3 UNIT

Syntax Directed Translation

CONTENTS

Part-1 :	Syntax-Directed Translation : Syntax-Directed Translation Schemes Implementation of Syntax-Directed Translators	3–2B to 3–6B
Part-2 :	Intermediate Code	3–6B to 3–7B
Part-3 :	Postfix Notation Parse Trees and Syntax Trees	3–7B to 3–9B
Part-4 :	Three Address Code Quadruples and Triples	3–9B to 3–14B
Part-5 :	Translation of Assignment Statements	3–14B to 3–17B
Part-6 :	Boolean Expressions Statements that alter the Flow of Control	3–17B to 3–20B
Part-7 :	Postfix Translation : Array References in Arithmetic Expressions	3–20B to 3–22B
Part-8 :	Procedures Call	3–22B to 3–24B
Part-9 :	Declarations Statements	3–24B to 3–26B

PART-1

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

CONCEPT OUTLINE

- Syntax directed definition/translation is a generalization of context free grammar.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Udit

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 2013-14, Marks 10**

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 scheme is useful because it enables the compiler designer to express the generation of intermediate code directly in terms of the syntactic structure of the source language.
6. The syntax directed translation is partitioned into two subsets called the synthesized and inherited attributes of grammar.

Scanned with CamScan

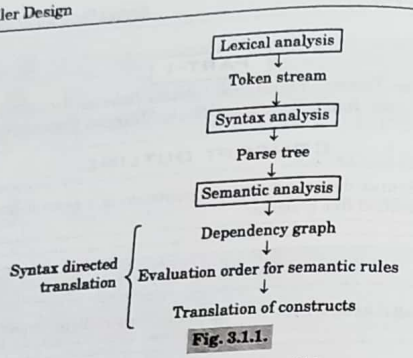


Fig. 3.1.1.

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

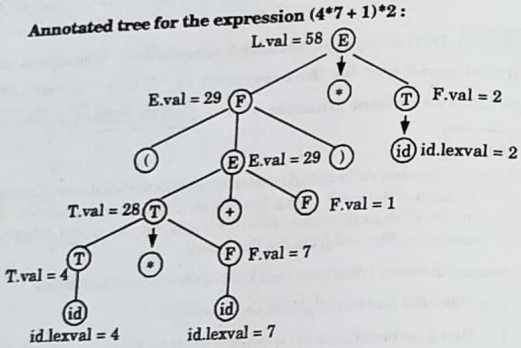


Fig. 3.2.1.

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

AKTU 2014-15, Marks 10

Answer

Syntax directed translation : *Refer Q. 3.1, Page 3–2B, 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.*

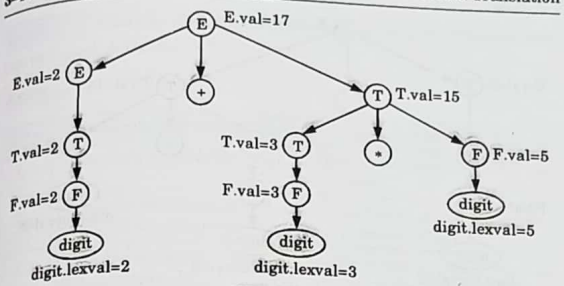


Fig. 3.2.1.

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.

Scanned with CamScan

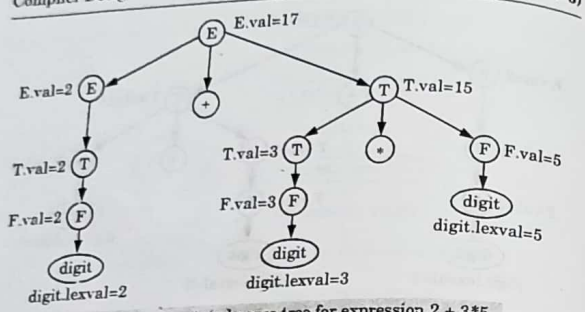


Fig. 3.3.1. An annotated parse tree for expression 2 + 3*5.

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.

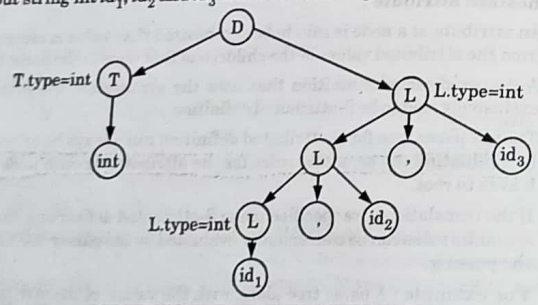


Fig. 3.3.2. Parse tree with inherited attributes for the string int id_1, id_2, id_3.

Que 3.4. **What is the difference between S-attributed and L-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.

CONCEPT OUTLINE

- Three address code is sequence of statements of the form $X = Y$ op Z where X and Y are operands and op is operator.
- A syntax tree is condensed form of parse tree.

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.

Scanned with CamScan

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.

PART-3

Postfix Notation, Parse Trees and Syntax Trees.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.6. **What is postfix translation ? Explain it with suitable example.** **AKTU 2014-15, Marks 10**

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 :

$$(20 + t_1 * 6 + 12) \qquad t_1 = 5 -$$
$$20 + t_2 + 12 \qquad t_2 = t_1 6^*$$
$$t_3 + 12 \qquad t_3 = 20\ t_2 +$$
$$t_4 \qquad t_4 = t_3\ 12 +$$

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 $A \rightarrow X_1 X_2 \ldots X_K$. Then for a vertex, label A, the children node, will be $X_1 X_2 \ldots X_K$.
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, $p_1, p_2, \ldots\ldots\ldots, p_5$ 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);

Scanned with CamScan

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. The call mknode ('–', p_1, p_2) then constructs the interior node with the leaves for *a* and 4 as children. The syntax tree will be :

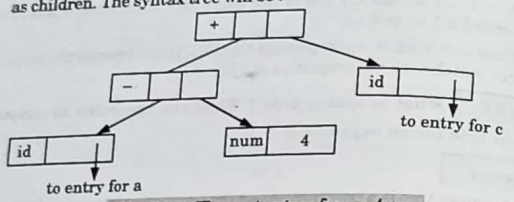


Fig. 3.8.1. The syntax tree for a – 4 + c.

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 : $a * (b + c) - d/2$

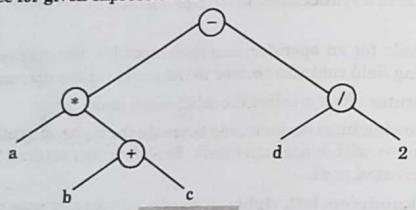


Fig. 3.9.1.

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

$$\begin{aligned} t_4 &= t_2\, t_3 - \\ &= t_2\, d\, 2 / - \\ &= a t_1 * d\, 2 / - \\ &= abc + * d\, 2 / - \end{aligned}$$

PART-4

Three Address Code, Quadruples and Triples.

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 atmost 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 :

$$\begin{aligned} t_1 &:= b + c \\ t_2 &:= t_1 + d \\ a &:= t_2 \end{aligned}$$

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 ? Write the three address code for the following code segment :**
while *A* < *C* and *B* < *D* do
if *A* = 1 then *C* = *C* + 1
else while *A* < = *D* do *A* = *A* + 2. **AKTU 2013-14, Marks 10**

Answer

Different ways to write three address code are :

1. **Quadruple representation :**
   a. The quadruple is a structure with atmost 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.

Scanned with CamScan

For example : Consider the input statement $x := -a * b + -a * b$
The three address code is

t_1 := 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. **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 3.13. **Generate three address code for the following code segment :**
while ($a < b$) do
If ($c < d$) then $x = y + z$

AKTU 2014-15, Marks 10

Answer

L_1 : if $a < b$ goto L_2
goto L_4
L_2 : if $c < d$ goto L_3
L_3 : $t_1 := y + z$
$x := t_1$

Scanned with CamScan

goto L_3
L_4 : Next

Que 3.14. **Describe various representation of three address codes. Translates the expression : – (a + b) * (c + d) + (a + b + c).**

AKTU 2014-15, Marks 10

Answer

Three address code representation : Refer Q. 3.10, Page 3–10B, Unit-3.

Numerical : The three address code for given expression :

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

i. The quadruple representation :

Location	Operator	Operand 1	Operand 2	Result
(1)	+	a	b	t_1
(2)	+	c	d	t_2
(3)	*	t_1	t_2	t_3
(4)	+	t_1	c	t_4
(5)	+	t_3	t_4	t_5

ii. The triple representation :

Location	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	+	c	d
(3)	*	(1)	(2)
(4)	+	(1)	c
(5)	+	(3)	(4)

iii. The indirect triple representation :

	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	+	c	d
(3)	*	(11)	(12)
(4)	+	(11)	c
(5)	+	(13)	(14)

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

Que 3.15. **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, $n_1 = 10$, $n_2 = 20$.

$$B[i, j] = ((i \times n_2) + j) \times w + (\text{base} - ((\text{low}_1 \times n_2) + \text{low}) \times w)$$

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

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

Similarly,

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

and,

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

Hence,

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

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

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

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

Therefore, three address code will be

$$t_1 = 20 \times i$$
$$t_2 = t_1 + j$$
$$t_3 = \text{base} - 84$$
$$t_4 = 12 \times t_2$$
$$t_5 = t_4 + 3 \times t_3$$

PART-5

Translation of Assignment Statements.

CONCEPT OUTLINE

- Assignment statements can be converted into semantic action using syntax directed translation scheme.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 3.16. **How would you convert the following into intermediate code ? Give a suitable example.**

i. Assignment statements

ii. Case statements

AKTU 2016-17, Marks 15

Scanned with CamScan

Answer

i. **Assignment statements :**

Production rule	Semantic actions

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

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

$E \to id$	{ *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
$E \to id$	E.place := a	
$E \to id$	E.place := b	
$E \to E_1 + E_2$	E.place := t_1	$t_1 := a + b$
$E \to id$	E.place := c	
$E \to id$	E.place := d	
$E \to E_1 + E_2$	Eplace := t_2	$t_2 := c + d$
$E \to E_1 * E_2$	E.place := t_3	$t_3 := (a + b)*(c + d)$
$S \to id := E$		$x := t_3$

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
}

Scanned with CamScan

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 L_1
if $ch = 2$ goto L_2
L_1 : $t_1 := a + b$
$c := t_1$
goto last
L_2 : $t_2 := a - b$
$c := t_2$
goto last
last :

PART-6

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 2013-14, 2015-16; Marks 10

OR

Write short notes on backpatching. **AKTU 2014-15, Marks 05**

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 :

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

$M \rightarrow \varepsilon$

The translation scheme is as follows :

i. $B \rightarrow B_1 \parallel MB_2$ {backpatch (B_1.falselist, M.instr);

Scanned with CamScan

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.truelist = B_1.truelist;
B.falselist = B_1.falselist;}

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

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

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

viii. $M \rightarrow \varepsilon${$M$.instr = 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 ≤, ≥, ≠, <, >. 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-7

Postfix Translation : Array References in Arithmetic Expressions.

Scanned with CamScan

CONCEPT OUTLINE

- Production can be factored to achieve postfix form.

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 3–7B, 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.

PART-8

Procedures Call.

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.

Scanned with CamScan

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 ith element of array A begins in location,

$$base + (i - low) * d$$

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 :

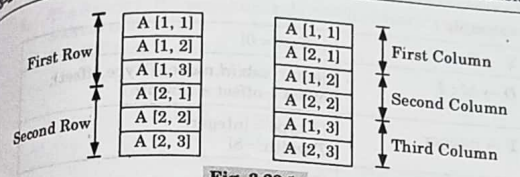


Fig. 3.22.1.

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,

$$(base + (i_1 - low_1) * n_2 + i_2 - low_2) * w$$

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 $n_2 = [high_2 - low_2 + 1]$.
7. Assuming that i_1 and i_2 are only values that are not known at compile time, we can rewrite above expression as :

$$((i_1 * n_2) + i_2) * w + (base - ((low_1 * n_2) + low_2) * w)$$

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

$$((...((i_1 n_2 + i_2)\, n_3 + i_3)...)\, n_k + I_i) * w + base - ((...((low_1\, n_2 + low_2) n_3 + low_3)...)\, n_k + low_k) * w$$

PART-9

Declarations Statements.

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.

Scanned with CamScan

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.

Que 3.24. **Give the syntax directed definition for if-else statement.**

Answer

1. Consider a grammar for if-else

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

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

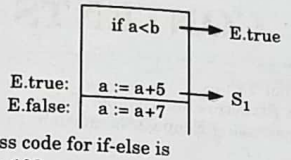
S_2.next := S.next

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 : $L1$ $a := a+5$ /*E.true*/

103 : $L2$ $a := a+7$

Hence, E.code is "if $a < b$" $L1$ denotes E.true and $L2$ denotes E.false is shown by jumping to line 103 (*i.e.*, S.next).

☺☺☺

Scanned with CamScan

4 UNIT

Symbol Tables

CONTENTS

Part-1 : Symbol Tables : ..... **4–2B to 4–9B**
Data Structure for Symbol Tables
Representing Scope Information

Part-2 : Run-Time Administration : ..... **4–9B to 4–13B**
Implementation of Simple Stack
Allocation Scheme

Part-3 : Storage Allocation in ..... **4–13B to 4–14B**
Block Structured Language

Part-4 : Error Detection and Recovery : ..... **4–15B to 4–19B**
Lexical Phase Errors
Syntactic Phase Errors
Semantic Errors

Doubt to hard

PART-1

Symbol Tables : Data Structure for Symbol Tables, Representing Scope Information.

CONCEPT OUTLINE

- Symbol table is a compile time data structure that is used by compiler to collect information about source program.
- Scope information characterizes the declaration of identifiers.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.1. **Discuss symbol table with its capabilities ?**

Answer

1. A symbol table is a data structure used by a compiler to keep track of scope, life and binding information about names which are used in the source program to identify the various program elements, like variables, constants, procedures, and the labels of statements.
2. A symbol table must have the following capabilities :
   a. **Lookup :** To determine whether a given name is in the table.
   b. **Insert :** To add a new name (a new entry) to the table.
   c. **Access :** To access the information related with the given name.
   d. **Modify :** To add new information about a known name.
   e. **Delete :** To delete a name or group of names from the table.

Que 4.2. **What are the symbol table requirements ? What are the demerits in the uniform structure of symbol table ?**

Answer

The **basic requirements of a symbol table are as follows :**

1. **Structural flexibility :** Based on the usage of identifier, the symbol table entries must contain all the necessary information.
2. **Fast lookup/search :** The table lookup/search depends on the implementation of the symbol table and the speed of the search should be as fast as possible.
3. **Efficient utilization of space :** The symbol table must be able to grow or shrink dynamically for an efficient usage of space.

Scanned with CamScan

4. **Ability to handle language characteristics :** The characteristic of a language such as scoping and implicit declaration needs to be handled.

Demerits in uniform structure of symbol table :

1. The uniform structure cannot handle a name whose length exceed upper bound or limit or name field.
2. If the length of a name is small, then the remaining space is wasted.

Que 4.3. How names can be looked up in the symbol table ? Discuss. **AKTU 2016-17, Marks 10**

Answer

1. The symbol table is searched (looked up) every time a name is encountered in the source text.
2. When a new name or new information about an existing name is discovered, the content of the symbol table changes.
3. Therefore, a symbol table must have an efficient mechanism for accessing the information held in the table as well as for adding new entries to the symbol table.
4. In any case, the symbol table is a useful abstraction to aid the compiler to ascertain and verify the semantics, or meaning of a piece of code.
5. It makes the compiler more efficient, since the file does not need to be re-parsed to discover previously processed information.

For example : Consider the following outline of a C function :

```
void scopes ( )
  {
  int a, b, c;                /* level 1 */
  .......
        {
      int a, b;               /* level 2 */
      ....
      }
      {
      float c, d;             /* level 3 */
        {
        int m;                /* level 4 */
        .....
        }
      }
  }
```

The symbol table could be represented by an upwards growing stack as :

i. Initially the symbol table is empty.

ii. After the first three declarations, the symbol table will be

c	int
b	int
a	int

iii. After the second declaration of Level 2.

b	int
a	int
c	int
b	int
a	int

iv. As the control come out from Level 2.

c	int
b	int
a	int

v. When control will enter into Level 3.

d	float
c	float
c	int
b	int
a	int

vi. After entering into Level 4.

m	int
d	float
c	float
c	int
b	int
a	int

vii. On leaving the control from Level 4.

d	float
c	float
c	int
b	int
a	int

viii. On leaving the control from Level 3.

c	int
b	int
a	int

Scanned with CamScan

ix. On leaving the function entirely, the symbol table will be again empty.

Que 4.4. **What is the role of symbol table ? Discuss different data structures used for symbol table.** **AKTU 2013-14, Marks 10**

OR

Discuss the various data structures used for symbol table with suitable example. **AKTU 2014-15, Marks 10**

Answer

Role of symbol table :

1. It keeps the track of semantics of variables.
2. It stores information about scope.
3. It helps to achieve compile time efficiency.

Different data structure used in implementing symbol table are as :

1. **Unordered list :**
   a. Simple to implement symbol table.
   b. It is implemented as an array or a linked list.
   c. Linked list can grow dynamically that eliminate the problem of a fixed size array.
   d. Insertion of variable take O(1) time , but lookup is slow for large tables *i.e.*, $O(n)$.
2. **Ordered list :**
   a. If an array is sorted, it can be searched using binary search in $O(\log_2 n)$.
   b. Insertion into a sorted array is expensive that it takes $O(n)$ time on average.
   c. Ordered list is useful when set of names is known *i.e.*, table of reserved words.
3. **Search tree :**
   a. Search tree operation and lookup is done in logarithmic time.
   b. Search tree is balanced by using algorithm of AVL and Red-black tree.
4. **Hash tables and hash functions :**
   a. Hash table translate the elements in the fixed range of value called hash value and this value is used by hash function.

   b. Hash table can be used to minimize the movement of elements in the symbol table.
   c. The hash function helps in uniform distribution of names in symbol table.

Example : Consider a part of C program

```
int x, y;
msg ();
```

1. Unordered list :

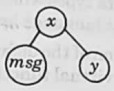
S. No.	Name	Type
1	x	int
2	msg	function
3	y	int

2. Ordered list :

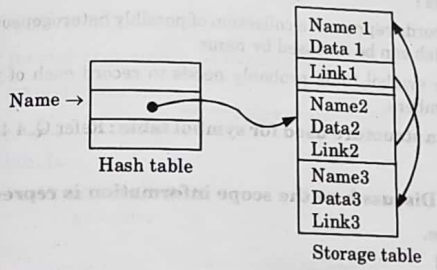
Id	Name	Type
Id1	x	int
Id2	y	int
Id3	msg	function

Id
Id1
Id2
Id3

3. Search tree :



4. Hash table :



Que 4.5. **Describe symbol table and its entries. Also, discuss various data structure used for symbol table.** **AKTU 2015-16, Marks 10**

Answer

Symbol table : Refer Q. 4.1, Page 4–2B, Unit-4.

Scanned with CamScan

Entries in the symbol table are as follows :

1. **Variables :**
   a. Variables are identifiers whose value may change between executions and during a single execution of a program.
   b. They represent the contents of some memory location.
   c. The symbol table needs to record both the variable name as well as its allocated storage space at runtime.
2. **Constants :**
   a. Constants are identifiers that represent a fixed value that can never be changed.
   b. Unlike variables or procedures, no runtime location needs to be stored for constants.
   c. These are typically placed right into the code stream by the compiler at compilation time.
3. **Types (user defined) :**
   a. A user defined type is combination of one or more existing types.
   b. Types are accessed by name and reference a type definition structure.
4. **Classes :**
   a. Classes are abstract data types which restrict access to its members and provide convenient language level polymorphism.
   b. This includes the location of the default constructor and destructor, and the address of the virtual function table.
5. **Records :**
   a. Records represent a collection of possibly heterogeneous members which can be accessed by name.
   b. The symbol table probably needs to record each of the record's members.

Various data structure used for symbol table : Refer Q. 4.4, Page 4–5B, Unit-4.

Que 4.6. Discuss how the scope information is represented in a symbol table.

Answer

1. Scope information characterizes the declaration of identifiers and the portions of the program where it is allowed to use each identifier.
2. Different languages have different scopes for declarations. For example, in FORTRAN, the scope of a name is a single subroutine, whereas in ALGOL, the scope of a name is the section or procedure in which it is declared.

3. Thus, the same identifier may be declared several times as distinct names, with different attributes, and with different intended storage locations.
4. The symbol table is thus responsible for keeping different declaration of the same identifier distinct.
5. To make distinction among the declarations, a unique number is assigned to each program element that in return may have its own local data.
6. Semantic rules associated with productions that can recognize the beginning and ending of a subprogram are used to compute the number of currently active subprograms.

There are mainly two semantic rules regarding the scope of an identifier :

a. Each identifier can only be used within its scope.
b. Two or more identifiers with same name and are of same kind cannot be declared within the same lexical scope.

The scope declaration of variables, functions, labels and objects within a program is shown below :

Scope of variables in statement blocks :

```
{ int x ;
  ...
  { int y ;
    ...
  }
  ...
}
```

Scope of variable x

Scope of variable y

Scope of formal arguments of functions :

```
int mul (int n) {
  ...
}
```

Scope of argument n

Scope of labels :

```
void jumper () {
  ... goto sim;
  ...
  sim++ ;
  ... goto sim;
  ...
}
```

Scope of label sim

7. **Scope in class declaration (scope of declaration) :** The portion of the program in which a declaration can be applied is called the scope of that declaration. In a procedure, a name is said to be local to the procedure if it is in the scope of declaration within the procedure, otherwise the name is said to be non-local.

Scanned with CamScan

Scope of object fields and methods :

```
class X {
  public :
    void A ()
    {
      m = 1;
    }
  private :
    int m;
    ...
}
```

← Scope of variable m and method A

PART-2

Run-Time Administration : Implementation of Simple Stack Allocation Scheme.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.7. **Write a short note on activation record.**

AKTU 2014-15, Marks 05

Answer

Activation record is a data structure that contains the important state information for a particular instance of a function call.

Activation record contains the following :

1. The values of actual parameters.
2. The count of number of arguments.
3. The return address.
4. The return value.
5. The value of SP (Stack Pointer) for the activation record.

Local data
Old SP
Return value
Return address
Arg count
Actual parameters

Fig. 4.7.1. Pictorial representation of activation record.

Que 4.8. **How to sub-divide a run-time memory into code and data areas. Explain.**

AKTU 2016-17, Marks 10

Answer

Sub-division of run-time memory into codes and data areas is shown in Fig. 4.8.1.

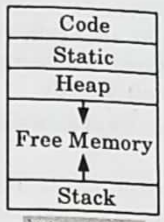


Fig. 4.8.1.

1. **Code :** It stores the executable target code which is of fixed size and do not change during compilation.
2. **Static allocation :**
   a. The static allocation is for all the data objects at compile time.
   b. The size of the data objects is known at compile time.
   c. The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.
   d. In static allocation, the compiler can determine amount of storage required by each data object. Therefore, it becomes easy for a compiler to find the address of these data in the activation record.
   e. At compile time, compiler can fill the addresses at which the target code can find the data on which it operates.
3. **Heap allocation :** There are two methods used for heap management :
   a. **Garbage collection method :**
      i. When all access path to a object are destroyed but data object continue to exist, such type of objects are said to be garbaged.
      ii. The garbage collection is a technique which is used to reuse that object space.
      iii. In garbage collection, all the elements whose garbage collection bit is 'on' are garbaged and returned to the free space list.
   b. **Reference counter :**
      i. Reference counter attempt to reclaim each element of heap storage immediately after it can no longer be accessed.
      ii. Each memory cell on the heap has a reference counter associated with it that contains a count of number of values that point to it.

Scanned with CamScanner

PART-4

Error Detection and Recovery : Lexical Phase Errors, Syntactic Phase Errors, Semantic Errors.

CONCEPT OUTLINE

- Errors are detected during different phases of compiler.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 4.14. Define error recovery. What are the properties of error message ? Discuss the goals of error handling.

Answer

Error recovery : Error recovery is an important feature of any compiler, through which compiler can read and execute the complete program even it have some errors.

Properties of error message are as follows :

1. Message should report the errors in original source program rather than in terms of some internal representation of source program.
2. Error message should not be complicated.
3. Error message should be very specific and should fix the errors at correct positions.
4. There should be no duplicacy of error messages, i.e., same error should not be reported again and again.

Goals of error handling are as follows :

1. Detect the presence of errors and produce "meaningful" diagnostics.
2. To recover quickly enough to be able to detect subsequent errors.
3. Error handling components should not significantly slow down the compilation of syntactically correct programs.

Que 4.15. What are lexical phase errors, syntactic phase errors and semantic phase errors ? Explain with suitable example.

AKTU 2013-14, 2015-16; Marks 10

Answer

1. **Lexical phase error :**
   a. A lexical phase error is a sequence of character that does not match the pattern of token *i.e.*, while scanning the source program, the compiler may not generate a valid token from the source program.
   b. Reasons due to which errors are found in lexical phase are :
      i. The addition of an extraneous character.
      ii. The removal of character that should be presented.
      iii. The replacement of a character with an incorrect character.
      iv. The transposition of two characters.

   For example :
      i. In Fortran, an identifier with more than 7 characters long is a lexical error.
      ii. In Pascal program, the character ~, & and @ if occurred is a lexical error.

2. **Syntactic phase errors (syntax error) :**
   a. Syntactic errors are those errors which occur due to the mistake done by the programmer during coding process.
   b. Reason due to which errors are found in syntactic phase are :
      i. Missing of semicolon
      ii. Unbalanced parenthesis and punctuation

   For example : Let us consider the following piece of code :

```
int x;
int y //Syntax error
```

   In example, syntactic error occurred because of absence of semicolon.

3. **Semantic phase errors :**
   a. Semantic phase errors are those errors which occur in declaration and scope in a program.
   b. Reason due to which errors are found :
      i. Undeclared names

Scanned with CamScanner

ii. Type incompatibilities

iii. Mismatching of actual arguments with the formal arguments.

For example : Let us consider the following piece of code :

scanf("%f%f", *a*, *b*);

In example, *a* and *b* are semantic error because scanf uses address of the variables as &a and &b.

4. **Logical errors :**

a. Logical errors are the logical mistakes founded in the program which is not handled by the compiler.

b. In these types of errors, program is syntactically correct but does not operate as desired.

For example :

Let consider following piece of code :

$x = 4;$

$y = 5;$

average $= x + y/2$

The given code do not give the average of x and y because BODMAS property is not used properly.

Que 4.16. What do you understand by lexical error and syntactic error ? Also, suggest methods for recovery of errors.

AKTU 2014-15, Marks 10

OR

Explain logical phase error and syntactic phase error. Also suggest methods for recovery of error. **AKTU 2017-18, Marks 10**

Answer

Lexical and syntactic error : Refer Q. 4.15, Page 4–15B, Unit-4.

Various error recovery methods are :

1. **Panic mode recovery :**

a. This is the simplest method to implement and used by most of the parsing methods.

b. When parser detect an error, the parser discards the input symbols one at a time until one of the designated set of synchronizing token is found.

Panic mode correction often skips a considerable amount of input without checking it for additional errors. It gives guarantee not to go in infinite loop.

For example :

Let consider a piece of code :

$a = b + c$

$d = e + f;$

By using panic mode it skips $a = b + c$ without checking the error in the code.

2. **Phrase-level recovery :**

a. When parser detects an error, the parser may perform local correction on remaining input.

b. It may replace a prefix of the remaining input by some string that allows parser to continue.

c. A typical local correction would replace a comma by a semicolon, delete an extraneous semicolon or insert a missing semicolon.

For example :

Let consider a piece of code

while $(x > 0)$ $y = a + b;$

In this code local correction is done by phrase-level recovery by adding 'do' and parsing is continued.

3. **Error production :** If error production is used by the parser, we can generate appropriate error message and parsing is continued.

For example :

Let consider a grammar

Scanned with CamScan

$E \rightarrow +E \mid -E \mid *A \mid /A$

$A \rightarrow E$

When error production encounters $*A$, it sends an error message to the user asking to use '*' as unary or not.

4. **Global correction :**

a. Global correction is a theoretical concept.

b. This method increases time and space requirement during parsing.

☺☺☺

5 UNIT

Code Generation

CONTENTS

Part-1 :	Code Generation : Design Issues	5–2B to 5–3B
Part-2 :	The Target Language Address in Target Code	5–3B to 5–4B
Part-3 :	Basic Blocks and Flow Graphs Optimization of Basic Blocks Code Generator	5–4B to 5–10B
Part-4 :	Machine Independent Optimizations Loop Optimization	5–10B to 5–15B
Part-5 :	DAG Representation of Basic Blocks	5–15B to 5–21B
Part-6 :	Value Numbers and Algebraic Laws Global Data Flow Analysis	5–21B to 5–23B

Scanned with CamScan

PART-1

Code Generation : Design Issues.

CONCEPT OUTLINE

- Code generation take intermediate code as input and produce target program as output.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.1. **What is code generation ? Discuss the design issues of code generation.**

Answer

1. Code generation is the final phase of compiler.
2. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program as shown in Fig. 5.1.1.

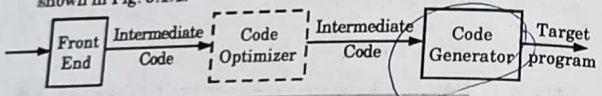


Fig. 5.1.1. Position of code generator.

Design issues of code generator are :

1. **Input to the code generator :**
   a. The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.
   b. The many choices for the IR include three address representations such as quadruples, triples, indirect triples and graphical representations such as syntax trees and DAGs.
2. **The target program :**
   a. The instruction set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high quality machine code.

   b. The most common target machine architectures are RISC (Reduced Instruction Set Computer), CISC (Complex Instruction Set Computer), and stack based.
3. **Instruction selection :**
   a. The code generator must map the IR program into a code sequence that can be executed by the target machine.
   b. If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates.
4. **Register allocation :**
   a. A key problem in code generation is deciding what values to hold in which registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values.
   b. Values that are not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.
   c. The use of registers is often subdivided into two subproblems :
      i. Register allocation, during which we select the set of variables that will reside in registers at each point in the program.
      ii. Register assignment, during which we pick the specific register that a variable will reside in.
5. **Evaluation order :**
   a. The order in which computations are performed can affect the efficiency of the target code.
   b. Some computation orders require fewer registers to hold intermediate results than others.

PART-2

The Target Language, Address in Target Code.

CONCEPT OUTLINE

- Target language is a language which is generated after the compilation of source program.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Scanned with CamScan

Que 5.2. **Discuss addresses in the target code.**

Answer

1. Addresses in the target code show how names in the IR can be converted into addresses in the target code by looking at code generation for simple procedure calls and returns using static and stack allocation.
2. Addresses in the target code represent executing program runs in its own logical address space that was partitioned into four code and data areas :
   a. A statically determined area code that holds the executable target code. The size of the target code can be determined at compile time.
   b. A statically determined data area static for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.
   c. A dynamically managed area heap for holding data objects that are allocated and freed during program execution. The size of the heap cannot be determined at compile time.
   d. A dynamically managed area stack for holding activation records as they are created and destroyed during procedure calls and returns. Like the heap, the size of the stack cannot be determined at compile time.

PART-3

Basic Blocks and Flow Graphs, Optimization of Basic Blocks, Code Generator.

CONCEPT OUTLINE

- Basic block is a sequence of consecutive statements in which flow of controls enters at start and leaves at the end without half.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.3. **Write an algorithm to partition a sequence of three address statements into basic blocks.** **AKTU 2016-17, Marks 10**

Answer

The algorithm for construction of basic block is as follows :

Input : A sequence of three address statements.

Output : A list of basic blocks with each three address statements in exactly one block.

Method :

1. We first determine the set of leaders, the first statement of basic block. The rules we use are given as :
   a. The first statement is a leader.
   b. Any statement which is the target of a conditional or unconditional goto is a leader.
   c. Any statement which immediately follows a conditional goto is a leader.
2. For each leader construct its basic block, which consist of leader and all statements up to the end of program but not including the next leader. Any statement not placed in the block can never be executed and may now be removed, if desired.

Que 5.4. **Explain flow graph with example.**

Answer

1. A flow graph is a directed graph in which the flow control information is added to the basic blocks.
2. The nodes to the flow graph are represented by basic blocks.
3. The block whose leader is the first statement is called initial blocks.
4. There is a directed edge from block B_{i-1} to block B_i if B_i immediately follows B_{i-1} in the given sequence. We can say that B_{i-1} is a predecessor of B_i.

For example : Consider the three address code as

1. prod := 0	2. $i := 1$
3. $t_1 := 4 * i$	4. $t_2 := a[t_1]$ /* computation of $a[i]$ */
5. $t_3 := 4 * i$	6. $t_4 := b[t_3]$ /* computation of $b[i]$ */
7. $t_5 := t_2 * t_4$	8. $t_6 := \text{prod} + t_5$
9. prod := t_6	10. $t_7 := i + 1$
11. $i := t_7$	12. if $i <= 10$ goto (3)

The flow graph for the given code can be drawn as follows :

Scanned with CamScan

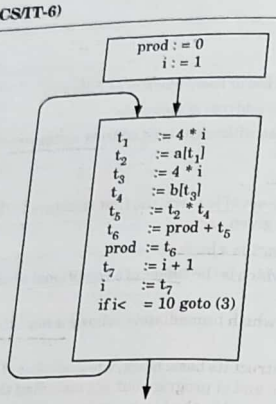


Fig. 5.4.1. Flow graph.

Que 5.5. **What is loop ? Explain what constitute a loop in a flow graph.**

Answer

Loop is a collection of nodes in the flow graph such that :

1. *All such nodes are strongly connected. That means there is always a path from any node to any other node within that loop.*
2. *The collection of nodes has unique entry. That means there is only one path from a node outside the loop to the node inside the loop.*
3. *The loop that contains no other loop is called inner loop.*

Following term constitute a loop in flow graph :

1. ***Dominators :***
   a. *In control flow graphs, a node d dominates a node n if every path from the entry node to n must go through d. This is denoted as d dom n.*
   b. *By definition, every node dominates itself.*
   c. *There are a number of related concepts :*
      i. *A node d strictly dominates a node n if d dominates n and d is not equal to n.*
      ii. *The immediate dominator or idom of a node n is the unique node that strictly dominates n but does not strictly dominate any other node that strictly dominates n. Every node, except the entry node, has an immediate dominator.*

Udit K. Gupta

      iii. The dominance frontier of a node d is the set of all nodes n such that d dominates an immediate predecessor of n, but d does not strictly dominate n. It is the set of nodes where d's dominance stops.
      iv. A dominator tree is a tree where each node's children are those nodes it immediately dominates. Because the immediate dominator is unique, it is a tree. The start node is the root of the tree.
2. **Natural loops :**
   a. Loop in a flow graph can be denoted by $n \rightarrow d$ such that d dom n.
   b. These edges are called back edges and for a loop there can be more than one back edge.
   c. If there is $p \rightarrow q$ then q is a head and p is a tail and head dominates tail.
   d. The natural loop can be defined by a back edge $n \rightarrow d$ such that there exists a collection of all the nodes that can reach to n without going through d and at the same time d can also be added to this collection.
3. **Pre-header :**
   a. The pre-header is a new block created such that successor of this block is the header block.
   b. All the computations that can be made before the header block can be made before the pre-header block.

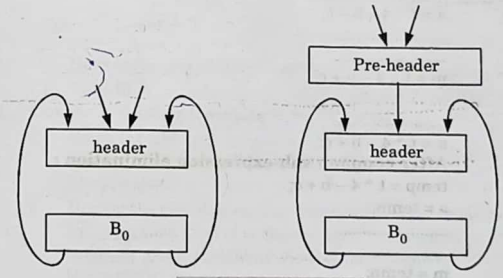


Fig. 5.5.1. Pre-header.

4. **Reducible flow graph :**
   a. The reducible graph is a flow graph in which there are two types of edges, forward edges and backward edges.
   b. These edges have following properties :
      i. The forward edge forms an acyclic graph.
      ii. The backward edges are such edges whose head dominates their tail.

Scanned with CamScanner

c. The program structure in which there is exclusive use of if-then, while-do or goto statements generates a flow graph which is always reducible.

Que 5.6. **Discuss in detail the process of optimization of basic blocks. Give an example.** **AKTU 2016-17, Marks 10**

OR

What are different issues in code optimization ? Explain it with proper example. **AKTU 2013-14, Marks 10**

Answer

Different issues in code optimization are :

1. **Function preserving transformation :** The function preserving transformations are basically divided into following types :

a. **Common sub-expression elimination :**

i. A common sub-expression is nothing but the expression which is already computed and the same expression is used again and again in the program.

ii. If the result of the expression not changed then we eliminate computation of same expression again and again.

For example :

Before common sub-expression elimination :

a = t * 4 – b + c;

.....

.....

m = t * 4 – b + c;

.....

.....

n = t * 4 – b + c;

After common sub-expression elimination :

temp = t * 4 – b + c;

a = temp;

.....

.....

m = temp;

.....

.....

n = temp;

iii. In given example, the equation $a = t * 4 - b + c$ is occurred most of the times. So it is eliminated by storing the equation into temp variable.

b. **Dead code elimination :**

i. Dead code means the code which can be emitted from program and still there will be no change in result.

ii. A variable is live only when it is used in the program again and again. Otherwise, it is declared as dead, because we cannot use that variable in the program so it is useless.

iii. The dead code occurred during the program is not introduced intentionally by the programmer.

For example :

Define False = 0

!False = 1

If(!False)

{

.....

.....

}

i=0
if(i==1)
{
a=x+r
}

iv. If false becomes zero, is guaranteed then code in 'IF' statement will never be executed. So, there is no need to generate or write code for this statement because it is dead code.

Variable

c. **Copy propagation :**

i. Copy propagation is the concept where we can copy the result of common sub-expression and use it in the program.

ii. In this technique the value of variable is replaced and computation of an expression is done at the compilation time.

For example :

$pi = 3.14;$

var. $\Rightarrow r = 5;$

Area = $pi * r * r$;

Here at the compilation time the value of pi is replaced by 3.14 and r by 5.

d. **Constant folding (compile time evaluation) :** ⇒ Constant propogation π=3.14 A=3.14×r×r.

i. Constant folding is defined as replacement of the value of one constant in an expression by equivalent constant value at the compile time.

ii. During the compilation, execution efficiency can be improved by using such kind of certain actions *i.e.*, replacement of one constant by an equivalent constant can improve efficiency of the compiler.

iii. In constant folding all operands in an operation are constant. Original evaluation can also be replaced by result which is also a constant.

For example : $a = 3.14157/2$ can be replaced by $a = 1.570785$ thereby eliminating a division operation.

2. **Algebraic simplification :**

a. Peephole optimization is an effective technique for algebraic simplification.

b. The statements such as

Scanned with CamScan

$x := x + 0$

or $x := x * 1$

can be eliminated by peephole optimization.

Que 5.7. **Write a short note on transformation of basic blocks.**

Answer

Transformation :

1. A number of transformations can be applied to basic block without changing set of expression computed by the block.
2. Transformation helps us in improving quality of code and act as optimizer.
3. There are two important classes as local transformation that can be applied to the basic block :

   a. **Structure preserving transformation :** They are as follows :

      i. **Common sub-expression elimination :** Refer Q. 5.6, Page 5–8B, Unit-5.

      ii. **Dead code elimination :** Refer Q. 5.6, Page 5–8B, Unit-5.

      iii. **Interchange of statement :** Suppose we have a block with the two adjacent statements,

      $temp1 = a + b$

      $temp2 = m + n$

      Then we can interchange the two statements without affecting the value of the block if and only if neither 'm' nor 'n' is temporary variable temp1 and neither 'a' nor 'b' is temporary variable temp2. From the given statements we can conclude that a normal form basic block allow us for interchanging all the statements if they are possible.

   b. **Algebraic transformation :** Refer Q. 5.6, Page 5–8B, Unit-5.

PART-4

Machine Independent Optimizations, Loop Optimization.

CONCEPT OUTLINE

- *Code optimization is used by compiler to improve the efficiency of generated object code.*
- *Loop optimization is done to improve the efficiency of program.*

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.8. **What is code optimization ? Discuss the classification of code optimization.**

Answer

Code optimization :

1. The code optimization refers to the techniques used by the compiler to improve the execution efficiency of generated object code.
2. It involves a complex analysis of intermediate code and performs various transformations but every optimizing transformation must also preserve the semantic of the program.

Classification of code optimization :

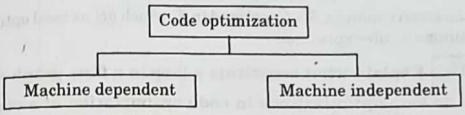


Fig. 5.8.1. Classification of code optimization.

1. **Machine dependent :** The machine dependent optimization can be achieved using following criteria :
   a. Allocation of sufficient number of resources to improve the execution efficiency of the program.
   b. Using immediate instructions wherever necessary.
   c. The use of intermix instructions along with the data increases the speed of execution.
2. **Machine independent :** The machine independent optimization can be achieved using following criteria :
   a. The code should be analyzed completely and use alternative equivalent sequence of source code that will produce a minimum amount of target code.
   b. Use appropriate program structure in order to improve the efficiency of target code.
   c. By eliminating the unreachable code from the source program.
   d. Move two or more identical computations at one place and make use of the result instead of each time computing the expressions.

Que 5.9. **Explain local optimization.**

Answer

1. Local optimization is a kind of optimization in which both the analysis and the transformations are localized to a basic block.

Scanned with CamScanner

2. The transformations in local optimization are called as local transformations.
3. The name of transformation is usually prefixed with 'local' while referring to the local transformation.
4. There are "local" transformations that can be applied to program to attempt an improvement.

For example : The elimination of common sub-expression, provided A is not an alias for B or C, the assignments :

$$A := B + C + D$$
$$E := B + C + F$$

might be evaluated as

$$T_1 := B + C$$
$$A := T_1 + D$$
$$E := T_1 + F$$

In the given example, $B + C$ is stored in T_1 which act as local optimization of common sub-expression.

Que 5.10. **Explain what constitute a loop in a flow graph and how will you do loop optimizations in code optimization of a compiler.**

AKTU 2014-15, Marks 10

OR

Write a short note on loop optimization.

AKTU 2017-18, Marks 05

Answer

Following term constitute a loop in flow graph : Refer Q. 5.5, Page 5–6B, Unit-5.

Loop optimization is a process of increasing execution time and reducing the overhead associated with loops.

The loop optimization is carried out by following methods :

1. **Code motion :**
   a. *Code motion is a technique which moves the code outside the loop.*
   b. *If some expression in the loop whose result remains unchanged even after executing the loop for several times, then such an expression should be placed just before the loop (i.e., outside the loop).*
   c. *Code motion is done to reduce the execution time of the program.*
2. **Induction variables :**
   a. *A variable x is called an induction variable of loop L if the value of variable gets changed every time.*
   b. *It is either decremented or incremented by some constant.*
3. **Reduction in strength :**
   a. *In strength reduction technique the higher strength operators can be replaced by lower strength operators.*

   b. *The strength of certain operator is higher than other.*
   c. *The strength reduction is not applied to the floating point expressions because it may yield different results.*
4. **Loop invariant method :** In loop invariant method, the computation inside the loop is avoided and thereby the computation overhead on compiler is avoided.
5. **Loop unrolling :** In this method, the number of jumps and tests can be reduced by writing the code two times.

For example :

```
int i = 1;
while(i<=100)
{
    a[i]=b[i];
    i++;
}
```

Can be written as →

```
int i = 1;
while(i<=100)
{
    a[i]=b[i];
    i++;
    a[i]=b[i];
    i++ ;
}
```

6. **Loop fusion or loop jamming :** In loop fusion method, several loops are merged to one loop.

For example :

```
for i:=1 to n do
for j:=1 to m do
a[i,j]:=10
```

Can be written as →

```
for i:=1 to n*m do
a[i]:=10
```

Que 5.11. **Consider the following sequence of three address codes :**

1. **Prod : = 0**
2. I **: = 1**
3. T_1 **: = 4*I**
4. T_2 **: = addr (A) – 4**
5. T_3 **: = T_2 [T_1]**
6. T_4 **: = addr (B) – 4**
7. T_5 **: = T_4 [T_1]**
8. T_6 **: = T_3*T_5**
9. **Prod : = Prod + T_6**
10. $I = I + 1$
11. **If I <= 20 goto (3)**

Perform loop optimization. **AKTU 2015-16, Marks 10**

OR

Consider the following three address code segments :

***PROD* : = 0**

***I* : = 1**

Scanned with CamScan

*T1 : = 4*I*
T2 : = addr(A) – 4
T3 : = T2 [T1]
T4 : = addr(B) – 4
T5 : = T4[T1]
*T6 : = T3*T5*
PROD : = PROD + T6
I : = I + 1
If i < = 20 goto (3)

a. Find the basic blocks and flow graph of above sequence.
b. Optimize the code sequence by applying function preserving transformation optimization technique.

AKTU 2017-18, Marks 10

Answer

1. As first statement of program is leader statement.
   ∴ PROD = 0 is a leader.
2. Fragmented code represented by two blocks is shown below :

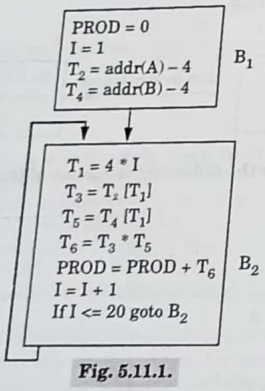


Fig. 5.11.1.

3. A variable *I* is called an induction variable of loop *L* if every time the variable *I* changes values.
4. Generally induction variables are

$$a := I*b \text{ or } a := I \pm b$$

where *b* is constant. In Fig. 5.11.2, *I* and T_1 are induction variables.

5. The values of *I* varies from 1 to 20 and value T_1 varies from (4, 8, ... , 80).

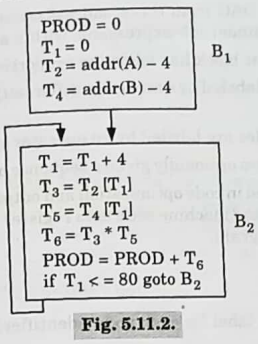


Fig. 5.11.2.

PART-5

DAG Representation of Basic Blocks.

CONCEPT OUTLINE

- DAG stands for Directed Acyclic Graphs.
- DAG is used for implementation of basic blocks.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.12. **What is DAG ? How DAG is created from three address code ? Write algorithm for it and explain it with a relevant example.**

AKTU 2013-14, Marks 10

OR

Write a short note on direct acyclic graph.

AKTU 2017-18, Marks 05

Answer

DAG :

1. The abbreviation DAG stands for Directed Acyclic Graph.
2. DAGs are useful data structure for implementing transformations on basic blocks.
3. A DAG gives picture of how the value computed by each statement in the basic block is used in the subsequent statement of the block.

Scanned with CamScan

4. Constructing a DAG from three address statement is a good way of determining common sub-expressions within a block.
5. A DAG for a basic block has following properties :
   a. Leaves are labeled by unique identifier, either a variable name or constants.
   b. Interior nodes are labeled by an operator symbol.
   c. Nodes are also optionally given a sequence of identifiers for labels.
6. Since, DAG is used in code optimization and output of code optimization is machine code and machine code uses register to store variable used in the source program.

Algorithm :

Input : A basic block.

Output : A DAG with label for each node (identifier).

Method :

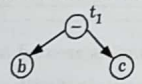
1. Create nodes with one or two left and right children.
2. Create linked list of attached identifiers for each node.
3. Maintain all identifiers for which a node is associated.
4. Node (identifier) represents value that identifier has the current point in DAG construction process. Symbol table store the value of node (identifier).
5. If there is an expression of the form $x = y$ op z then DAG contain "op" as a parent node and node(y) as a left child and node(z) as a right child.

For example :

Given expression : $a * (b - c) + (b - c) * d$

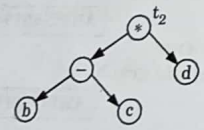
The construction of DAG with three address code will be as follows :

Step 1 :



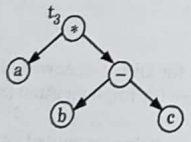
$t_1 = b - c$

Step 2 :



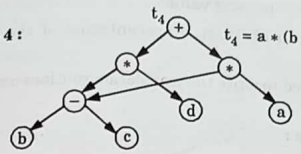
$t_2 = (b - c) * d$

Step 3 :



$t_3 = a * (b - c)$

Step 4 :



$t_4 = a * (b - c) + (b - c) * d$

Que 5.13. **What is DAG ? What are its advantages in context of optimization ?**

AKTU 2014-15, Marks 10

Answer

DAG : Refer Q. 5.12, Page 5–15B, Unit-5.

Advantage of DAG :

1. We automatically detect common sub-expressions with the help of DAG algorithm.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values which could be used outside the block.

Que 5.14. **How DAG is different from syntax tree ? Construct the DAG for the following basic blocks :**

$$a := b + c$$
$$b := b - d$$
$$c := c + d$$
$$e = b + c$$

Also, explain the key application of DAG.

AKTU 2015-16, Marks 15

Answer

DAG v/s Syntax tree :

1. Directed Acyclic Graph is used for transformations on the basic block. To apply the transformations on a basic block a DAG is constructed from three address statement.
2. A DAG can be constructed for the following types of label on nodes :
   a. Leaf nodes labeled as identifiers operator.

Scanned with CamScan

b. Interior nodes store operator values.

3. While syntax tree is an abstract representation of the language constructs.

4. The syntax trees are used to write the translation routines using syntax directed definitions.

DAG for the given code is :

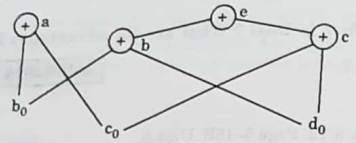


Fig. 5.14.1.

1. The two occurrences of sub-expressions $b + c$ compute the same value.
2. Value computed by a and e are same.

Applications of DAG :

1. **Scheduling :** Directed acyclic graphs representations of partial orderings have many applications in scheduling for systems of tasks.
2. **Data processing networks :** A directed acyclic graph may be used to represent a network of processing elements. In this representation, data enters a processing element through its incoming edges and leaves the element through its outgoing edges.
3. **Data compression :** Directed acyclic graphs may also be used as a compact representation of a collection of sequences. In this type of application, one finds a DAG in which the paths form the sequences.
4. It helps in finding statement that can be recorded.

Que 5.15. Define a directed acyclic graph. Construct a DAG and write the sequence of instructions for the expression :

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

AKTU 2016-17, Marks 15

Answer

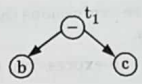
Directed acyclic graph : *Refer Q. 5.12, Page 5–15B, Unit-5.*

Numerical :

Given expression : $a + a * (b - c) + (b - c) * d$

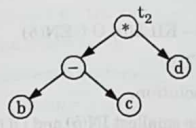
The construction of DAG with three address code will be as follows :

Step 1 :



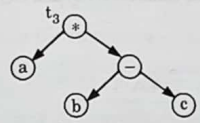
$t_1 = b - c$

Step 2 :



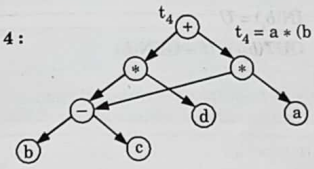
$t_2 = (b - c) * d$

Step 3 :



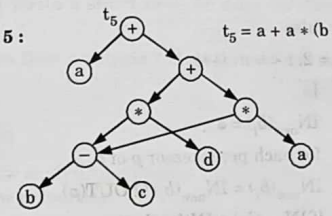
$t_3 = a * (b - c)$

Step 4 :



$t_4 = a * (b - c) + (b - c) * d$

Step 5 :



$t_5 = a + a * (b - c) + (b - c) * d$

Que 5.16. Give the algorithm for the elimination of local and global common sub-expressions algorithm with the help of example.

AKTU 2017-18, Marks 10

Answer

Algorithm for elimination of local common sub-expression : DAG algorithm is used to eliminate local common sub-expression.

DAG : Refer Q. 5.12, Page 5–15B, Unit-5.

Scanned with CamScan

Algorithm for elimination of global common sub-expression :

a. Global common sub-expressions are expressions that compute the same value but in different basic blocks.

b. To eliminate the global common sub-expression, we need to compute the following equations :

$$OUT(b) = IN(b) - KILL(b) \text{ O } GEN(b)$$

$$IN(b) = \text{O } OUT(p)$$

Here, we obtain the smallest solution.

c. The algorithm for computing the smallest IN(b) and OUT(b) is as follows :

1. $IN(b_1) = \phi$
   $OUT(b_1) = GEN(b_1)$;
2. For $(i = 2; i <= n; i++)$
   {
   $IN(b_i) = U$
   $OUT(b_i) = U - GEN(b)$
   }
3. flag = true
4. While (flag) do
   {
   flag = false
   for $(i = 2; i <= n; i++)$
   {
   $IN_{new}(b_i) = \phi$
   for each predecessor p of b_i
   $IN_{new}(b_i) = IN_{new}(b_i)$ O $OUT(p)$
   if $IN_{new}(b_i) \neq IN(b_i)$ then
   {
   flag = true
   $IN(b_i) = In_{new}(b_i)$
   $OUT(b_i) = IN(b_i) - KILL(b_i)$ O $GEN(b_i)$
   }
   }
   }

d. *After computing IN(b) and OUT(b), eliminating the global common sub-expressions is done as follows :*

i. For every statement s of the form $x = y \; op \; z$ such that $y \; op \; z$ is available at the beginning of the block containing s, and neither y nor z is defined prior to the statement $x = y \; op \; z$ in that block, do :

1. Find all definitions reaching up to the s statement block that have $y \; op \; z$ on the right.
2. Create a new temp.
3. Replace each statement $U = y \; op \; z$ found in step 1 by :

   temp = $y \; op \; z$

   U = temp
4. Replace the statement $x = y \; op \; z$ in block by x = temp.

PART-6

Value Numbers and Algebraic Laws, Global Data Flow Analysis.

Questions-Answers

Long Answer Type and Medium Answer Type Questions

Que 5.17. **Write a short note on data flow analysis.**

OR

What is data flow analysis ? How does it use in code optimization ?

AKTU 2014-15, Marks 10

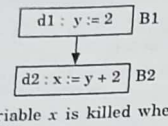
Answer

1. Data flow analysis is a process in which the values are computed using data flow properties.
2. In this analysis, the analysis is made on data flow.
3. A program's Control Flow Graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.
4. The information gathered is often used by compilers when optimizing a program.
5. A simple way to perform data flow analysis of programs is to set up data flow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes, *i.e.*, it reaches a fix point.
6. Reaching definitions is used by data flow analysis in code optimization.

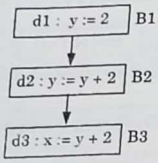
Scanned with CamScan

Reaching definitions :

1. A definition D reaches at point p if there is a path from D to p along which D is not killed.



2. A definition D of variable x is killed when there is a redefinition of x.



3. The definition $d1$ is said to a reaching definition for block $B2$. But the definition $d1$ is not a reaching definition in block $B3$, because it is killed by definition $d2$ in block $B2$.

Que 5.18. **Write short notes (any two) :**

i. **Global data flow analysis**
ii. **Loop unrolling**
iii. **Loop jamming**

AKTU 2013-14, Marks 10

AKTU 2015-16, Marks 15

OR

Write short note on global data analysis.

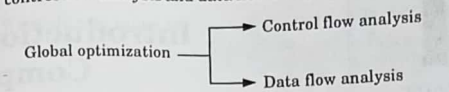
AKTU 2017-18, Marks 05

Answer

i. **Global data flow analysis :**

1. The local optimization has a very restricted scope on the other hand the global optimization over a broad scope such as procedure or function body.
2. For a global optimization a program is represented in the form of program flow graph. The program flow graph is a graphical representation in which each node represents the basic block and edges represent the flow of control from one block to another.

3. There are two types of analysis performed for global optimizations : control flow analysis and data flow analysis.



ii. **Loop unrolling :** Refer Q. 5.10, Page 5–12B, Unit-5.
iii. **Loop fusion or loop jamming :** Refer Q. 5.10, Page 5–12B, Unit-5.

Que 5.19. **Discuss the role of macros in programming language.**

AKTU 2014-15, Marks 05

Answer

Role of macros in programming language are :

1. It is use to define word that are used most of the time in program.
2. It automates complex task.
3. It helps to reduce the use of complex statement in a program.
4. It makes the program run faster.

☺☺☺

Scanned with CamScan