UNIT-III

INTERMEDIATE CODE GENERATION

3.1 DEFINITIONS

Form of Syntax Directed Definition:

In a syntax directed definition a production the form

$A \rightarrow \propto$ has a set of semantic rules

$b := f(C1, C2, ..., Ck)$ where f is a function and either.

There are two methods for associating semantic rules with the production.

They are: (1) Syntax directed definition. (2) Translation scheme.

Syntax Directed Definition:

They are high level specifications for translation. They hide implementation details and free the user from specifying explicitly the order in which translation takes place.

3.2 EVALUATION ORDERS FOR SYNTAX DIRECTED DEFINITION

Syntax Directed definition is a generalization of CFG in which each grammar symbol is associated with a set of attributes. Attributes are divided into 2 sets called synthesized and inherited attributes.

Synthesized Attributes:

Sythesized attributes are frequently used.

Syntax directed definition that uses synthesized attribute is used extensively is called as S-attributed definition.

Inherited Attribute:

Inherited attribute value of a node derives its value from the parent and or siblings of the node if any. It is useful for expressing the dependency.

Example: Inherited attribute is used to keep track of the appearance of the attribute to the right or left of the assignment. This helps to decide whether is the l-values of r-values.

Example: Consider the production and its semantic check.

Production	*Semantic Rules*
$D \rightarrow TL$	L.in = T.Type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ real	T.type = real
$L \rightarrow L1$ id	L1.in = L.in
$L \rightarrow$ id	add type (id.entry, L.in)
	add r type (id . entry , L . in).

Keywords: int, real.

$T \rightarrow$ non-terminal has a synthesized attribute type value is determined by the key word in the declaration.

Parse Tree with inherited attribute

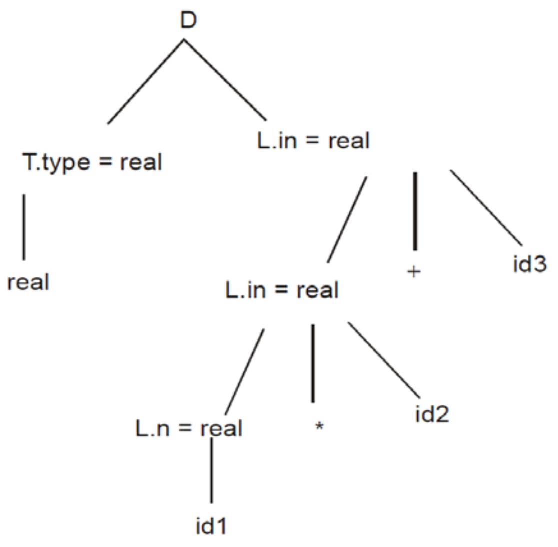


Figure 3.1

Translation Scheme:

Indicate the order in which semantic rules are to be evaluated, hence some implementation details are to be shown.

Conceptual view of Syntax directed translation:

In both syntax directed definition and translation the steps followed are the same. Parse the input token, build the parse the, transverse the free as needed to evaluate the semantic rules at the parse tree nodes. Evaluation of the semantic rules generate code, save information in a symbol table, issue error messages. The translation of the token stream is the result of evaluating the semantic rules.

input string $\rightarrow$ parse tree $\rightarrow$ dependency $\rightarrow$ evaluation order for semantic rule.

It is not necessary that all implementation needs to follow the same rule. Syntax directed translation can be implemented in single pass by evaluating semantic rules during parsing without the construction parse tree or dependency graph.

As it is single pass compile time efficiency is important.

Example 3.1:

Syntax directed definition of a simple desk calculator.

Production	**Semantic Rules**
$L \rightarrow En$	Print (E . Val)
$E \rightarrow E_1 + T$	E . Val = E1 . Val + T . Val
$E \rightarrow T$	E . Val = T . val
$T \rightarrow F$	T . Val = F . Val
$F \rightarrow (E)$	F . Val = E . Val
$F \rightarrow$ digit	F . Val = digit lexval.

The token digit has a synthesized attribute lexval whose value is assumed to be supplied by the lexical analyzer. The rule associated with the production

$L \rightarrow En$ for the starting non-terminal L is a procedure between prints as output the value of the arithmetic expression generated by E.

In a syntax directed definition terminals can have synthesized attributes since there does not exist and semantic rules for terminals.

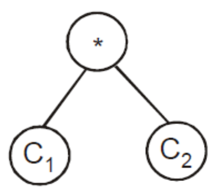
Values for attributes of terminal are supplied by the lexical analyzer.

Annotated Parse Tree:

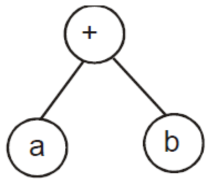
Parse tree showing the values of attribute at each node.

The process of computing the attribute values at the node is called annotating or decorating the parse free.

Example:



(i) $C_1 * C_2$



(ii) (a+b)

3.3 INTERMEDIATE LANGUAGES

3.3.1 SYNTAX TREE

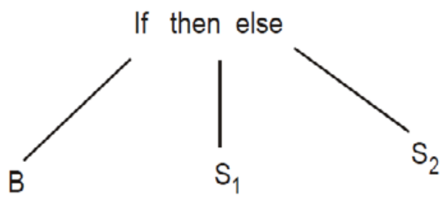
The syntax tree is an intermediate representation. This allows translation removed from parsing. Translation routines are called during parsing but with 2 restriction.

1) A grammar suitable for parsing may not reflect the natural hierarchical structure of the language.
2) The parsing method constraints the modes in which the parse tree are considered.

Syntax Tree:

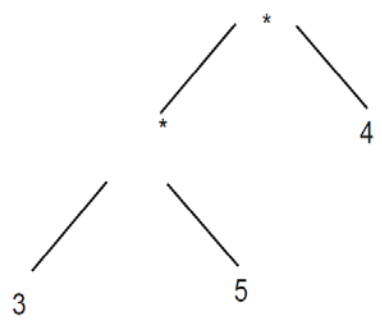
Syntax Tree is a form of parse tree which is useful to represent the language construct. Consider the production:

S $\rightarrow$ if B then S1 else S2



In Syntax Tree operators and key words are present in the intermediate nodes that would be the pattern of the leaves in the Parse Tree.

Parse Tree for the expr 3 * 5 + 4



BOTTOM UP EVALUATION OF S-ATTRIBUTE

Parse Tree for an S-attributed definition can be annotated by evaluating the semantic rules for the attributes at each node (Bottom up evaluation).

Example: S-attribute definition for the Examples 3 * 5 + 4*n*

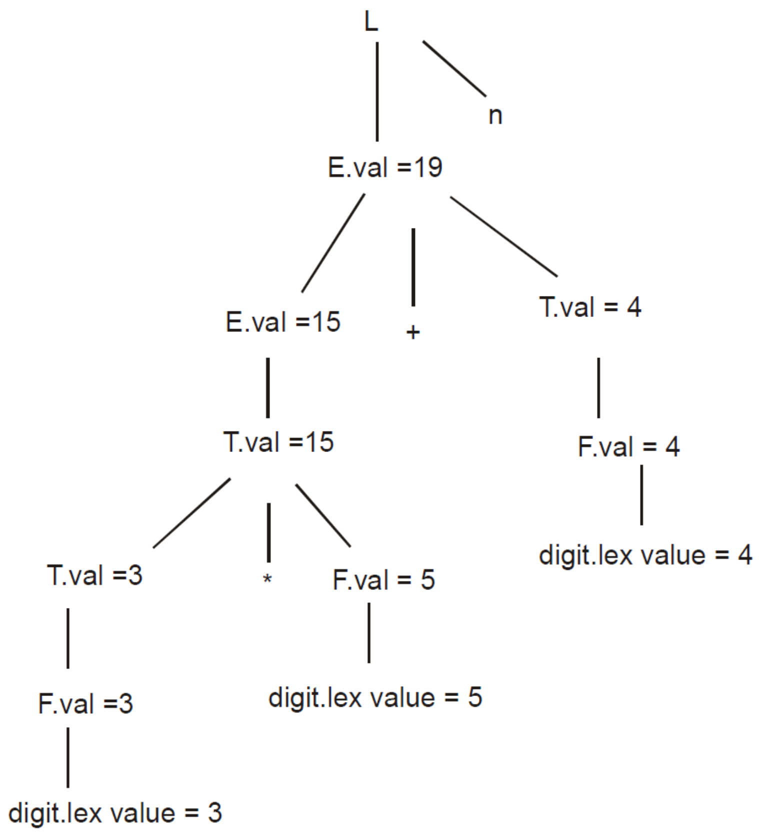


Figure 3.2

How to calculate the values.

Let as draw the LMD to understand.

$L \rightarrow Ea$

$E \rightarrow E1 + T$

$E \rightarrow T + T$

$E \rightarrow T * F + T$

$E \rightarrow F * F + T$

$E \rightarrow F * F + F$

$E \rightarrow digit * F + F$

E $\rightarrow$ digit * digit + F

E $\rightarrow$ digit * digit + digit

E $\rightarrow$ 3 * digit + digit

E $\rightarrow$ 3 * 5 + digit

E $\rightarrow$ 3 * 5 + 4

E $\rightarrow$ 19

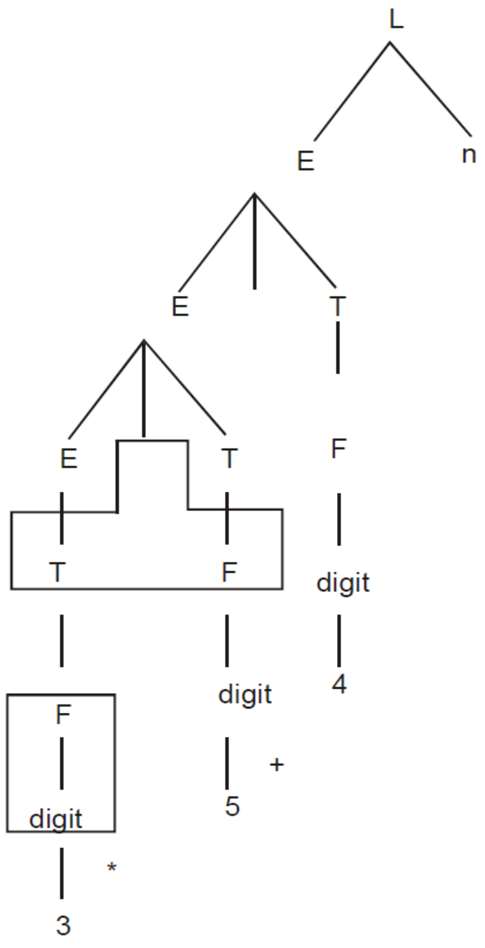


Figure 3.3

15 + 4 = **19**

Consider the square region which corresponds to the production F $\rightarrow$ digit. The semantic rule in F . Val = digit.lex val is 3 hence to takes the value 3. Consider the mode for production

T * F.

T $\rightarrow$ T * F

The equivalent semantic rule is

$T.val = T_1 . val * F . val$

This has 3 from left child and 5 from the right child.

Hence 3 * 5 = 15.

The value associated with the production for the starting. Non-Terminal L $\rightarrow$ En gives the output generated by 5.

3.3.2 THREE ADDRESS CODE

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

$x := y\ op\ z,$

where *x, y, z* are names, constants or compiler generated temporary variables.

where *op* stands for any operator such as fixed or floating point arithmetic operator or a logical operator.

Thus 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 the compiler generated temporary name.

Advantages:

1) 3 address code is desirable for target code generation and optimization.
2) The use of names for the intermediate values computed by a program allows 3 address code to be easily rearranged unlike postfix notation.

   The syntax tree and DAG are represented by the following 3 address code sequence:

$$a = b * c + b * c.$$

Code for syntax tree;	**Code for dag;**
$t_1 = b * c$	$t_1 = b * c$
$t_2 = b * c$	$t_2 = t_1 + t_1$
$t_3 = t_1 + t_2$	$a = t_2$
$a = t_3$	

The reason for the term "3 address code" is that each statement usually contains 3 addresses, 2 for operands and one for the result.

Types of 3 address statements:

The common 3 address statements are:

1) Assignment statements of the form $x := y \; op \; z$ where *op* is a binary arithmetic or logical operation.
2) Assignment instructions of the form $x := op \; y$ where *op* is a unary operation.
3) Copy statements of the form $x := y$.
4) The unconditional jump goto L.
5) Conditional jumps such as "if *x* relop *y* goto L".
6) Indexed assignments of the form $x := y[i]$ and $x[i] := y$.
7) (7) Address and pointer assignments of the form $x := * \, y$, $x := * \, y$, $* \, x := y$.
8) param x_1 , param x_2 , call *p*, *n*, return *y* for procedure calls.

Implementations of three address statements:

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

The three address codes are generally implemented using any one of the following representations:

1) Quadruple
2) Triple
3) Indirect Triple.

Quadruple:

A record structure is used to represent 3 address codes with 4 fields are listed below:

(1) op (operator)	(2) arg1
(3) arg2	(4) result

The 'op' field contains an internal code for the operator.

The 3 address statement $x := y \; op \; z$ is represented by placing *y* in arg1, z in arg2 and 'x' in result.

The contents of fields arg1, arg2 and result are normally pointers to the symbol table entries for the names represented by 3 fields. Statements with unary operators like $x := \; y$, $x = y$ do not use arg2.

Quadruples:

	op	**arg1**	**arg2**	**result**
0	*	b	c	t_1
1	*	b	c	t_2
2	+	t_1	t_2	t_3
3	:=	t_3	a	-

Triples:

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

If use do so 3 address statements can be represented by records with only 3 fields op, arg1 and arg2. The fields arg1 and arg2 are either pointers to the symbol table or pointers into the triple structure. Paranthesized numbers represents pointer to the triple structure. While the symbol table pointers are represented by the name themselves.

	op	**arg1**	**arg2**
(0)	*	b	c
(1)	*	b	c
(2)	+	(0)	(1)
(3)	:=	a	(2)

Indirect Triples:

Another implementation of 3 address code is that, listing pointers to triples rather than listing the triples themselves. This implementation is naturally called indirect triples.

	statement
(0)	(101)
(1)	(102)
(2)	(103)
(3)	(104)

	op	**arg1**	**arg2**
101	*	b	c
102	*	b	c
103	+	(101)	(102)
104	assign	a	(103)

Comparison of Representations:

- Using quadruple notation a 3 address statement defining or using a temporary can immediately access the location for that temporary via the symbol table.

- In quadruple notation, if we move a statement computing '*x*', the statements using '*x*' require no change. However in the triple notation moving a statement that defines a temporary value requires is to change all references to that statement. This problem makes triples difficult to be used in an optimizing compiler. Indirect triples presents no such problem.

  Quadruple and indirect triples notations requires the same amount of space and they are equally efficient for re-ordering of code. However indirect triples can save some space compared with quadruples.

3.3.3 TYPES AND DECLARATIONS

Definition: A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system.

Different type systems can be used by the same language for different compilers.

Static and Dynamic Type Checking:

When the checking is done by compiler it is called as static type checking.

When the checking is done during run time to the target program is called as dynamic checking.

Dynamic checking is possible only when the target program has the type of an element and the value of the element.

A language is strongly typed if its compiler can guarantee if the input program can be executed without type errors.

Error Recovery:

The function of type checking is to detect the error in the program. Still additional functionality is needed when errors are discovered.

1) The simplest is to report the nature and the location of the error.
2) To recover from errors, and can continue type checking.
3) Error handling affects the type checking rules hence rules must be prepared to cope with the errors.

3.3.4 TRANSLATIONS OF EXPRESSIONS

The checking rules are of the form:

if two type expressions are equal then return a certain type else return type-error.

Hence it is important to have preline definition when the two type expressions are equivalent.

Structural Equivalence:

Two expressions are said to be structural equivalent

(i) If they are formed by applying the same constructor are they are of same type.

(ii) Two type expressions are said to be structurally equivalent iff they are identical.

(eg.) Type expression integer is equivalent to integer only because they are same basic type.

Pointer (integer) is equivalent to pointer (integer) because they are formed by applying the same construction.

Algorithm to Check for Structural Equivalence:

```
(1) function sequiv ($s$, $t$) : boolean;
        begin
(2) if $s$ and $t$ are the same basic type then
(3) return true
(4) else if $s$ = array ($s_1$, $s_2$) and $t$ = array ($t_1$, $t_2$) then
(5) return sequiv ($s_1$, $t_1$) and sequiv ($s_2$, $t_2$)
(6) else if s = $s_1 \times s_2$ and $t = t_1 \times t_2$ then
(7) return sequiv ($s_1$ , $t_1$) and sequiv ($s_2$ , $t_2$)
(8) else if s = pointer ($s_1$) and $t$ = pointer ($t$1) then
(9) return sequiv ($s_1$ , $t_1$)
(10)        else if s = $s_1 \to s_2$ and t = $t_1 \to t_2$ then
(11)        return sequiv ($s_1$, $t_1$) and sequiv ($s_2$ , $t_2$)
        else
(12)        return false
        End
```

Figure 3.5

Names for Type Expression:

In languages like Pascal types can be given names.

Example: type link = * cell;

var next = link; P : * cell;

last = link;

Link is an identifier, declared as a name for type

↑ Cell The Question is whether next, last, P all have identical types?

The answer depends upon the implementation. Because, in Pascal they did not define "identical type".

Cycles in Representation of Types:

Linked lists, Trees are defined recursively. Such data structures are implemented in Pascal by records or struct in 'e'.

```
type link = ↑cell;
    cell = record
        info : integer;
        next : link.
    end;
```

Link in defined in terms of cell.

Cell is defined in terms of link.

Hence is is a recursion.

Figurately represented as:

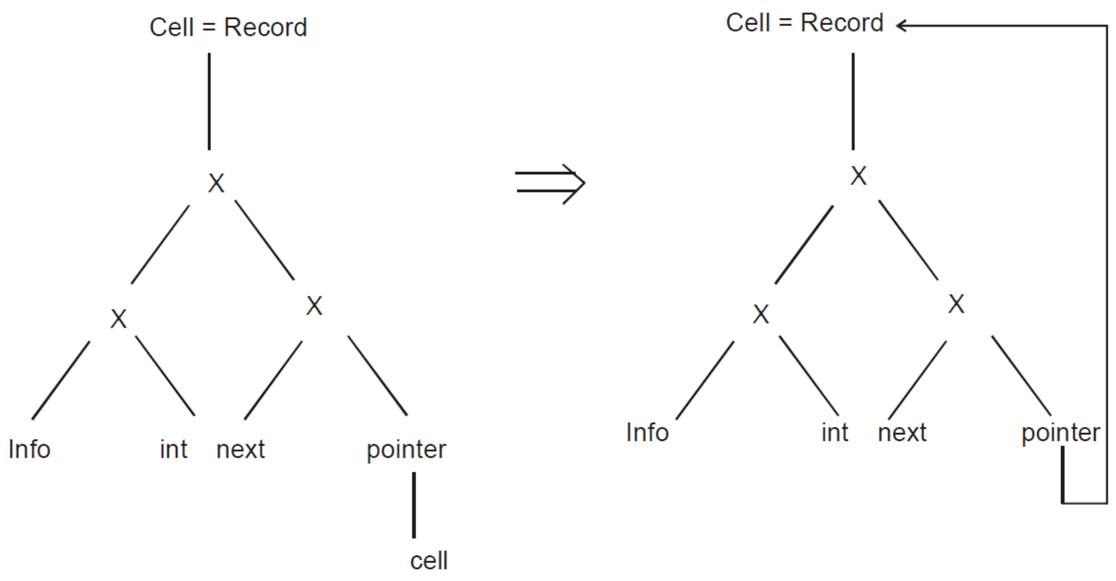


Figure 3.4

C avoids cycles:

```
struct cell
```

```
{
        int info;
        struct cell * next;
}
```

needs the type names to be declared before they are used.

3.3.5 TYPE CONVERSIONS

Consider the expression *x* + *i* where *x* is type float and *i* is type integer.

Within the system representation of integer and float are different and hence different machine instructions are used for operations on integer and real.

The compiler needs to convert one of the operand, so that both operands are of same type when addition takes place.

Usually integer is converted to real and perform the operation.

x + *i* in post fix representtaion will be

x i int real real + ..

(inttoreal) operator converts integer to real and real + perform real addition on its operands.

Coercions:

Conversion from one type to another is said to be implicit if it is to be done automatically by the compiler.

Implicit conversions are called as coercions.

Conversions is said to be explicit if the programmer writes the instruction for conversion.

3.3.6 SPECIFICATION OF SIMPLE TYPE CHECKER

The type checker is a translation scheme, that generates the type of each expression from the types of subexpresisons.

It can handle arrays, pointers, statements and functions.

Example: Consider grammar

P → D; E

D → D; D|id : T

T → Char|int|array [num] of T| T

E → literal|num|id|E mod E| E[E] | E .

Some generations of this grammar

key : integer;

key mod 1999.

The data types of the language are char, integer, type-error to signal errors.

Example: array [256] of char.

array (1..256, char) assuming the index starts at 1.

$\uparrow$ integer $\Rightarrow$ Pointer (integer).

The grammar is translated as:

P $\rightarrow$	D ; E	
D $\rightarrow$	D ; D	
D $\rightarrow$	**id** : T	{addtype (**id**.entry, T.type)}
T $\rightarrow$	**char**	{T.type := char}
T $\rightarrow$	**integer**	{T.type := integer}
T $\rightarrow$	$\uparrow$ T1	{T.type := pointer (T1.type)}
T $\rightarrow$	**array [num] of T**1	{T.type := array (1..**num**.val, T1.type)}

(i) The Semantic Rules

E $\rightarrow$ literal {E.type = char}

E $\rightarrow$ num {E.type = integer}.

Use a lookup (e) to fetch the type saved in the symbol table entry pointed by e. When an (d) is present in the expr the production changes as E $\rightarrow$ id.

(ii) Consider the Production

E $\rightarrow$ E_1 mod E_2.

The symantic rule will be:

{E.type = if E_1.type = integer and E_2.type = integer then integer else type-error}.

Type Checking of Statements:

As the statements do not have value special type called void can be assigned to them. When an error is detected the type assigned to the statement is type error.

Example: Assignment statement, Conditional statement, Statements separated by *i* ;

The statement and in translation for type checking.

S $\rightarrow$ id = E {S.type := if id.type = E.type then void else type-error}

$S \rightarrow$ if E then S1 {S.type = if E.type = boolean then S1.type else type-error}

$S \rightarrow$ which E do S1 {S.type = if E type = boolean then S1.type else type-error}

$S \rightarrow$ S1; S2 {S.type = if S1.type = void and S2.type; void then void else type-error}.

Type Checking for Functions:

The production $E \rightarrow E(E)$

Here the function is an argument.

Expression is the application of one expression to another.

The rule for associating type expression with Non-Terminal T can be argumented by the production.

$T \rightarrow T1$ ‘ $\rightarrow$ ’ T2 {T.type = T1.type $\rightarrow$ T2.type}.

Rule for checking the type {a function application is $F \rightarrow E_1$ (E_2), E.type = if E_2.type = S and E_1 type = $S \rightarrow t$ then t else type-error}.

3.4 BOOLEAN EXPRESSIONS

Boolean expressions are composed of the boolean operators (and, or and not). We shall now consider Boolean expressions generated by the following grammar:

$E \rightarrow$ E or E | E and E | not E | (E) | id relop id | true | false

- ‘or’ and ‘and’ are left associative
- ‘or’ has lowest precedence than “and”, then “not”.

Methods of Translating Boolean Expressions

There are two principal methods of representing the value of a boolean expression.

FIRST METHOD

1) Encoding True and False numercially.

- ‘1’ is used to denote true.
- ‘0’ is used to denote false.

(or)

- Any non-zero quantity may be used to denote true.
- And zero may denote false.

(or)

- We could let any non-negative quantity denote true.
- Any negative number denote false.

SECOND METHOD

2) Representing the value of a boolean expression by a position reached in a program.

For example, given the expression E_1 or E_2, if we determine that E1 is true, then we can conclude that the entire expression is true without having to evaluate E_2.

We shall now see both methods for the translation of boolean expressions to three address code.

Numerical Representation

Let us first consider the implementation of boolean expressions using '1' to denote true and '0' to denote false. Expressions will be evaluated completely, from left to right, in a manner similar to arithmetic expressions.

For example the translation for:

a or *b* and not *c*

is the three address sequence which follows:

```
t₁ := not c       // 'not' has the highest precedence
t₂:= b and t₁     // and has the next highest precdence
t₃:= a or t₂      // 'or' has the least precedence.
```

A relational expression such as $a < b$ is equivalent to the conditional statements if $a < b$ then 1 else 0, which can be translated into the three address code sequence.

***Note*:**

Assume that "emit" places three address statements into an output file in the right format and "nextstat" gives the index of the next three addess statement in the output sequence, and that emit increments nextstat after producing each three address statement.

- "E . place" refers to the name that will hold the value of E.
- "newtemp" generates the temporary variables as t_1 , t_2 , ...

3.5 TRANSLATION SCHEME USING A NUMERICAL REPRESENTATION FOR BOOLEANS

$E \rightarrow E_1$ or E_2 {E . place := newtemp;

emit (E . place ‘:=’ E_1. place ‘or’ E_2. place)}

$E \rightarrow E_1$ and E_2 {E . place := newtemp;

emit (E . place ‘:=’ E_1. place ‘and’ E_2. place)}

$E \rightarrow$ not E_1 {E . place := newtemp;

emit (E . place ‘:=’ ‘not’ E1. place)}

$E \rightarrow (E_1)$ {E . place := E_1 . place}

$E \rightarrow id_1$ = relop id_2 {E . place := newtemp;

emit (‘if’ id_1. place relop.op id_2. place ‘goto’ nextstat + 3);

emit (E . place := ‘0’);

emit (‘goto’ nextstat + 2);

emit (E . place ‘:=’ ‘1’)}

$E \rightarrow$ true {E . place := newtemp;

emit (E . place ‘:=’ ‘1’)}

$E \rightarrow$ false {E. place := newtemp;

emit (E . place ‘:=’ ‘0’)}

The scheme would generate the three address code as follows for the expression, $a < b$ and $c < d$.

100: if a < b goto 103

101: t_1:= 0

102: goto 108

103: t_1 := 1

104: if c < d goto 107

105: t_2:= 0

106: goto 108

107: t_2:= 1

108: $t_3 = t_1$ and t_2 .

Short Circuit Code

We can also translate a boolean expression into three address code without generating code for any of the boolean operators and without having the code necessarily evaluate the entire expression. This style of evaluation is sometimes called “short circuit” or “jumping” code.

Flow of Control Statements

We now consider the translation of boolean expression into three address code in the context of if-then, if-then-else, and while-do statements such as those generated by the following grammar.

S $\rightarrow$ if E then S1

| if E then S_1 else S_2

| while E do S_1.

In the translation we assume that a three address statement can be symbolically labeled and that the function newlabel returns a new symbolic label each time it is called. We assume that “E. true” is the label to which control flows if E is true. “E . false” is the label to which control flows if E is false. “S . next” is a label that is attached to the first three address instruction to be executed after the code for S.

In translating the if-then statement S $\rightarrow$ if E then S1, a new label “E . true” is created and attached to the first three address instruction generated for the statement S_1 is as follows:

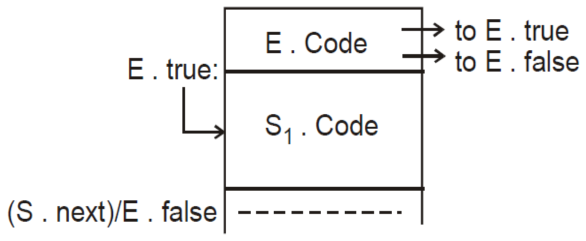


Figure 3.5

In translating the if then else statement, S $\rightarrow$ if E then S_1 else S_2 the code for the Boolean expression E has a jump to the first instruction of the code for S_1 if ‘E’ is true and to the first instruction of the code for S_2 if E is false as shown below.

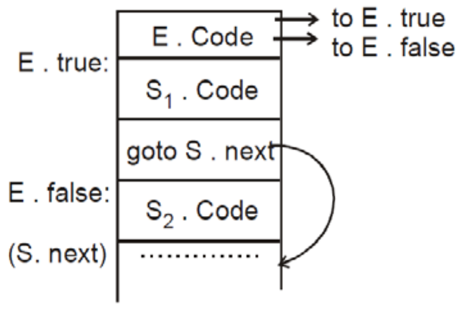


Figure 3.6

While-Do

The code for S → while E do S1 is formed as shown below:

A new label "S . begin" is created and attached to the first instruction generated for E. Another new label E . true is attached to the first instruction for S_1.

The code for 'E' generates a jump to this label if E is true and jump to S. next if E is false.

After the code for S_1 we place the instruction goto S. begin which causes a jump back to the beginning of the code for the boolean expression.

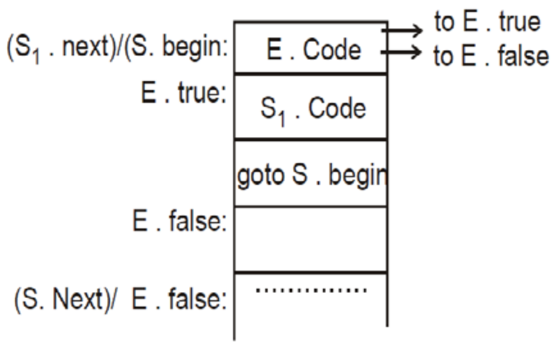


Figure 3.7

Syntax directed definition for flow of control statements

Production	***Semantic Rules***

S → if E then S_1	E . true := newlabel;
	E . false := S . next;
	S_1 . next := S . next;
	S . code := E . Code || gen (E . true ':') || S_1 . code

S → if E then S_1 else S_2	E . true := newlabel;

	E . false := newlabel; S_1 . next := S . next; S_2 . next := S . next; S . code := E . Code || gen (E . true ':') || S_1 . code || gen ('goto' S . next) || gen (E . false ':') || S_2 . code
S $\rightarrow$ while E do S_1	S . begin := newlabel; E . true := newlabel; E . false := S . next; S_1 . next := S . begin; S . code := gen (S . begin ':') || E . code || gen (E . true ':') || S_1 . code || gen ('goto' S . begin).

Control Flow Translation of Boolean Expression

If E . code is the code produced for the boolean expressions E, E is translated into a sequence of three address statement that evaluates 'E' as a sequence of conditional and unconditional jumps to one of the two locations. The basic idea behind the translation is the following:

- Suppose 'E' is of the form $a < b$.

  Then the generated code is of the form:

  if $a < b$ goto E . true

  goto E . false

Suppose E is of the form E_1 or E_2 . If E_1 is true, then we immediately know that E itself is true. So E_1 . true is same as E. true.

If E_1 is false, E_2 must be evaluated, so we make E_1. false be the label of the first statement in the code for E . In such case when E_2 is false, 'E' is also false else if E_2 is true E is also true. For expression E of the form not E_1. We just interchange the true and false exits of E.

Syntax directed definition to produce Three Address Code for Boolean.

NOTE: gen function prints the 3 address code to be generated.

Production	**Semantic Rules**
$E \to E_1$ or E_2	E_1 . true := E . true; E_1 . false := newlabel; {Thin label points to first statement of E_2 } E_2 . true := E . true; E_2 . false := E. false; E_2 . code := E_1 . code || gen (E_1 . false ‘:’) || E_2 . code
$E \to E_1$ and E_2	E_1 . true := newlabel; || which points to the 1st stmt of E_2 E_1 . false := E . false; || if E_1 is false, E is also false E_2 . true := E . true; E_2 . false := E . false; E . code := E_1 . code || gen (E_1 . true ‘:’) || E_2 . code
$E \to$ not E_1	E_1 . true := E . false; E_1 . false := E . true; E . code := E_1 . code
$E \to (E_1)$	E_1 . true := E . true E_1 . false := E . false; E . code := E_1 . code
$E \to id_1$ relop id_2	E . code = gen (‘if’ id_1 . place relop op id_2 . place ‘goto’ E . true || gen (‘goto’ E . false)
$E \to$ true $E \to$ false	E . code := gen (‘goto’ E . true) E . code := gen (‘goto’ E . false).

Let us consider the expression which follows:

$a < b$ and $c < d$

Suppose the true and false exits for the entire expression have been set to L true and L false. Then using the preceeding syntax directed definition we would obtain the following code:

```
if a < b goto L1
        goto L . false
        L1: if c < d goto L . True
              goto L false.
```

***Example* 2:**

Let us consider the expression:

$a < b$ or $c < d$.

The corresponding 3 address code is

```
    if a < b goto L . true
          goto L1
    L1: if c < d goto L . true
          goto L . false
```

NOTE: that the code generated by the method is not optimal. Because the 2nd statement can be eliminated without changing the value of the code. So, optimization technique must be used to eliminate such redundant instructions.

CASE STATEMENTS

The “switch” or “case statement is available in a variety of languages; even the Fortran computed and assigned *goto’s* can be regarded as varieties of the switch statement. Our switch-statement is shown :

```
    switch expression
        begin
            case value: statement
            case value: statement
            . . .
            case value: statement
            default: statement
        end
```

Figure: Switch-Statement Syntax

There is a selector expression, which is to be evaluated, followed by n constant values that the expression might take, perhaps including a *default* “value”, which always matches the expression if no other value does. The intended translation of a switch is code to:

1) Evaluate the expression.

2) Find which value in the list of cases is the same as the value of the expression. Recall that the default value matches the expression if none of the values explicitly mentioned in cases does.

3) Execute the statement associated with the value found.

Step (2) in an n-way branch, which can be implemented in one of several ways. If the number of cases is not too great, say 10 at most, then it is reasonable to use a sequence of conditional *goto's*, each of which tests for an individual value and transfers to the code for the corresponding statement.

- A simple loop can be generated by the compiler to compare the value of the expression with each values in the table, if no other match is found the last default entry is sure to match.
- **NOTE:** At the end of the table the value of the expression is paired with the label for the default statements.

METHOD 2:

- If the number of values exceeds 10, it is more efficient to construct a hash table for the values with the labels of the various statements as entries.
- If no entry for the value possessed by the switch expression is found, a jump to the default statement can be generated.

The hash table may appear as follows:

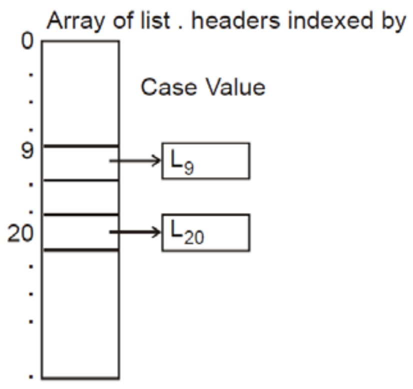


Figure 3.8

Syntax directed translation of case statements

Consider the following switch statement:

switch E

begin

Case V_1 : S_1

```
            Case $V_2$ : $S_2$
            ⋮
            Case $V_{n-1}$ : $S_{n-1}$
            default : $S_n$
        end
```

With syntax directed translation scheme it is convenient to translate the case statement into intermediate code, which follows:

Translation of a Case Statement

```
            Code to evaluate E into t.
            goto test
    $L_1$ :      Code for $S_1$
            goto next
    $L_2$ :      Code for $S_2$
            goto next
            ⋮
    $L_n$ :      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 form of intermediate code the tests all appear at the end, so that a simple code generator generate efficient target code for it.

To translate into the preceding intermediate form, when we see the keyword “switch”, we generate two new labels test and next, and a new temporary ‘*t*. Then as we parse the expression

E, we generate code to evaluate ‘E’ into ‘*t*’. After processing E_1we generate the jump goto test.

Then as we see each case keyword. We create a new label L_i and enter it into the symbol table, along with its location (address) and other possible details if necessary. The symbol table may appear as follows:

Symbol Table

Label	**Address**
L_1	2000
L_1	2500
⋮	
L_n	3000

We maintain a stack in which we place value V_i and the pointer to the symbol table entry L_i.

The stack may be as shown next:

Case Stack

Value	**Pointer to the Symbol Table Entry**
V_1	Pointer to L in symtab
⋮	
V_n	Pointer to L_n in the symtab

Reading the pointer value pairs on the case stack. We can generate a sequence of 3 address statements of the form:

Case	V_1	L_1
Case	V_2	L_2
⋮		
Case	V_{n-1}	L_{n-1}
Case	t	L_n
label	next	

Where '*t*' is the name holding the value of the selector expression 'E' and Ln is the label for the default statement.

NOTE: Case V_i L_i is the synonym for

" if $t = V_i$ goto L_i ."