state of the calling procedure and retrun address of the called procedure is *stack*.

## 5.10 Multiple Choice Questions

Q1. Which of the following is not a commonly used intermediate representation in compiler design?

a. Abstract Syntax Tree (AST)
b. Three-Address Code (3AC)
c. Assembly language
d. Quadruples

Q2. Which intermediate language is closer to the high-level source code and represents the program's control flow graph?

a. Three-Address Code
b. Quadruples
c. Intermediate Code
d. Intermediate Representation

Q3. The primary advantage of using an intermediate language during the compilation process is:

a. It reduces the time required for lexical analysis
b. It makes the compiler implementation simpler
c. It allows for machine-independent optimizations
d. It eliminates the need for a symbol table

Q4. Which of the following statements is true regarding the choice of an intermediate language in compiler design?

a. The intermediate language is always machine-specific
b. The choice of intermediate language has no impact on compiler efficiency
c. The intermediate language should strike a balance between machine independence and efficiency
d. Intermediate languages are only used in interpreters, not compilers

Q5. Which phase of the compiler is responsible for translating the high-level source code into the intermediate language?

a. Lexical analysis
b. Semantic analysis
c. Intermediate code generation
d. Code optimization

Q6. What is the primary purpose of intermediate code in a compiler?

a. Enhancing runtime performance

b. Representing the program's logic in a machine-independent form

c. Improving lexical analysis

d. Directly generating machine code

Q7. In the context of intermediate code generation, what does DAG stand for?

a. Directed Acyclic Graph

b. Data Analysis Graph

c. Directed Assembly Generator

d. Dynamic Algorithm Generator

Q8. What is the main purpose of backpatching?

a. Resolving dependencies between different compiler phases

b. Generating optimized machine code

c. Updating addresses in the generated code after code generation

d. Enhancing the efficiency of lexical analysis

Q9. What does the term "backpatch" refer to in the context of compiler design?

a. Correcting errors in the source code

b. Retroactively updating addresses in the generated code

c. Enhancing the backward compatibility of a compiler

d. Improving the performance of lexical analysis

Q10. In the context of code generation, what is a typical scenario where backpatching is useful?

a. Handling syntax errors

b. Resolving jumps in control flow statements

c. Optimizing arithmetic expressions

d. Constructing symbol tables

## 5.11 Summary

Intermediate code generation is an important phase in the compilation process and is situated between the syntax analysis and code generation stages. The primary goal is to produce an intermediate representation of the source code that is both machine-independent and more agreeable to optimization. This intermediate code acts as a bridge, capturing the essential semantics of the source code while abstracting away machine-specific details.

The generated intermediate code is designed to be independent of both the source programming language and the target machine architecture. This independence allows for flexibility in compiling programs written in different languages for various platforms.

During intermediate code generation, some fundamental optimizations may be applied, including constant folding, common subexpression elimination, and dead code elimination. These optimizations target improvements in the efficiency and performance of the final generated code.

Backpatching is a technique frequently employed during intermediate code generation to manage situations where the target addresses of jumps or branches are unknown until later stages of compilation. It involves retroactively updating these addresses.

In summary, intermediate code generation assumes a critical role in the compilation process. By creating an intermediate representation, it facilitates subsequent analysis, optimization, and translation into machine code. This process contributes to the generation of efficient and target-specific executable code while maintaining a level of abstraction from the intricacies of the source language.

## 5.12 Key Words

- Syntax Trees
- Directed Acyclic Graphs
- Quadruples
- Triples
- Indirect Triples
- Short Circuit Code
- Backpatching