20/10/2023

UNIT- 3

Phase 3 + Phase 4

Semantic Analysis Intermediate Code Generation

$E \rightarrow E+T$ { }	2+3*4	2+3*4
$E \rightarrow T$ { }	14	234*+
$T \rightarrow T*f$ { }		Postfix
$T \rightarrow f$ { }		
$f \rightarrow num$ { }		

Grammar + Semantic Rules = SDT (Syntax Directed Translation)

Syntax Directed Translation

→ It is not possible for a CFG to represent certain property such as uniqueness in type declaration or type compatibility in performing arithematic Operation or defining the region of variables being used in the program.

In compilation process these are certain features which are beyond the syntax of the language. Parser uses a CFG to validate the i/p string and produce o/p for next phase of the compiler. Output could be ~~next~~ either a parse tree or abstract syntax tree. Now ~~the~~ to interleave ~~syntax~~ Semantic anylsis phase of the compiler we use SDT.

SDT are augmented rules to the grammar that facilitates semantic grammar. SDT involves passing information bottom-up and/or top-down the parse tree in the form of attributes

20/10/2023

UNIT- 3

Phase 3 + Phase 4

Semantic Analysis   Intermediate Code Generation

E → E+T { }
E → T { }
T → T*f { }
T → f { }
f → num { }

2+3*4	2+3*4
14	234*+
	Postfix

Grammar + Semantic Rules = SDT (Syntax Directed Translation)

Syntax Directed Translation

→ It is not possible for a CFG to represent certain property such as uniqueness in type declaration or type compatibility in performing arithematic Operation or defining the region of variables being used in the program.

In compilation process these are certain features which are beyond the syntax of the language. Parser uses a CFG to validate the i/p string and produce o/p for next phase of the compiler. Output could be ~~next~~ either a parse tree or abstract syntax tree.

Now ~~the~~ to interleave ~~syntax~~ semantic anylsis phase of the compiler we use SDT.

SDT are augmented rules to the grammar that facilitates semantic grammar. SDT involves passing information bottom-up and/or top-down the parse tree in the form of attributes attached to the node.

SDT rules use –

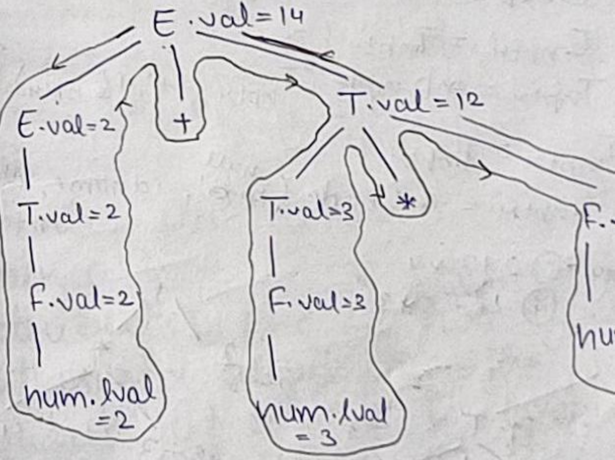
1) Lexical value
2) Constant
3) Attributes associated to the no their definition.

→ The general approach to SDT is parse tree and compute the v at the nodes of the tree by v some order.

Ques) Define SDT to evaluate a g expression with (+, *)

Solⁿ:

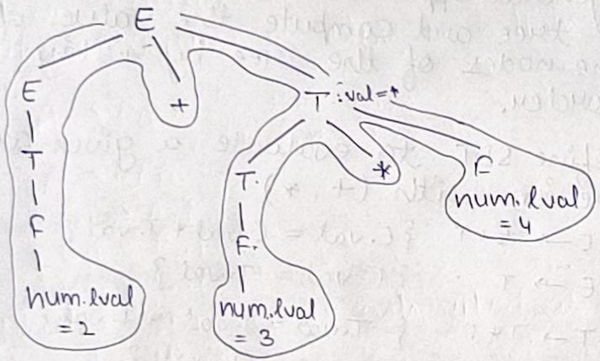
E ⟶ E+T { E.val = E.val + T.v
E → T { E.val = T.val }
T → T*f { T.val = T.val * f
T → f { T.val = f.val }
f → num { f.val = num.lva



Q2) Design SDT to convert Infix

Same grammar.

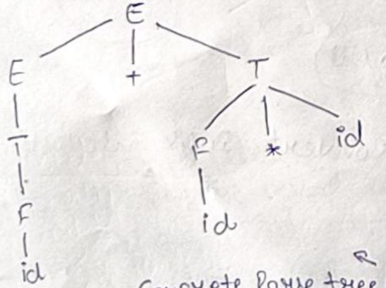
Sol^n 2)
E → E+T { print ("+"); }
E → T { }
T → T*F { print ("*"); }
T → F { }
F → num { print ("num.lval"); }

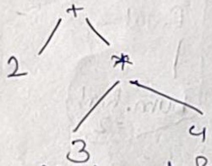


Ques) Define SDT to build abstract syntax tree.

Sol^n)
E → E+T { E.n~~t~~ptr = mk node (E.nptr, '+', T.nptr)} mk ↓ make
E → T { E.nptr = T.nptr }
T → T*F { T.nptr = mk node (T.nptr, '*', f~~E~~.nptr) }
T → F { T.nptr = f.nptr }
F → id { f.nptr = mk node (~~[illegible]~~ null, idname, null) }

Let the string: (i) 2+3*4
(ii) 4 + 5*3

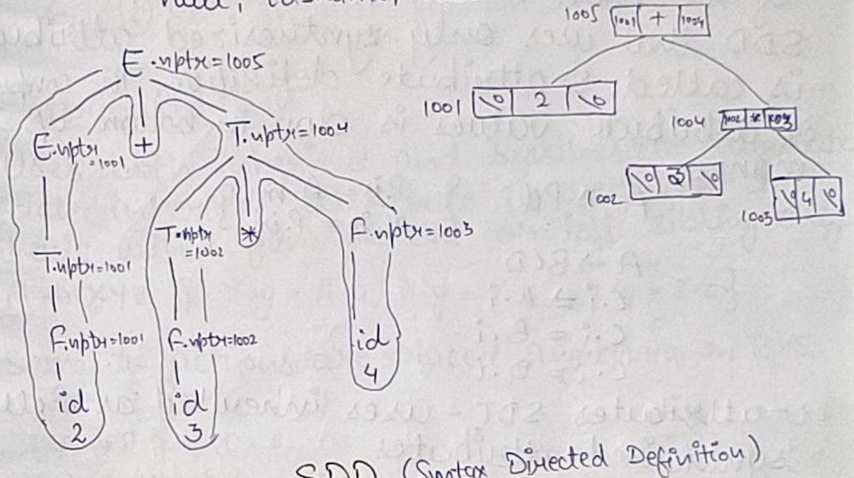




Abstract Parse Tree
(It does not show the details about the variables)

Concrete Parse tree / Parse Tree / Syntax tree/Annotated Parse tree.

- nptr (node pointer)
- mk node (make node) → function will create a node with three fields –
null, id_name, null



SDD (Syntax Directed Definition)

SDD is a generalization of CFG in which each grammar production $X \longrightarrow \alpha$ is associated with it a set of semantic rules of the form
$a := f(b_1 b_2 --- b_K)$ where a ~~b~~ is an attribut obtain form function f.

Attribute: The attribute can be a string, a number, a type, a memory location etc.

Attributes are of two types.

1) Synthesized attribute: The value of synthesize attributes at a node is computed from the values of attributes at the children of that node in the parse tree.

X → ABC { X.x = f(A.a, B.b, C.c) }

2) Inherited attribute: The inherited attributes can be computed from the values of the attributes at the siblings and parent of the node.

SDD that uses only synthesized attributes is called S-attribute definition. The computation of attribute values is done in bottom up manner.

$A \rightarrow PQ \quad \{ P.in = A.in$
$\quad\quad\quad\quad\quad Q.in = P.in \}$

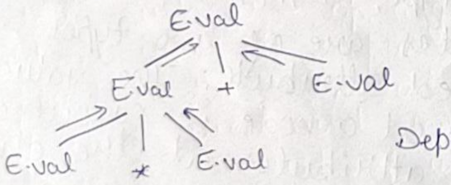
$A \rightarrow BCD$
$C.i \rightleftharpoons A.i$
$C.i = B.i$
$C.i = D.i$

L-attributes SDT - uses inherited as well as synthesised attributes.

Dependency Graph.

→ The directed graph that represent the interdependencies both synthesized and inherited attributes at nodes in the parse tree is called dependency graph.

$E \rightarrow E_1 + E_2$
$E \rightarrow E_1 * E_2$



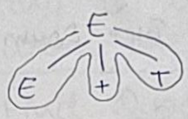
Dependency graph

S-attributes SDT

1) Uses only synthesised attributes
2) Semantic action are placed at right end of production.

3) Attributes are evaluated during BUP
↑
Bottom up parser.

$A \rightarrow BCC \{ \}$



To reach E we have evaluated E+T in BUP fashion scanned all the children.

L-attributed SDT

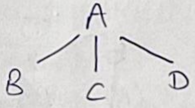
1- Uses both inherited and synthesized attribute. Each inherited attribute is restricted to inherit either from parent or left sibling only.

$A \rightarrow XYZ \; \{ Y.y = A.a, \; Y.y = X.x, \; Y.y \overset{\times}{=} Z.z \}$

2) Semantic actions are placed anywhere on RHS.

$A \rightarrow \{ \} BC$
$A \rightarrow P \{ \} Q$
$A \rightarrow XY \{ \}$

3) Attributes are evaluated by traversing parse tree depth first, left to right.



By the time we reach A we have seen all parent & left sibling (tree) have be watched

Ex $A \rightarrow BC \quad \{ B.S = A.S \}$

a) S-att
b) L-att
c) both
d) ~~Have~~ none.

Intermediate Code.

Ex: (a+b)*(a+b+c)

Intermediate Code.
- Linear form
  - Postfix
  - Three address Code.
- Tree form
  - Syntax Tree
  - DAG

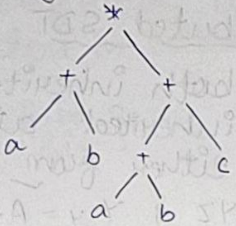
Postfix

ab+ab+c+*

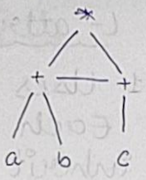
Three address Code.

$t_1 = a+b$
$t_2 = a+b$
$t_3 = t_2+c$
$t_4 = t_1 * t_3$

Syntax Tree



DAG



DAG
↓
Directed Acyclic Graph

Implementation of 3-address code.

Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields. There are 3 representations used for three address code such as—

i) Quadruples
ii) Triples
iii) Indirect Triples.

Quadruples

→ The quadruples is a structure with at the most 4 fields such as - op, arg1, arg2, result. 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.

Triples.

→ In the triples representation the use~~d~~ of temp. variable is avoided by reffering the pointers in the symbol table.

Indirect Triples.

→ In the indirect triples representation t listing of triples is been done. And listing pointers are used instead of using statem

Ques) Find Quadruple, Triple and indirect tripl for the given statement

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

Ans) $t_1 = a+b$
$t_2 = -t_1$
$t_3 = c+d$
$t_4 = t_2 * t_3$
$t_5 = a+b$
$t_6 = t_5 + c$
$t_7 = t_4 + t_6$

Quadruples

	op	op1	op2	result.
1	+	a	b	t_1
2	−	t_1	---	t_2
3	+	c	d	t_3
4	*	t_2	t_3	t_4
5	+	a	b	t_5
6	+	t_5	c	t_6
7	+	t_4	t_6	t_7

triple.

	op	op1	op2
1	+	a	b
2	−	(1)	
3	+	c	d
4	*	(2)	(3)
5	+	a	b
6	+	(5)	c
7	+	(4)	(6)

Indirect- Triple.

(i)	(1)
(ii)	(2)
(iii)	(3)
(iv)	(4)
(v)	(5)
(vi)	(6)
(vii)	(7)

Advantage Quadruple.

→ Statement can move around.

Disadvantage Quadruple.

→ Too much space is wasted.

Advantage Triple.

→ Space is not wasted

Disadvantage Triple.

→ Statements can not mo

Advantage Indirect Triple.

→ Statement can be move

Disadvantage Indirect Tripl

→ Two memory access is requir

Ques) find all the three for given statement.

i) $x = -a * b + -a * b$ (ii) $-(a*b) + (c+d) - (a+b+c+d)$

Ques) Explain Merit and Demerit for Quadruple, Indirect and ~~Indirect~~ Triple.

Types of 3-address Code.

i) $x =$ ~~y(z)~~ y op z (e.g $x = a+b$)

ii) $x =$ op z (e.g ~~x~~ unary operator)

iii) $x = y$ (assignment operator)

iv) ~~Ques~~ If x (~~b~~rel op) y Goto L
↑relational

v) Goto L

vi) $A[i] = x$ (Array variable)
$y = A[i]$

vii) $x = *P$
$y = \&x$ (Pointer variable)

3-address code for 2D array.

$$\begin{bmatrix} 00 & 01 & 02 \\ 10 & 11 & 12 \\ 20 & 21 & 22 \end{bmatrix}$$

RMR

00	01	02	10	11	12	20	21	22

$\underbrace{00\ 01\ 02}_{Row1}\ \underbrace{10\ 11\ 12}_{Row2}\ \underbrace{20\ 21\ 22}_{Row3}$

CMR (Column Major Representation)

00	10	20	01	11	21	02	12	22

$\underbrace{00\ 10\ 20}_{Col1}\ \underbrace{01\ 11\ 21}_{Col2}\ \underbrace{02\ 12\ 22}_{Col3}$

$A[2, 1]$ ← No of Elements in One row
$= 2 \times 3 + 1$
$= 7 * 4$ ← If we take int. for bytes.

skip 2 rows
$A = [2, 2]$ ← skip two elements

matrix name
$x = A[y, z]$ A(10*20)

$(y * \underset{\uparrow}{20} + z) * 4$
Elements in 1 row.

a+b+c

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

$(y*20+z)*4$

$t_1 = y*20$
$t_2 = t_1 + z$
$t_3 = t_2 * 4$
$t_4 =$ base address of A.
$x = \underbrace{t_4}_{\text{Base}}[\underbrace{t_3}_{\text{offset}}]$

Ques1) Generate 3-address code for the statement $x = A[i, j]$ for an array of size 10x 20. Assume $low_1 = 1$ and $low_2 = 1$, $n_1 = 10$ & $n_2 = 20$.

Sol: $A[i,j] = ((i \times n_2) + j) * w\} + \{\underline{Base} - ((low_1 * n_2) + low_2) * \underline{w}$
$= Base + (i - low_1) * n2 + (j - low2)) * w$
$= Base + ((i-1) * 20 + (j-1)) * 4$
$= Base + (20i - 20 + j - 1) * 4$
$= Base + (80i - 80 + 4j - 4)$
$= (Base - 84) + (4 * (20i + j))$

3-address code.

$t_1 = 20 * i$
$t_2 = t_1 + j$
$t_3 = c$ //computation of base-84
$t_4 = 4 * t_2$
$t_5 = t_3[t_4]$
$x = t_5$

Q) Translate the following C-code into 3 address code.

```
int i;
int a[10][10];
i=0;
while (i<10)
{ a[i][i]=1;
  i++;
}
```

Consider that values are stored in RMR. Assume 4 bytes calculation per word.

Solⁿ: $a[i,j] = Base + ((i - low_1) * n_2 + (j - low_2)) * w$

$= Base + ((i-0) * 10 + (j-0)) * 4$ → question me j ki jagh i diya.

$= Base + (10i + i)4$

$= Base + 44i$

Now 3-address code is —

```
1) i = 0
2) if (i<10) goto 4
3) goto 9
4) t2 = 44 * i
5) a[t2] = 1;   or [ t2 = base address of a
                   [ t2[t1] = 1;
6) t2 = i+1
7) i = t2
8) goto 2
9) Stop
```

if / while / for / do

if x (rel op) y goto —
goto —

Formula 1D array

$\boxed{A[i] = base + i \times w}$

Ques) Generate the 3-address code for

$C[i, j] = A[i,j] + B[i,j] + C[i,j] + D[i+j]$

size $A=B=C=10 \times 20$
$w = 4$.
$low_1 = 1$
$low_2 = 1$

~~Ans~~ $\boxed{A[i] = base + i * w}$ formula 1D array.

Let $w = 4$

Solⁿ 3-address code

```
t1 = 4*i
t2 = Base address of A
t2[t1]
```

Ques) Generate three-address code for the following program in c

```
while (i>10)
{ x = 0;
  a = a+5;
}
```

Solⁿ: 3-address code.

```
100 - if (i>10) goto 102
101 - goto 106
102 - x = 0
103 - t1 = a+5
104 - a = t1
105 - goto 100
106 - Lnext.
```

Back patching.

→ Leaving the labels as empty and filling them later is called ~~bo~~ back patching.

Ques) while ((A < C) && (B < D)) do
if A == 1 then C = C+1
else while A <= D do A = A+2

Solⁿ 3-address code.

```
100) if (A< C) goto 102
101) goto 115
102) if (B < D) goto 104
103) goto 115
104) if (A==1) goto 106
105) goto 110
106) t1 = C+1
107) C = t1
108) goto 100
109) if (A <= D) goto 111
110) goto 100
111) t2 = A+2
112) A = t2
113) goto 110
114) goto 100
115) Next
```

Case Statement

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

3-address code

```
1) if (ch=1) goto L1
2) if (ch=2) goto L2
   L1:  t1 = a+b;
        c = t1
        goto Last
   L2:  t2 = a-b;
        c = t2
        goto Last
3) Last
```

3-address code for Procedure Call.

```
Param a1
Param a2
________
________
Param an
Call P, n
```

P(a1, a2 ---- an)

n- no. of parameters.

Ques) Generate a three address code

```
main ( )
{
   -----
   -----
   a(x);
   ----
   ----
}
a (int x)
{ x=x+1;
}
```

Soln) 3-address Code

```
1) Call main ( )
2) Param x
3) Call a, 1   ← It denotes we are giving 1 parameter.
4) t2 = x + 1
5) x = t2
6) Stop
```

Q) Generate a 3-address code for the procedure cal

```
void main ( )
{ int x, y;
  --------
  Swap (&x, &y)
  ----
  ----
}
void swap (int *a, int *b)
  { int i;
      i = *b
      *b = *a
      *a = i
  }
```

Soln) 3-address code

```
1) Call main
2) Param &x
3) Param &y
4) Call swap, 2
5) i = *b        ↑ It mea
6) *b = *a       we are
7) *a = i;       giving 2
8) Stop          argument
```

Q-) ① Generate 3-address code for the following ~~procedure~~ segment call

```
1) c = 0
2) do {
     if a<b then
       x++;
     else
       x--;
     c++; } while c<5
```

```
1) c = 0
2) if (a<b) goto 4
3) goto 5
4) x++
5) x--
6) c++
7) if (c<5) goto 2
8) goto 9
9) Next
```

Q) ② ~~Gene~~ For (i=1; i<=10; i++){
 a[i] = x*5; }

Soln)

```
100) i=1
101) if (i<=10) goto 103
102) goto ~~Next~~ 106
103) t1 = x*5
104) a[i] = t1
105) i++
106) Next
```

Ans ①

```
        C = 0
    L1: if ( i<10) goto L2
          goto L3
    L2: $t_2 = x+1;$
          $x = t_1$
          gota L4
    L3: $t_2 = x-1$
          $x = t_2$
          goto L4
    L4: $t_3 = c+1$
          $c = t_3$
    L5: if (c<5) goto L1
          goto Last
    Last: Stop
```

Ans 2)

```
1- i = 1
2. if (i<=10) goto 4
3- goto 11
4- t = x*5.
5- $t_1 = 4*i$
6- $t_2$ = base address of a
7- $t_2[t_1] = t$
8- $t_3 = i+1$
9- $i = t_3$
10- goto 2
11- stop
```