Tree Traversal :
- Depth first
- Breadth first

SDT combines both grammar as well as semantic rules.

SDT = CFG + semantic rule

In SDT, every non terminals can get 0 or more attributes. In semantic rule, attribute may be string, numbers, memory locations etc. The semantic actions are enclosed within {}. The most general approach for SDT is To construct a parse tree or a syntax tree, & Then To compute values of attributes at nodes of The Tree by visiting nodes of the Tree.

Syntax Directed Definitions (SDD) :

SDD = CFG + Attributes + rules

2 Types :
1. Synthesized
2. Inherited

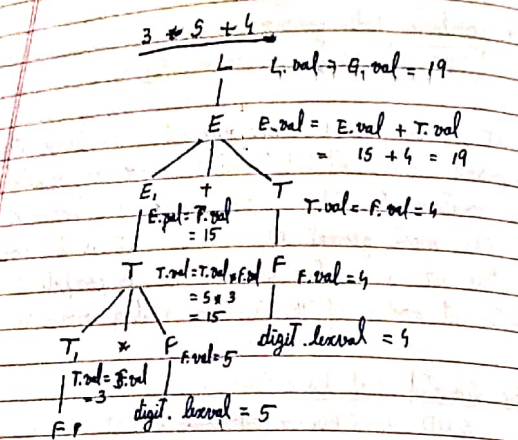
Synthesized attribute for a non terminal A at a parse tree node N is defined by a semantic rule associated with production at N. A synthesized attribute at node N is defined only in terms of attribute values at children of N & N itself.

An inherited attribute for a non-terminal B at a parse tree node N is defined by a semantic rule associated with production at parent of N. An inherited attribute at node N defined only in terms of attribute values at N's parents, N itself & N's siblings

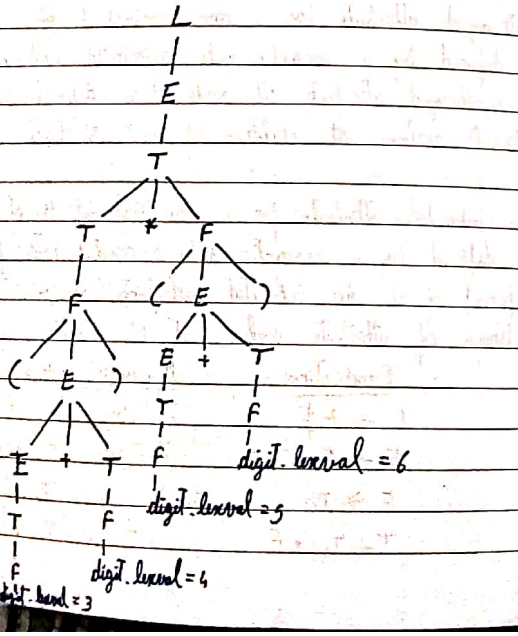
Production	Semantic rule
$L \rightarrow E\$$	{ L.val = E.val }
$E \rightarrow E_1 + T$	{ E.val = E_1.val + T.val }
$E \rightarrow T$	{ E.val = T.val }
$T \rightarrow T_1 * F$	{ T.val = T_1.val * F.val }

$T \to F$ { T.val = F.val }
$F \to (E)$ { F.val = E.val }
$F \to digit$ { F.val = digit.lexval }

3 * 5 + 4

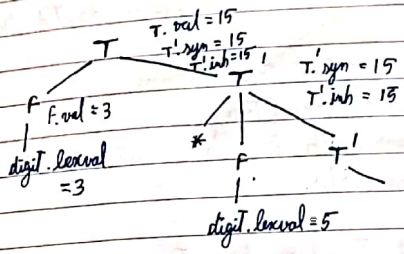


(3 + 4) * (5 + 6)



$T \to FT'$ { T'.inh = F.val
$T' \to *FT'$ T.val = T'.syn
$T' \to \varepsilon$ T'.inh = T'.inh * F.val
$F \to digit$ T'.syn = T'.inh
T'.syn = T'.inh
F.val = digit.lexval

3 * 5



- We associate information with programming language constructs b attatching attributes to grammar symbols.
- Values of these attributes are evaluated by semantic rules associated with production rules.
- Evaluation of these semantic rules:
  - May generate intermediate codes
  - May put information into symbol table
  - May perform type checking
  - May issue error checking
  - May perform some other activities
  - Infact, they may perform almost any activities
- An attribute may hold almost anything.
  - a string, a number, a memory location, a complex record

Syntax-Directed Definitions & Translations Schemas

1. When we associate semantic rules with productions, we use two notations:
   - Syntax Directed Transl Definitions
   - Translation schemes

4. Syntax Directed Definitions

- Conceptually with both syntax directed translation & Translation schemes we
  - parse input token stream
  - Build parse tree
  - Reverse the tree to evaluate the semantic rules at the parse tree nodes.

Input string → parse tree → dependency graph → evaluation order for semantic rules

Conceptual views of syntax directed translation.

1. A syntax-directed definition is a generalization of a context-free grammar in which:
   - Each grammar symbol

synthesized attribute : $E \rightarrow E1 + E2 \quad \{ E.val = E1.val + E2.val \}$
Inherited attribute : $A \rightarrow XYZ \quad \{ Y.val = 2 * A.val \}$

Annotated parse tree :

1. Parse tree showing the values of attributes at each node is called annotated parse tree.
2. Values of attributes in nodes of annotated parse tree are either

Example of SDD:

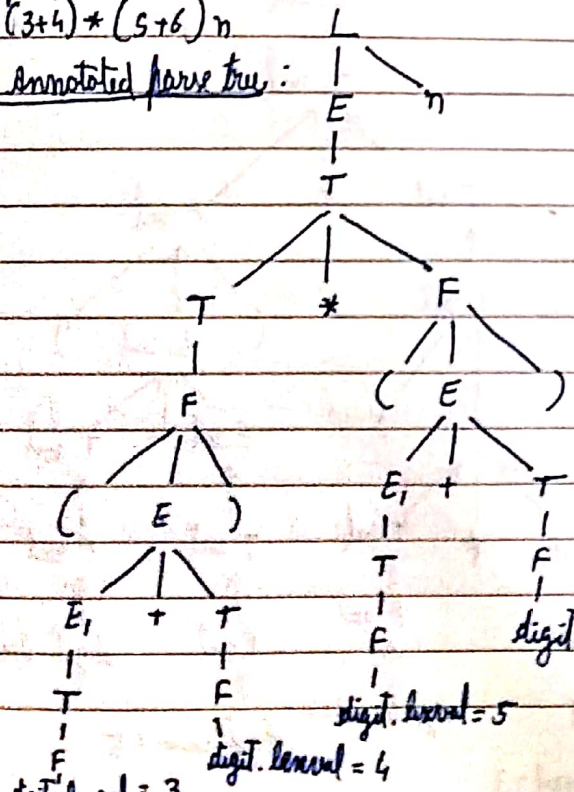
Production	Semantic rules
0 $L \to En$	print (E.val)
$E \to E_1 + T$	E.val = E_1.val + T.val
$E \to T$	
$T \to T_1 * F$	
$T \to F$	
$F \to (E)$	
$F \to$ digit	

S-attributed definition:

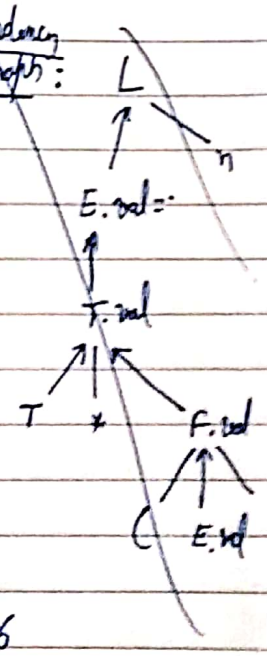
- A syntax directed translation that uses synthesized attributes exclusively is said to be a S-attributed definition
- A parse tree for an S-attributed definition can be annotated by evaluating the semantic rules for attributes at each node, bottom up from leaves to the root.

i) (3+4)*(5+6)n

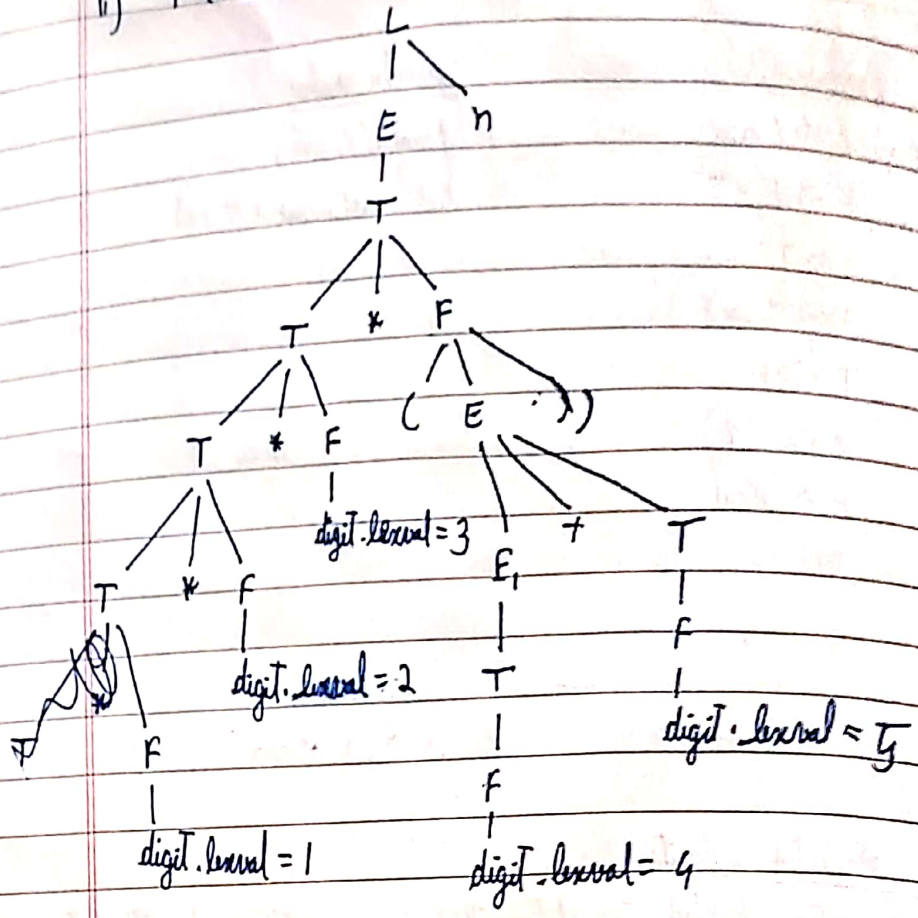
Annotated parse tree:



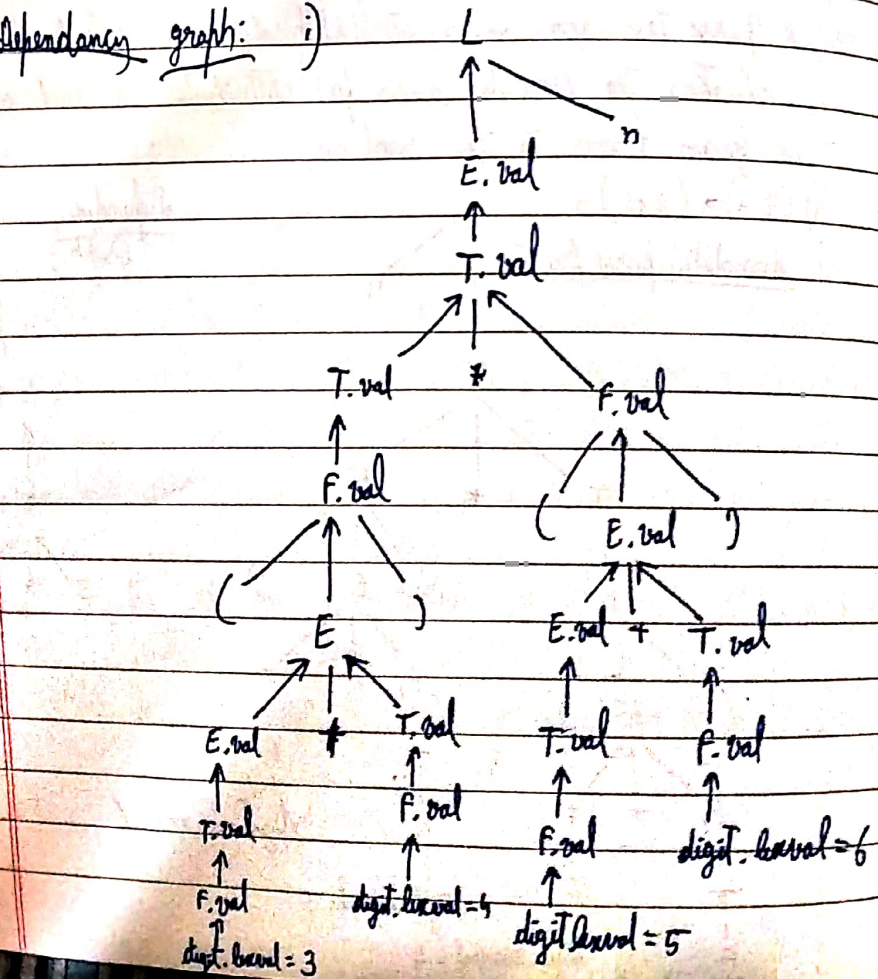
Dependency graph:



ii) 1 * 2 * 3 * (4+5) n

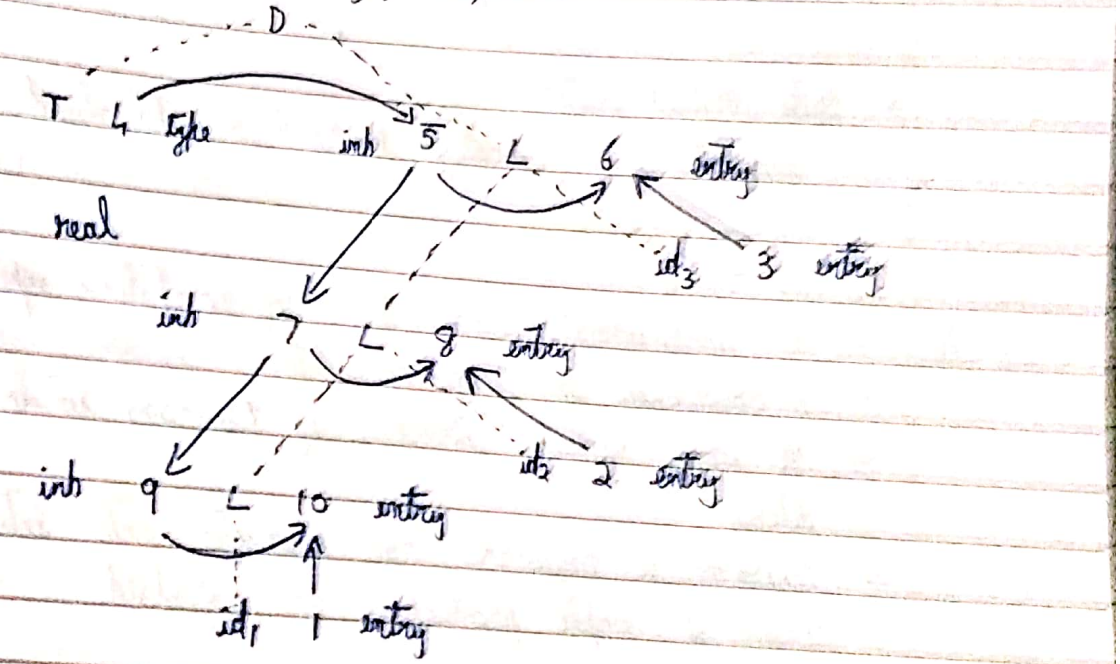


Dependancy graph: i)



9

$D \to TL$	$L.in = T.type$
$T \to int$	$T.type = integer$
$T \to real$	$T.type = real$
$L \to L_1, id$	$L_1.in = L.in$ addType (id.entry, L.in)
$L \to id$	addType (id.entry, L.in)



Dependency graph for a declaration float id_1, id_2, id_3.

int a, b;
double a=4, b;

Evaluating Semantic Rules :

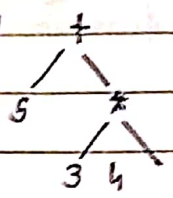
- Parse Tree methods :
  - At compile time evaluation order obtained from Topological sort of dependency graph
  - Fails if dependency graph has a cycle.

- Rule-Based methods :
  - semantic rules analyzed by hand or specialized

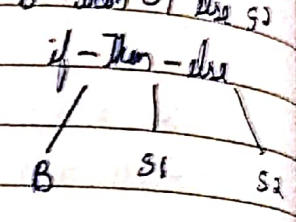
Syntax Trees :

- an intermediate representation of compiler's input
- A condensed form of parse tree
- It shows syntactic structure of program while omitting irrelevant details
- Operators & keywords are associated with interior nodes
- chains of simple productions are collapsed.

Eg: Expression



if B Then S1 else S2

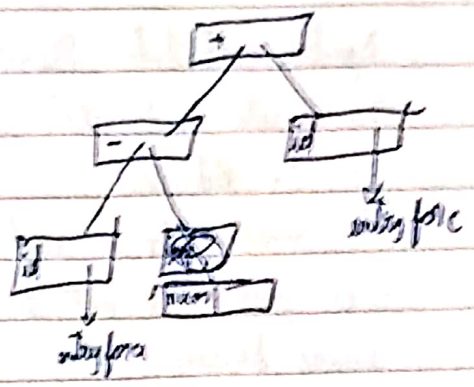


- leaves : identifiers or constants
- Internal nodes : labelled with operations
- Children of a node

Constructing Syntax Tree for expressions

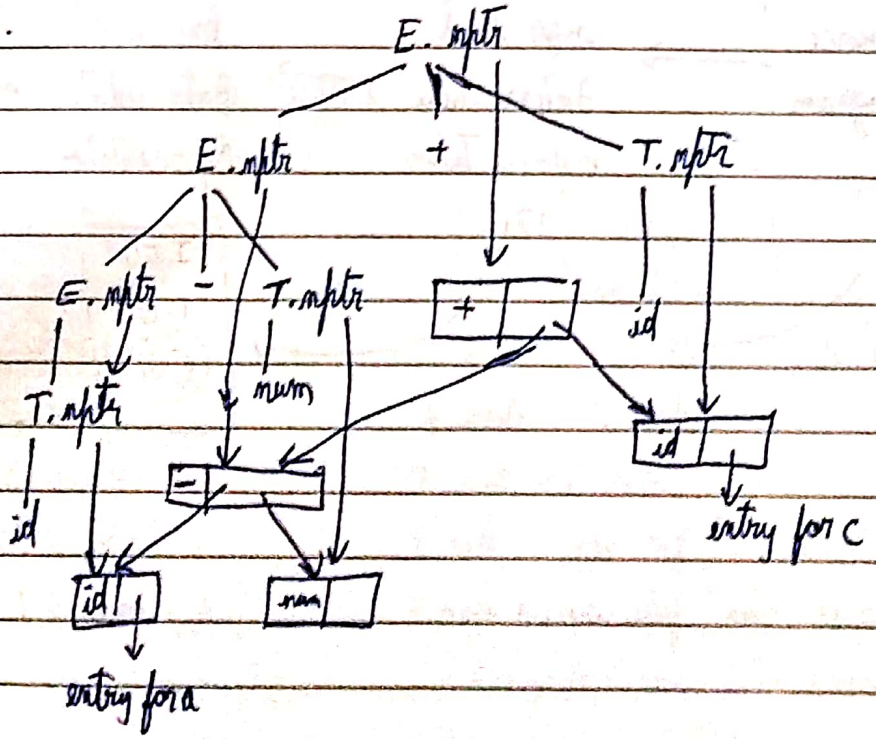
- Each node can be implemented as a record with several fields.
- Operator node : one field identifies the operator (called label of node) & remaining fields contains pointers to operands.
- Nodes may also contain fields to hold values (pointers to value) of attributes attached to nodes.
- Functions used to create.

Eg: a - 4 + c



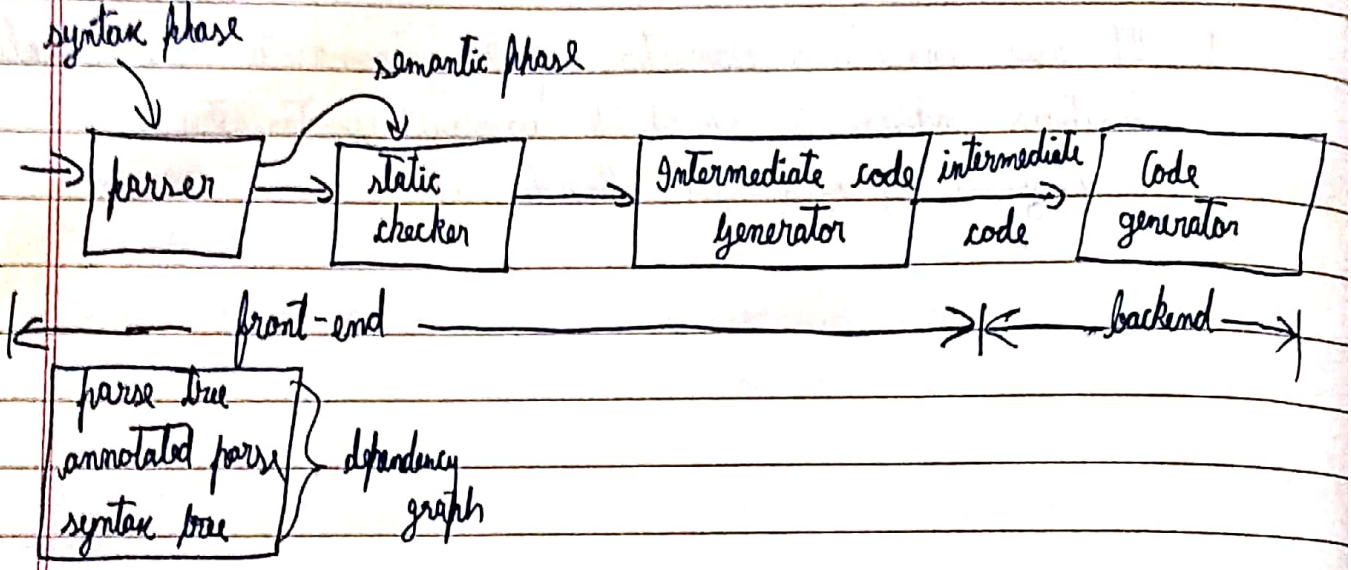
SDD for constructing syntax Tree

1. It uses underlying productions of the grammar to schedule calls of functions mkleaf & mknode to construct syntax tree
2. Employment of synthesized attribute nptr

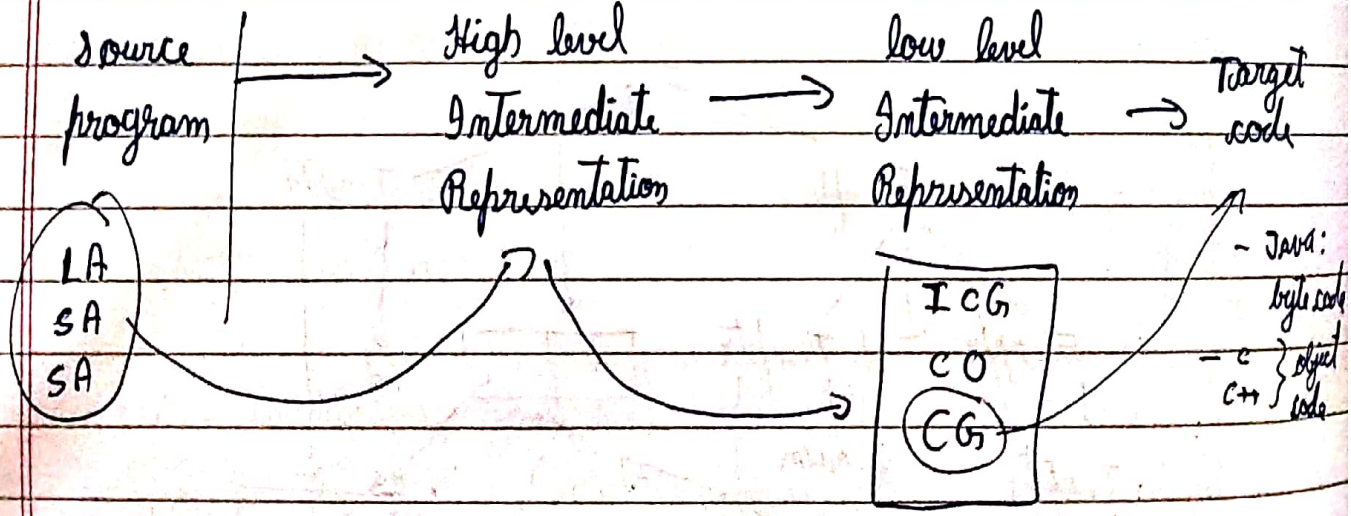


S-Attributed Definitions

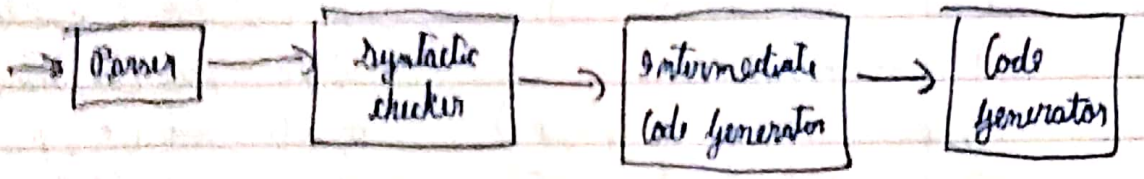
- Syntax-attributed Definitions are used to specify syntax-directed translation.
- To create a translator for an arbitrary syntax-directed definition can be difficult.
- We would like to evaluate semantic rules during parsing (i.e., in a single pass, we will parse & we will also evaluate semantic rules during parsing).
- We will look at 2 sub-classes of syntax-directed definitions.



sequence of intermediate representations

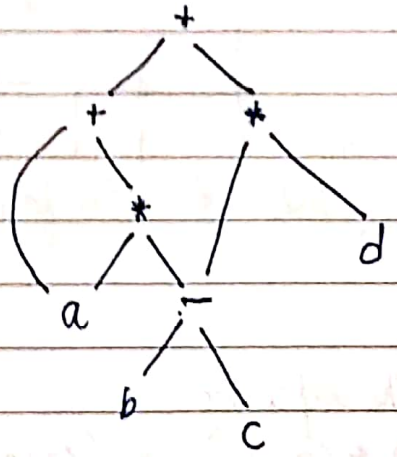


Intermediate



Variants of syntax trees :

- It is sometimes beneficial to create a DAG instead of Tree for expression
- This way we can easily show the common sub-expressions & then use that knowledge during code generation.
- Example : a+a * (b-c) + (b-c) * d



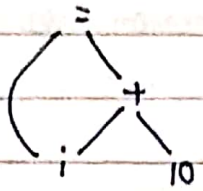
SDD for creating DAG's

Production	Semantic rules
1) $E \Rightarrow E1 + T$	E.node = new Node ('+', E1.node, T.node)
2) $E \Rightarrow E1 - T$	E.node = new Node ('-', E1.node, T.node)
3) $E \Rightarrow T$	E.node = T.node
4) $T \Rightarrow (E)$	T.node = E.node
5) $T \Rightarrow id$	T.node = new leaf (id, id.entry)
6) $T \Rightarrow num$	T.node = new leaf (num, num.val)

Eg:

1) p1 = leaf (id, entry - a)
2) p2 = leaf (id, entry - a) = p1
3) p3 = leaf (id, entry - b)
4) p4 = leaf (id, entry - c)
5) ~~p5 = leaf (id~~
5) p5 = Node ('-', p3, p4)
6) p6 = Node ('*', p1, p5)
7) p7 = Node ('+', p1, p6)
8) p8 = leaf (id, entry - b) = p3
9) p9 = leaf (id, entry - c) = p4
10) ~~p10 = leaf (id~~
10) p10 = Node ('-', p3, p4) = p5
11) p11 = leaf (id, entry - d)
12) p12 = Node ('*', p5, p11)
13) p13 = Node ('+', p7, p12)

Value number method for constructing DAG

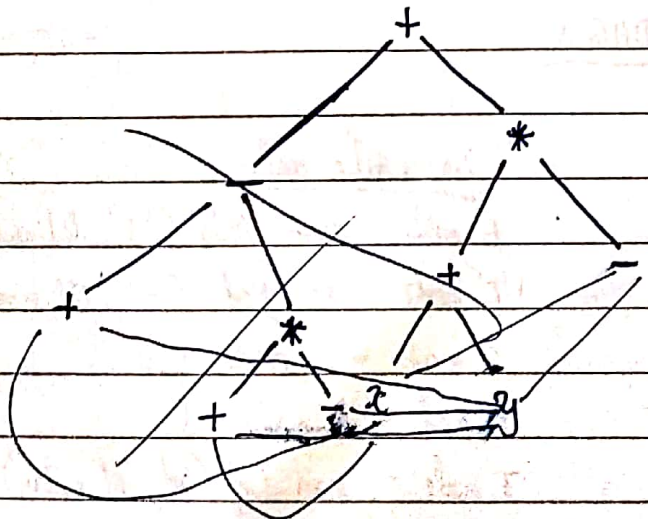


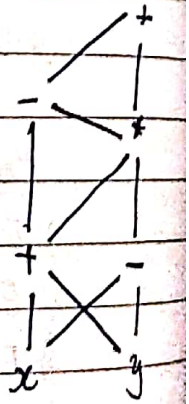
id			→ to entry for i
num	10		
+	1	2	
3	1	3	

Nodes of a DAG for i = i + 10 allocated in an array

Construct DAG for the expression

((x+y) - ((x+y) * (x-y))) + ((x+y) * (x-y))





1) p1 = leaf (id, entry - x)
2) p2 = leaf (id, entry - y)
3) p3 = Node ('+', p1, p2)
4) p4 = leaf (id, entry - x) = p1
5) p5 = leaf (id, entry - y) = p2
6) p6 = Node ('-', p1, p2)

7) P7

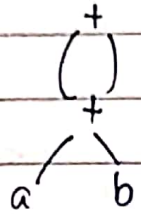
Construct DAG & identify value no.s for subexpressions for following expressions, assuming + associates from left.

a) a+b+(a+b)

b) ~~a+b+a+b~~ a+b+a+b

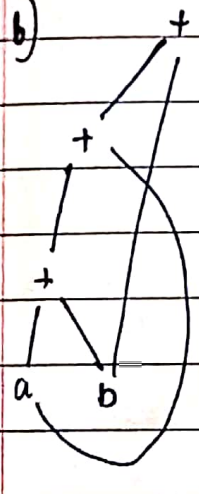
c) a+a+(a+a+a+(a+a+a+a))

a)



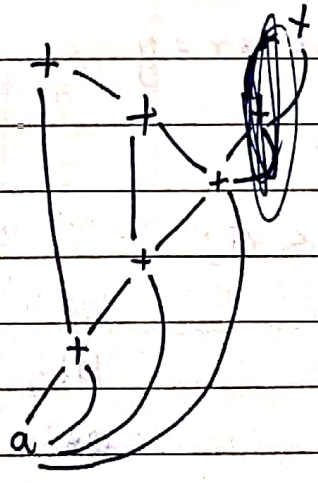
1	id	a	
2	id	b	
3	+	1	2
4	+	3	3

b)



1	id	a	
2	id	b	
3	+	1	2
4	+	3	1
5	+	4	2

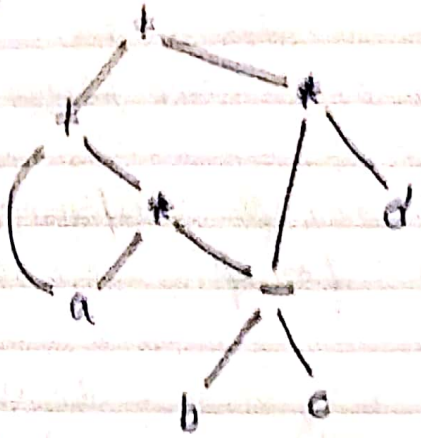
c)



1	id	a	
2	+	1	1
3	+	2	1
4	+	3	1
5	+	3	4
6	+	2	5

3 - address code

- In a 3 - address code, there is at most one operator at the right side of an instruction
- Eg:



$t1 = b - c$
$t2 = a * t1$
$t3 = a + t2$
$t4 = t1 * d$
$t5 = t3 + t4$

Forms of 3 address instructions:

- $x = y$ op z
- $x =$ op y
- $x = y$
- goto L
- If x goto L & if False x goto L
- If x relop y goto L
- Procedure calls using
  - param x
  - call p, n
  - $y =$ call p, n
- $x = y[i]$ & $x[i] = y$
- $x =$ & y and $x = *y$ and $*x = y$

Example:

- do $i = i+1$; while $(a[i] < v)$;

```
L:  t1 = i+1
    i = t1
    t2 = i*8
    t3 = a[t2]
    if t3 < v goto L
```

Symbolic labels

```
100: t1 = i+1
101: i = t1
102: t2 = i*8
103: t3 = a[t2]
104: if t3 < v goto 100
```

Position numbers

Data structures for 3-address codes:

- Quadruples
  - Has 4 fields: op, arg1, arg2 & result
- Triples
  - Temporaries are not used & instead references to instructions are made
- Indirect Triples
  In addition to triples, we use a list of pointers to Triples.

Example:

- b * minus c + b * minus c

3-address code

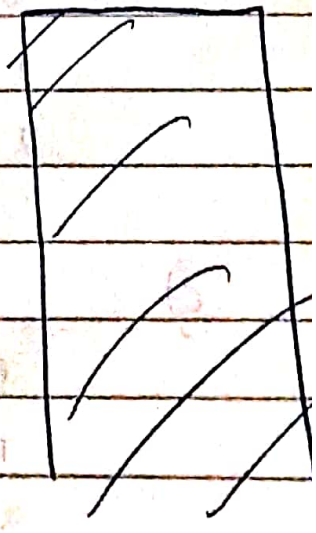
```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

Quadruples

	op	arg1	arg2	result
0	minus	c		t1
1	*	b	t1	t2
2	minus	c		t3
3	*	b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

Triples

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



Indirect Triples

	op
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

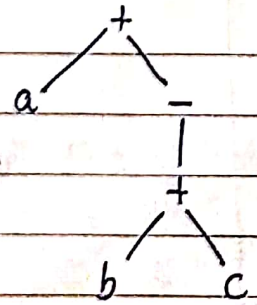
	op	arg1	arg2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Exercise:

Translate arithmatic expression a + -(b*c) into

a) syntax tree
b) Quadruples
c) Triples
d) Indirect Triples

= a)



b)

	op	arg1	arg2	result
0	*	b	c	t1
1	minus	t1		t2
2	+	a	t2	t3

c)

	op	arg1	arg2
0	+	b	c
1	minus	(0)	
2	+	a	(1)

d)

	instruction		op	arg1	arg2
35	(0)	0	+	b	c
36	(1)	1	minus	(0)	
37	(2)	2	+	a	(1)

$a = b[i] + c[j]$

Quadruples

(0)	=[]	b	i	t1
(1)	=[]	c	j	t2
(2)	+	t1	t2	t3
(3)	=	t3		a

Indirect Triples

0)	
1)	
2)	
3)	

Triples

(0)	=[]	b	i
(1)	=[]	c	j
(2)	+	(0)	(1)
(3)	=	(0)	(2)

$a[i] = b * c - b * d$

Quadruples

(0)	*	b	c	t1
(1)	*	b	d	t2
(2)	-	t1	t2	t3
(3)	[]=	a	i	t4
(4)	=	t3		t4

Indirect Triples

(0)	*	b	c
(1)	*	b	d
(2)	-	(0)	(1)
(3)	[]=	a	i
(4)	=	(3)	(2)

(0)
(1)
(2)
(3)
(4)

Triples

(0)	*	b	c
(1)	*	b	d
(2)	-	(0)	(1)
(3)	[]=	a	i
(4)	=	(3)	(2)

c.a = f(g(x)) + 2

(0)	+	y	1	t1
(1)	param	t1		
(2)	call	f	1	t2
(3)	+	t2	2	t3
(4)	=	t3		x

Triples

(0)	+	y	1
(1)	param	(0)	
(2)	call	f	1
(3)	+	(2)	2
(4)	=	x	(3)

Indirect Triples

(0)	+	y	1
(1)	param	(0)	
(2)	call	f	1
(3)	+	(2)	2
(4)	=	x	(3)

(0)
(1)
(2)
(3)
(4)

x = *p + &y

Static Single-Assignment form :

- Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations.
- 2 distinctive aspects distinguish SSA from 3-address code.
- First is that all assignments in SSA are to variables with distinct names; hence the term single static-assignment.

p = a+b	$p_1 = a+b$
q = p-c	$q_1 = p_1 - c$
p = q*d	$b_2 = q_1$
p = l-p	
q = p+q	