3
UNIT

Syntax-Directed Translations

CONTENTS

PART-1

Syntax-Directed Translation : Syntax-Directed Translation Schemes, Implementation of Syntax-Directed Translators.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.1. Define syntax directed translation. Construct an annotated parse tree for the expression (4 * 7 + 1) * 2, using the simple desk calculator grammar. **AKTU 2018-19, Marks 07**

Answer

1. Syntax directed definition/translation is a generalization of context free grammar in which each grammar production $X \rightarrow \alpha$ is associated with a set of semantic rules of the form $a := f(b_1, b_2, b_k)$, where a is an attribute obtained from the function f.
2. Syntax directed translation is a kind of abstract specification.
3. It is done for static analysis of the language.
4. It allows subroutines or semantic actions to be attached to the productions of a context free grammar. These subroutines generate intermediate code when called at appropriate time by a parser for that grammar.
5. The syntax directed translation is partitioned into two subsets called the synthesized and inherited attributes of grammar.

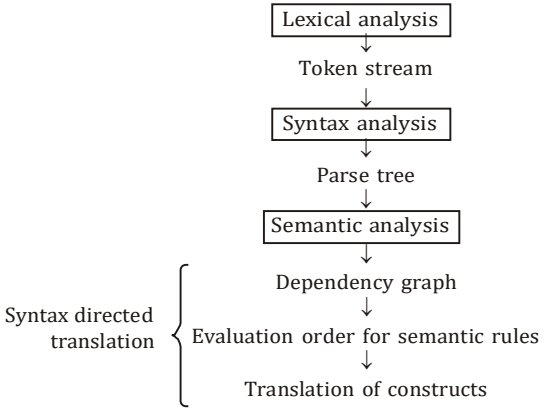


Fig. 3.1.1.

Annotated tree for the expression (4*7 + 1)*2 :

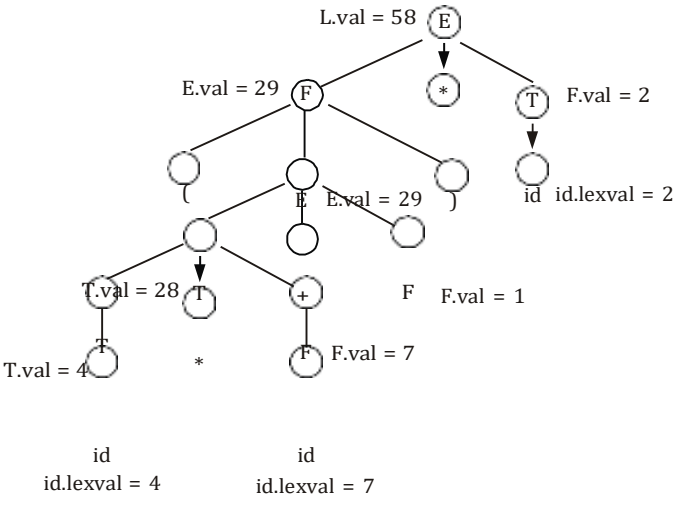
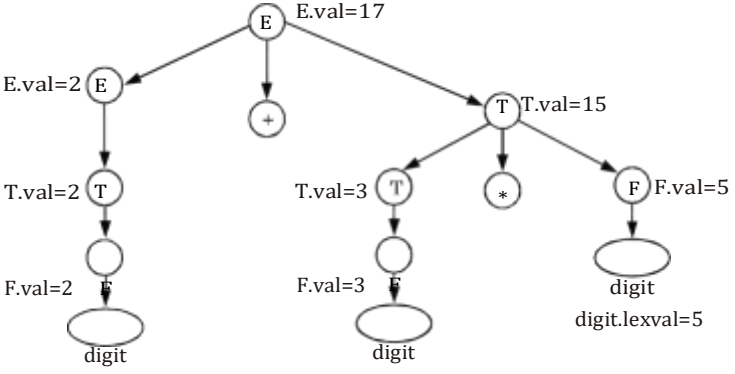


Fig. 3.1.2.

Que 3.2. **What is syntax directed translation ? How are semantic actions attached to the production ? Explain with an example.**

Answer

Syntax directed translation : Refer Q. 3.1, Page 3–2C, Unit-3.
Semantic actions are attached with every node of annotated parse tree.
Example : A parse tree along with the values of the attributes at nodes (called an "annotated parse tree") for an expression 2 + 3*5 with synthesized attributes is shown in the Fig. 3.2.1.



digit.lexval=2 digit.lexval=3

Fig. 3.2.1.

Que 3.3. Explain attributes. What are synthesized and inherited attribute ?

Answer

Attributes :

1. Attributes are associated information with language construct by attaching them to grammar symbols representing that construct.
2. Attributes are associated with the grammar symbols that are the labels of parse tree node.
3. An attribute can represent anything (reasonable) such as string, a number, a type, a memory location, a code fragment etc.
4. The value of an attribute at parse tree node is defined by a semantic rule associated with the production used at that node.

Synthesized attribute :

1. An attribute at a node is said to be synthesized if its value is computed from the attributed values of the children of that node in the parse tree.
2. A syntax directed definition that uses the synthesized attributes is exclusively said to be S-attributed definition.
3. Thus, a parse tree for S-attributed definition can always be annotated by evaluating the semantic rules for the attributes at each node from leaves to root.
4. If the translations are specified using S-attributed definitions, then the semantic rules can be conveniently evaluated by the parser itself during the parsing.

   For example : A parse tree along with the values of the attributes at nodes (called an "annotated parse tree") for an expression 2 + 3*5 with synthesized attributes is shown in the Fig. 3.3.1.

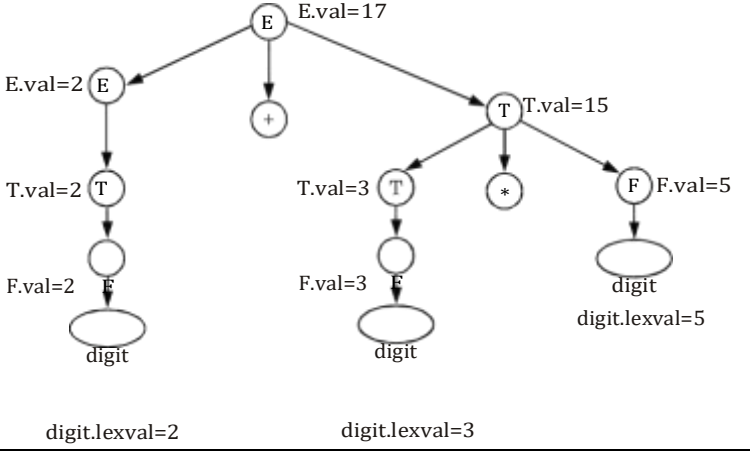


Fig. 3.3.1. An annotated parse tree for expression 2 + 3*5.

Inherited attribute :

1. An inherited attribute is one whose value at a node in a parse tree is defined in terms of attributes at the parent and/or sibling of that node.
2. Inherited attributes are convenient for expressing the dependence of a programming language construct.

   For example : Syntax directed definitions that uses inherited attribute are given as :

$D \rightarrow TL$	L.type : = T.Type
$T \rightarrow$ int	T.type : = integer
$T \rightarrow$ real	T.type : = real
$L \rightarrow L_1, id$	L_1.type : = L.type enter (id.prt, L.type)
$L \rightarrow id$	enter (id.prt, L.type)

The parse tree, along with the attribute values at the parse tree nodes, for an input string int id_1, id_2 and id_3 is shown in the Fig. 3.3.2.

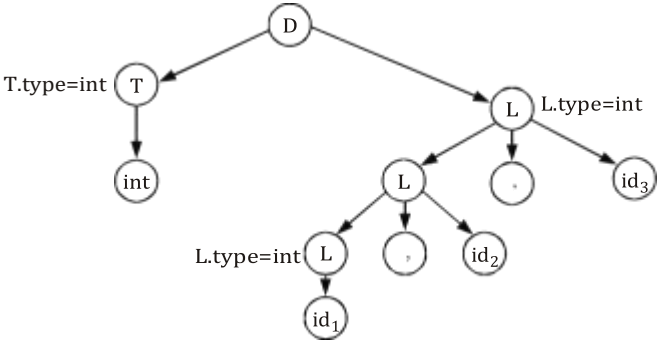


Fig. 3.3.2. Parse tree with inherited attributes for the string int id_1, id_2, id_3.

Que 3.4. What is the difference between S-attributed andL-attributed definitions ?

Answer

S. No.	S-attributed definition	L-attributed definition
1.	It uses synthesized attributes.	It uses synthesized and inherited attributes.
2.	Semantics actions are placed at right end of production.	Semantics actions are placed at anywhere on RHS.
3.	S-attributes can be evaluated during parsing.	L-attributes are evaluated by traversing the parse tree in depth first, left to right.

PART-2

Intermediate Code, Postfix Notation, Parse Trees and Syntax Trees.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.5. What is intermediate code generation and discuss benefits of intermediate code ?

Answer

Intermediate code generation is the fourth phase of compiler which takes parse tree as an input from semantic phase and generates an intermediate code as output.

The benefits of intermediate code are :

1. Intermediate code is machine independent, which makes it easy to retarget the compiler to generate code for newer and different processors.
2. Intermediate code is nearer to the target machine as compared to the source language so it is easier to generate the object code.
3. The intermediate code allows the machine independent optimization of the code by using specialized techniques.
4. Syntax directed translation implements the intermediate code generation, thus by augmenting the parser, it can be folded into the parsing.

Que 3.6. What is postfix translation ? Explain it with suitable example.

Answer

Postfix (reverse polish) translation : It is the type of translation in which the operator symbol is placed after its two operands.

For example :

Consider the expression : (20 + (–5)* 6 + 12)

Postfix for above expression can be calculate as :

(20 + t_1 * 6 + 12)	t_1 = 5 –
20 + t_2 + 12	t_2 = $t_1$6*
t_3 + 12	t_3 = 20 t_2 +
t_4	t_4 = t_3 12 +

Now putting values of t_4, t_3, t_2, t_1

t_4 = t_3 12 +

20 t_2 + 12 +
20 t_1 6 * + 12 +
(20) 5 – 6 * + 12 +

Que 3.7. **Define parse tree. Why parse tree construction is only possible for CFG ?**

Answer

Parse tree : A parse tree is an ordered tree in which left hand side of a production represents a parent node and children nodes are represented by the production's right hand side.

Conditions for constructing a parse tree from a CFG are :

i. Each vertex of the tree must have a label. The label is a non-terminal or terminal or null (ε).

ii. The root of the tree is the start symbol, *i.e.*, S.

iii. The label of the internal vertices is non-terminal symbols $\in V_N$.

iv. If there is a production $A \rightarrow X_1X_2 \ldots\ldots X_K$. Then for a vertex, label A, the children node, will be $X_1X_2 \ldots\ldots X_K$.

v. A vertex n is called a leaf of the parse tree if its label is a terminal symbol $\in \Sigma$ or null (ε).

Parse tree construction is only possible for CFG. This is because the properties of a tree match with the properties of CFG.

Que 3.8. **What is syntax tree ? What are the rules to construct syntax tree for an expression ?**

Answer

1. A syntax tree is a tree that shows the syntactic structure of a program while omitting irrelevant details present in a parse tree.
2. Syntax tree is condensed form of the parse tree.
3. The operator and keyword nodes of a parse tree are moved to their parent and a chain of single production is replaced by single link.

Rules for constructing a syntax tree for an expression :

1. Each node in a syntax tree can be implemented as a record with several fields.
2. In the node for an operator, one field identifies the operator and the remaining field contains pointer to the nodes for the operands.
3. The operator often is called the label of the node.
4. The following functions are used to create the nodes of syntax trees for expressions with binary operators. Each function returns a pointer to newly created node.
   a. **Mknode(op, left, right) :** It creates an operator node with label op and two field containing pointers to left and right.

b. **Mkleaf(*id*, entry) :** It creates an identifier node with label *id* and the field containing entry, a pointer to the symbol table entry for the identifier.

c. **Mkleaf(num, val) :** It creates a number node with label num and a field containing val, the value of the number.

For example : Construct a syntax tree for an expression $a - 4 + c$. In this sequence, $p_1, p_2, \ldots\ldots\ldots\ldots, p_5$ are pointers to nodes, and entry *a* and entry *c* are pointers to the symbol table entries for identifier '*a*' and '*c*' respectively.

p_1 := mkleaf (*id*, entry *a*);
p_2 := mkleaf (num, 4);
p_3 := mknode ('–', p_1, p_2);
p_4 := mkleaf (*id*, entry *c*);
p_5 := mknode ('+', p_3, p_4);

The tree is constructed in bottom-up fashion. The function calls mkleaf (*id*, entry *a*) and mkleaf (num, 4) construct the leaves for *a* and 4. The pointers to these nodes are saved using p_1 and p_2. Call mknode ('–', p_1, p_2) then constructs the interior node with the leaves for *a* and 4as children. The syntax tree will be :

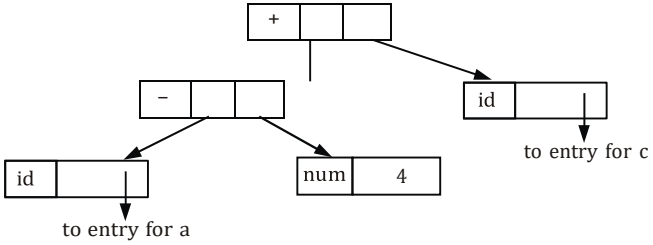


Fig. 3.8.1. The syntax tree for a – 4 + c.

Que 3.9. **Draw syntax tree for the arithmetic expressions : *a* * (*b* + *c*) – *d*/2. Also write the given expression in postfix notation.**

Answer

Syntax tree for given expression : $a * (b + c) - d/2$

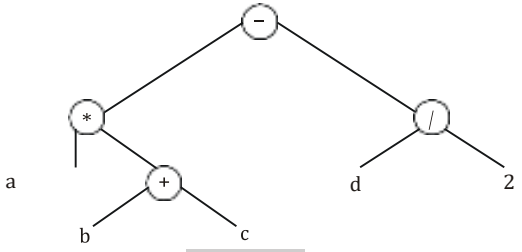


Fig. 3.9.1.

Postfix notation for $a * (b + c) - d/2$

$t_1 = bc +$	$(a * t_1 - d/2)$
$t_2 = a\ t_1\ *$	$(t_2 - d/2)$
$t_3 = d\ 2\ /$	$(t_2 - t_3)$
$t_4 = t_2\ t_3\ -$	(t_4)

Put value of t_1, t_2, t_3

$$\begin{aligned} t_4 &= t_2\ t_3\ - \\ &= t_2\ d\ 2\ /\ - \\ &= at_1\ *\ d\ 2\ /\ - \\ &= abc\ +\ *\ d\ 2\ /\ - \end{aligned}$$

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.10. Explain three address code with examples.

Answer

1. Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields.
2. The general form of three address code representation is :

   $$a := b \text{ op } c$$

   where a, b and c are operands that can be names, constants and op represents the operator.
3. The operator can be fixed or floating point arithmetic operator or logical operators or boolean valued data. Only single operation at right side of the expression is allowed at a time.
4. There are at most three addresses are allowed (two for operands and one for result). Hence, the name of this representation is three address code.

   For example : The three address code for the expression $a = b + c + d$ will be :

   $$t_1 := b + c$$
   $$t_2 := t_1 + d$$
   $$a := t_2$$

   Here t_1 and t_2 are the temporary names generated by the compiler.

Que 3.11. **What are different ways to write three address code ?**

Answer

Different ways to write three address code are :

1. **Quadruple representation :**

   a. The quadruple is a structure with at most four fields such as op, arg1, arg2, result.

   b. The op field is used to represent the internal code for operator, the arg1 and arg2 represent the two operands used and result field is used to store the result of an expression.

   For example : Consider the input statement $x := - a * b + - a * b$

   The three address code is

$t_1 := \text{uminus a}$
$t_2 := t_1 * b$
$t_3 := - a$
$t_4 := t_3 * b$
$t_5 := t_2 + t_4$
$x := t_5$

Op	Arg1	Arg2	Result
uminus	a		t_1
*	t_1	b	t_2
uminus	a		t_3
*	t_3	b	t_4
+	t_2	t_4	t_5
:=	t_5		x

2. **Triples representation :** In the triple representation, the use of temporary variables is avoided by referring the pointers in the symbol table.

   For example : $x := - a * b + - a * b$

   The triple representation is

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	x	(4)

3. **Indirect triples representation :** In the indirect triple representation, the listing of triples is done and listing pointers are used instead of using statement.

   For example : $x = - a * b + - a + b$

   The indirect triples representation is

Number	Op	Arg1	Arg2
(0)	uminus	*a*	
(1)	*	(11)	*b*
(2)	uminus	*a*	
(3)	*	(13)	*b*
(4)	+	(12)	(14)
(5)	:=	*x*	(15)

Location	Statement
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

Three address code of given statement is :

1. If $A > C$ and $B < D$ goto 2
2. If $A = 1$ goto 6
3. If $A <= D$ goto 6
4. $t_1 = A + 2$
5. $A = t_1$
6. $t_2 = C + 1$
7. $C = t_2$

Que 3.12. **Write the quadruples, triple and indirect triple for the following expression :**

$$(x + y) * (y + z) + (x + y + z)$$

AKTU 2018-19, Marks 07

Answer

The three address code for given expression :

$$t_1 := x + y$$
$$t_2 := y + z$$
$$t_3 := t_1 * t_2$$
$$t_4 := t_1 + z$$
$$t_5 := t_3 + t_4$$

i. The quadruple representation :

Location	Operator	Operand 1	Operand 2	Result
(1)	+	x	y	t_1
(2)	+	y	z	t_2
(3)	*	t_1	t_2	t_3
(4)	+	t_1	z	t_4
(5)	+	t_3	t_4	t_5

ii. The triple representation :

Location	Operator	Operand 1	Operand 2
(1)	+	*x*	*y*
(2)	+	*y*	*z*
(3)	*	(1)	(2)
(4)	+	(1)	*z*
(5)	+	(3)	(4)

iii. The indirect triple representation :

Location	Operator	Operand 1	Operand 2
(1)	+	*x*	*y*
(2)	+	*y*	*z*
(3)	*	(11)	(12)
(4)	+	(11)	*z*
(5)	+	(13)	(14)

Location	Statement
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)

Que 3.13. **Generate three address code for the following code :**

```
switch a + b
        {
        case 1 : x = x + 1
        case 2 : y = y + 2
        case 3 : z = z + 3
        default : c = c – 1

        }
```

AKTU 2015-16, Marks 10

Answer

101 : $t_1 = a + b$ goto 103
102 : goto 115
103 : $t = 1$ goto 105
104 : goto 107
105 : $t_2 = x + 1$
106 : $x = t_2$
107 : if $t = 2$ goto 109
108 : goto 111
109 : $t_3 = y + 2$
110 : $y = t_3$
111 : if $t = 3$ goto 113
112 : goto 115
113 : $t_4 = z + 3$
114 : $z = t_4$
115 : $t_5 = c - 1$
116 : $c = t_5$
117 : Next statement

Que 3.14. **Generate three address code for $C[A[i, j]] = B[i, j] + C[A[i, j]] + D[i, j]$ (You can assume any data for solving question, if needed). Assuming that all array elements are integer. Let A and B a 10×20 array with $low_1 = low = 1$.**

AKTU 2017-18, Marks 10

Answer

Given : $low_1 = 1$ and $low = 1, n_1 = 10, n_2 = 20$.

$$B[i, j] = ((i \times n_2) + j) \times w + (base - ((low_1 \times n_2) + low) \times w)$$

$$B[i, j] = ((i \times 20) + j) \times 4 + (base - ((1 \times 20) + 1) \times 4)$$

$$B[i, j] = 4 \times (20\, i + j) + (base - 84)$$

Similarly, $A[i, j] = 4 \times (20\, i + j) + (base - 84)$

and, $D[i, j] = 4 \times (20\, i + j) + (base - 84)$

Hence,
$$C[A[i, j]] = 4 \times (20\, i + j) + (base - 84) + 4 \times (20\, i + j) + (base - 84) + 4 \times (20\, i + j) + (base - 84)$$

$$= 4 \times (20\, i + j) + (base - 84)\,[1 + 1 + 1]$$

$$= 4 \times 3 \times (20\, i + j) + (base - 84) \times 3$$

$$= 12 \times (20\, i + j) + (base - 84) \times 3$$

Therefore, three address code will be

$$t_1 = 20 \times i$$

$$t_2 = t_1 + j$$

$$t_3 = base - 84$$

$$t_4 = 12 \times t_2$$

$$t_5 = t_4 + 3 \times t_3$$

PART-4

Translation of Assignment Statements.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.15. **How would you convert the following into intermediate code ? Give a suitable example.**

i. Assignment statements

ii. Case statements

AKTU 2016-17, Marks 15

Answer

i. Assignment statements :

Production rule	Semantic actions

$S \rightarrow id := E$	{ *id*_entry := look_up(*id*.name);
	if *id*_entry ≠ nil then
	append (*id*_entry ':=' *E*.place)
	else error; /* *id* not declared*/
	}

$E \rightarrow E_1 + E_2$	{ *E*.place := newtemp(); append (*E*.place ':=' E_1.place '+' E_2.place) }
$E \rightarrow E_1 * E_2$	{ *E*.place := newtemp(); append (*E*.place ':=' E_1.place '*' E_2.place) }
$E \rightarrow - E_1$	{ *E*.place := newtemp(); append (*E*.place ':=' 'minus' E_1.place) }
$E \rightarrow (E_1)$	{ *E*. place:= E_1.place }

$E \rightarrow id$	{ *id*_entry: = look_up(*id*.name);
	if *id*_entry ≠ nil then
	append (*id*_entry ':=' *E*.place)
	else error; /* *id* not declared*/
	}

1. The look_up returns the entry for *id*.name in the symbol table if it exists there.
2. The function append is used for appending the three address code to the output file. Otherwise, an error will be reported.
3. Newtemp() is the function used for generating new temporary variables.
4. *E*.place is used to hold the value of *E*.

   Example : $x := (a + b)*(c + d)$

   We will assume all these identifiers are of the same type. Let us have bottom-up parsing method :

Production rule	Semantic action attribute evaluation	Output
$E \rightarrow id$	E.place := a	
$E \rightarrow id$	E.place := b	
$E \rightarrow E_1 + E_2$	E.place := t_1	$t_1 := a + b$
$E \rightarrow id$	E.place := c	
$E \rightarrow id$	E.place := d	
$E \rightarrow E_1 + E_2$	E.place := t_2	$t_2 := c + d$
$E \rightarrow E_1 * E_2$	E.place := t_3	$t_3 := (a + b)*(c + d)$
$S \rightarrow id := E$		$x := t_3$

ii. Case statements :

Production rule	Semantic action

Switch E
{
case $v_1 : s_1$
case $v_2 : s_2$
...
case $v_{n-1} : s_{n-1}$
default : s_n
}
	Evaluate E into t such that $t = E$
	goto check
	L_1 : code for s_1
	goto last
	L_2 : code for s_2
	goto last
	
	L_n : code for s_n
	goto last
	check : 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
	last

switch expression
{
case value : statement
case value : statement
...
case value : statement
default : statement
}

Example :

```
switch(ch)
{
    case 1 : c = a + b;
    break;
    case 2 : c = a – b;
    break;
}
```

The three address code can be

```
if ch = 1 goto L1
if ch = 2 goto L2
L1 : t1 := a + b
     c := t1
     goto last
L2 : t2 := a – b
     c := t2
     goto last
last :
```

Que 3.16. Write down the translation procedure for control statement and switch statement. **AKTU 2018-19, Marks 07**

Answer

1. Boolean expression are used along with if-then, if-then-else, while-do, do-while statement constructs.
2. $S \rightarrow$ If E then $S1$ | If E then $S1$ else $S2$ | while E do $S1$ | do $E1$ while E.
3. All these statements 'E' correspond to a boolean expression evaluation.
4. This expression E should be converted to three address code.
5. This is then integrated in the context of control statement.

Translation procedure for if-then and if-then-else statement :

1. Consider a grammar for if-else

$$S \rightarrow \text{if } E \text{ then } S_1 | \text{if } E \text{ then } S_1 \text{ else } S_2$$

2. Syntax directed translation scheme for if-then is given as follows :

$S \rightarrow$ if E then S_1

E.true := new_label()

E.false := S.next

S_1.next := S.next

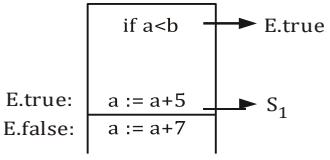
S.code := *E*.code || gen_code(*E*.true ':') || S_1.code

3. In the given translation scheme || is used to concatenate the strings.
4. The function gen_code is used to evaluate the non-quoted arguments passed to it and to concatenate complete string.
5. The *S*.code is the important rule which ultimately generates the three address code.

$S \rightarrow$ if *E* then S_1 else S_2

E.true := new_label()
E.false := new_label()
S_1.next := *S*.next
S_2.next := *S*.next
S.code := *E*.code ‖ gen_code(*E*.true ':') ||
S_1.code := gen_code('goto', *S*.next) ||
gen_code(*E*.false ':') ‖ S_2.code

For example : Consider the statement if $a < b$ then $a = a + 5$ else $a = a + 7$



The three address code for if-else is

100 : if $a < b$ goto 102

101 : goto 103

102 : *L*1 $a := a+5$ /*E.true*/

103 : *L*2 $a := a+7$

Hence, *E*.code is "if $a < b$" *L*1 denotes *E*.true and *L*2 denotes *E*.false is shown by jumping to line 103 (*i.e.*, *S*.next).

Translation procedure for while-do statement :

Production	Semantic rules
$S \rightarrow$ while *E* do *S*1	*S*.begin : = newlable *E*.true : = newlable *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*.being)

Translation procedure for do-while statement :

Production	Semantic rules
$S \rightarrow$ do $S1$ while E	S.begin : = newlable E.true : = S.begin E.false : = S.next S.code = $S1$.code ‖ E.code ‖ gen(E.true ':') ‖ gen('goto' S.being)

PART-5

Boolean Expressions, Statements that alter the Flow of Control.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.17. Define backpatching and semantic rules for boolean expression. Derive the three address code for the following expression : $P < Q$ or $R < S$ and $T < U$.

AKTU 2015-16, Marks 10

OR

Write short notes on backpatching.

Answer

1. Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions during the code generation process.
2. Backpatching refers to the process of resolving forward branches that have been used in the code, when the value of the target becomes known.
3. Backpatching is done to overcome the problem of processing the incomplete information in one pass.
4. Backpatching can be used to generate code for boolean expressions and flow of control statements in one pass.

To generate code using backpatching following functions are used :

1. **Makelist(i) :** Makelist is a function which creates a new list from one item where i is an index into the array of instructions.
2. **Merge(p_1, p_2) :** Merge is a function which concatenates the lists pointed by p_1 and p_2, and returns a pointer to the concatenated list.

3. **Backpatch(*p*, *i*) :** Inserts *i* as the target label for each of the instructions on the list pointed by *p*.

Backpatching in boolean expressions :

1. The solution is to generate a sequence of branching statements where the addresses of the jumps are temporarily left unspecified.
2. For each boolean expression *E* we maintain two lists :
   a. *E*.truelist which is the list of the (addresses of the) jump statements appearing in the translation of *E* and forwarding to *E*.true.
   b. *E*.falselist which is the list of the (addresses of the) jump statements appearing in the translation of *E* and forwarding to *E*.false.
3. When the label *E*.true (resp. *E*.false) is eventually defined we can walk down the list, patching in the value of its address.
4. In the translation scheme below :
   a. We use emit to generate code that contains place holders to be filled in later by the backpatch procedure.
   b. The attributes *E*.truelist, *E*.falselist are synthesized.
   c. When the code for *E* is generated, addresses of jumps corresponding to the values true and false are left unspecified and put on the lists *E*.truelist and *E*.falselist, respectively.
5. A marker non-terminal *M* is used to capture the numerical address of a statement.
6. nextinstr is a global variable that stores the number of the next statement to be generated.

   The grammar is as follows :

   $B \rightarrow B_1 \mid MB_2 \mid B_1$ AND $MB_2 \mid !B_1 \mid (B_1) \mid E_1$ rel $E_2 \mid$ True $\mid$ False

   $M \rightarrow \varepsilon$

   The translation scheme is as follows :

   i. $B \rightarrow B_1 \mid\mid MB_2$ {backpatch (B_1.falselist, *M*.instr);
   B.truelist = merge (B_1.truelist, B_2.truelist);
   B.falselist = B_2.falselist;}

   ii. $B \rightarrow B_1$AND MB_2
   {backpatch (B_1.truelist, *M*.instr);
   B.truelist = B_2.truelist;
   B.falselist = merge (B_1.falselist, B_2.falselist);}

   iii. $B \rightarrow !B_1$ {*B*.truelist = B_1.falselist;
   B.falselist = B_1.truelist;}

iv. $B \rightarrow (B_1)\{B.\text{truelist} = B_1.\text{truelist};$
$B.\text{falselist} = B_1.\text{falselist};\}$

v. $B \rightarrow E_1 \text{ rel } E_2\{B.\text{truelist} = \text{makelist (nextinstr)};$
$B.\text{falselist} = \text{makelist (nextinstr + 1)};$
append ('if' E_1.addr relop E_2.addr 'goto_');
append ('goto_');}

vi. $B \rightarrow$ true$\{B.\text{truelist} = \text{makelist (nextinstr)};$
append ('goto_');}

vii. $B \rightarrow$ false$\{B.\text{falselist} = \text{makelist (nextinstr)};$
append ('goto_');}

viii. $M \rightarrow \varepsilon\{M.\text{instr} = \text{nextinstr};\}$

Three address code :

100 : if P < Q goto_

101 : goto 102

102 : if R < S goto 104

103 : goto_

104 : if T < U goto_

105 : goto_

Que 3.18. Explain translation scheme for boolean expression.

Answer

Translation scheme for boolean expression can be understand by following example.

Consider the boolean expression generated by the following grammar :

$E \rightarrow E$ OR E

$E \rightarrow E$ AND E

$E \rightarrow$ NOT E

$E \rightarrow (E)$

$E \rightarrow id$ relop id

$E \rightarrow$ TRUE

$E \rightarrow$ FALSE

Here the relop is denoted by $\leq, \geq, \neq, <, >$. The OR and AND are left associate. The highest precedence is NOT then AND and lastly OR.
The translation scheme for boolean expressions having numerical representation is as given below :

Production rule	Semantic rule

$E \rightarrow E_1$ OR E_2	{
	 E.place := newtemp()
	 append(E.place ':=' E_1.place 'OR' E_2.place)
	 }

$E \rightarrow E_1$ AND E_2	{
	 E.place := newtemp()
	 append(E.place ':=' E_1.place 'AND' E_2.place)
	 }

$E \rightarrow$ NOT E_1	{
	 E.place := newtemp()
	 append(E.place ':=' 'NOT' E_1.place)
	 }

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

$E \rightarrow id_1$ relop id_2	{
	 E.place := newtemp()
	 append('if' id_1.place relop.op id_2.place 'goto' next_state + 3);
	 append(E.place ':=' '0');
	 append('goto' next_state +2);
	 append(E.place := '1')
	 }

$E \rightarrow$ TRUE	{
	 E.place := newtemp();
	 append(E.place ':=' '1')
	 }

$E \rightarrow$ FALSE	{
	 E.place := newtemp()
	 append(E.place ':=' 0)
	 }

PART-6

Postfix Translation : Array References in Arithmetic Expressions.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.19. Write a short note on postfix translation.

Answer

1. In a production $A \rightarrow \alpha$, the translation rule of A.CODE consists of the concatenation of the CODE translations of the non-terminals in α in the same order as the non-terminals appear in α.
2. Production can be factored to achieve postfix form.

Postfix translation of while statement :

Production : $S \rightarrow$ while $M1$ E do $M2$ $S1$

Can be factored as :

1. $S \rightarrow C\ S1$
2. $C \rightarrow W\ E$ do
3. $W \rightarrow$ while

A suitable transition scheme is given as :

Production rule	Semantic action
$W \rightarrow$ while	W.QUAD = NEXTQUAD
$C \rightarrow W\ E$ do	C W E do
$S \rightarrow C\ S1$	BACKPATCH (S1.NEXT, C.QUAD) S.NEXT = C.FALSE GEN (goto C.QUAD)

Que 3.20. What is postfix notations ? Translate $(C + D)^*(E + Y)$ into postfix using Syntax Directed Translation Scheme (*SDTS*).

AKTU 2017-18, Marks 10

Answer

Postfix notation : Refer Q. 3.6, Page 3–6C, Unit-3.

Numerical : Syntax directed translation scheme to specify the translation of an expression into postfix notation are as follow :

Production :

$$E \rightarrow E_1 + T$$
$$E_1 \rightarrow T$$
$$T \rightarrow T_1 \times F$$
$$T_1 \rightarrow F$$
$$F \rightarrow (E)$$
$$F \rightarrow id$$

Schemes :

$$E.\text{code} = E_1.\text{code} \parallel T_1\text{code} \parallel \text{'+'}$$

$$E_1.\text{code} = T.\text{code}$$

$$T_1.\text{code} = T_1.\text{code} \parallel F.\text{code} \parallel \text{'×'}$$

$$T_1.\text{code} = F.\text{code}$$

$$F.\text{code} = E.\text{code}$$

$$F.\text{code} = id.\text{code}$$

where '||' sign is used for concatenation.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.21. **Explain procedure call with example.**

Answer

Procedures call :

1. Procedure is an important and frequently used programming construct for a compiler.
2. It is used to generate code for procedure calls and returns.
3. Queue is used to store the list of parameters in the procedure call.
4. The translation for a call includes a sequence of actions taken on entry and exit from each procedure. Following actions take place in a calling sequence :
   a. When a procedure call occurs then space is allocated for activation record.
   b. Evaluate the argument of the called procedure.
   c. Establish the environment pointers to enable the called procedure to access data in enclosing blocks.
   d. Save the state of the calling procedure so that it can resume execution after the call.
   e. Also save the return address. It is the address of the location to which the called routine must transfer after it is finished.
   f. Finally generate a jump to the beginning of the code for the called procedure.

For example : Let us consider a grammar for a simple procedure call statement :

1. $S \rightarrow$ call *id*(Elist)
2. Elist $\rightarrow$ Elist, *E*
3. Elist $\rightarrow$ *E*

A suitable transition scheme for procedure call would be :

Production rule	Semantic action
$S \rightarrow$ call *id*(Elist)	for each item *p* on QUEUE do GEN (param *p*) GEN (call *id*.PLACE)
Elist $\rightarrow$ Elist, *E*	append E.PLACE to the end of QUEUE
Elist $\rightarrow$ *E*	initialize QUEUE to contain only *E*.PLACE

Que 3.22. Explain the concept of array references in arithmetic expressions.

Answer

1. An array is a collection of elements of similar data type. Here, we assume the static allocation of array, whose subscripts ranges from one to some limit known at compile time.
2. If width of each array element is '*w*' then the i^{th} element of array A begins in location,

$$\text{base} + (i - \text{low}) * d$$

where low is the lower bound on the subscript and base is the relative address of the storage allocated for an array *i.e.*, base is the relative address of A[low].

3. A two dimensional array is normally stored in one of two forms, either row-major (row by row) or column-major (column by column).
4. The Fig. 3.22.1 for row-major and column-major are given as :

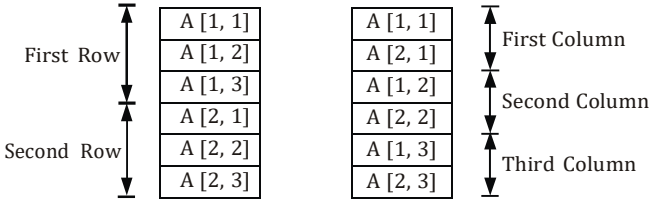


Fig. 3.22.1.

5. In case of a two dimensional array stored in row-major form, the relative address of $A[i_1, i_2]$ can be calculated by formula,

$$(\text{base} + (i_1 - \text{low}_1) * n_2 + i_2 - \text{low}_2)* w$$

where low_1 and low_2 are lower bounds on the values of i_1 and i_2 and n_2 is the number of values that i_2 can take.

6. That is, if high_2 is the upper bound on the value of i_2 then $n_2 = [\text{high}_2 - \text{low}_2 + 1]$.

7. Assuming that i_1 and i_2 are only values that are not known at compile time, we can rewrite above expression as :

$$((i_1 * n_2) + i_2)*w + (\text{base} - ((\text{low}_1 * n_2) + \text{low}_2) * w)$$

8. The generalize form of row-major will be,

$$((...((i_1 n_2 + i_2)\ n_3 + i_3)...)\ n_k + I_i)*w + \text{base} - ((...((\text{low}_1\ n_2 + \text{low}_2)n_3 + \text{low}_3)...)\ n_k + \text{low}_k) * w$$

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.23. Explain declarative statements with example.

Answer

In the declarative statements the data items along with their data types are declared.

For example :

$S \rightarrow D$	{offset:= 0}
$D \rightarrow id : T$	{enter_tab(*id*.name, *T*.type,offset); offset:= offset + *T*.width)}
$T \rightarrow$ integer	{*T*.type:= integer; *T*.width:= 8}
$T \rightarrow$ real	{*T*.type:= real; *T*.width:= 8}
$T \rightarrow$ array[num] of T_1	{*T*.type:= array(num.val, T_1.type) *T*.width:= num.val × T_1.width}
$T \rightarrow *T_1$	{*T*.type:= pointer(*T*.type) *T*.width:= 4}

1. Initially, the value of offset is set to zero. The computation of offset can be done by using the formula offset = offset + width.
2. In the above translation scheme, *T*.type, *T*.width are the synthesized attributes. The type indicates the data type of corresponding identifier and width is used to indicate the memory units associated with an identifier of corresponding type. For instance integer has width 4 and real has 8.
3. The rule $D \rightarrow id : T$ is a declarative statements for *id* declaration. The enter_tab is a function used for creating the symbol table entry for identifier along with its type and offset.
4. The width of array is obtained by multiplying the width of each element by number of elements in the array.
5. The width of pointer types of supposed to be 4.

VERY IMPORTANT QUESTIONS

Following questions are very important. These questions may be asked in your SESSIONALS as well as UNIVERSITY EXAMINATION.

Q. 1. Define syntax directed translation. Construct an annotated parse tree for the expression (4 * 7 + 1) * 2, using the simple desk calculator grammar.

Ans. Refer Q. 3.1.

Q. 2. Explain attributes. What are synthesized and inherited attribute ?

Ans. Refer Q. 3.3.

Q. 3. What is postfix translation ? Explain it with suitable example.

Ans. Refer Q. 3.3.

Q. 4. What is syntax tree ? What are the rules to construct syntax tree for an expression ?

Ans. Refer Q. 3.8.

Q. 5. What are different ways to write three address code ?

Ans. Refer Q. 3.11.

Q. 6. Write the quadruples, triple and indirect triple for the following expression :
$(x + y) * (y + z) + (x + y + z)$

Ans. Refer Q. 3.12.

Q. 7. How would you convert the following into intermediatecode ? Give a suitable example.

i. **Assignment statements**

ii. **Case statements**

Ans. Refer Q. 3.15.

Q. 8. Define backpatching and semantic rules for boolean expression. Derive the three address code for the followingexpression : $P < Q$ or $R < S$ and $T < U$.

Ans. Refer Q. 3.17.

☺☺☺