Syntax Directed Translations

11/10/22

1) Definition - Attributes / semantic rules.
2) Translation

$E = E + T$

$E.val = E.val + T.val$

⇓

semantic rule.

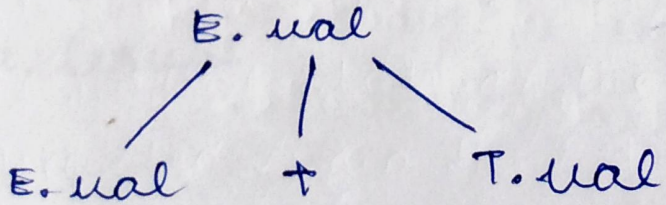
Attribute → val, attached to every grammar symbol.

attribute

synthesized — inherited

A syntax directed definition is a generalization of a CFG in which each grammar symbol had an associated set of attributes, partitioned into two subsets called the synthesized and inherited attributes of that grammar symbol. An attribute can represent a number, string, memory location etc. The value of an attribute at a parse tree node is defined by a semantic rule associated with the production used at that node. The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree; the value of an inherited attribute is computed from the values of attributes at the siblings and parent of that node.

E. val = E. val + T. val.



$A \rightarrow \alpha$

Semantic rule, $b = f(c_1, c_2, c_3, \ldots c_n)$

This is the form of a syntax directed definition.

Desk calculator program

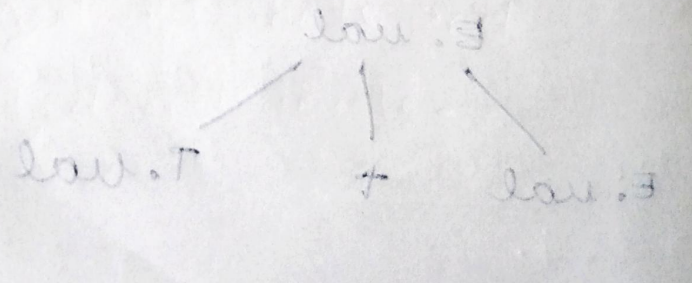
Production	Semantic Rules
$L \rightarrow En$	print ($E.val$)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ digit	$F.val =$ digit.lexval

Two attributes → val and lexval

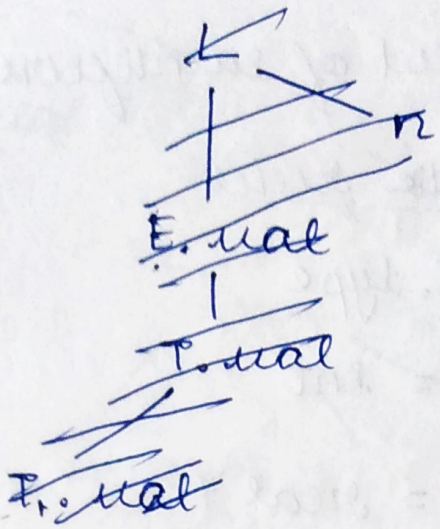
↓ all NT

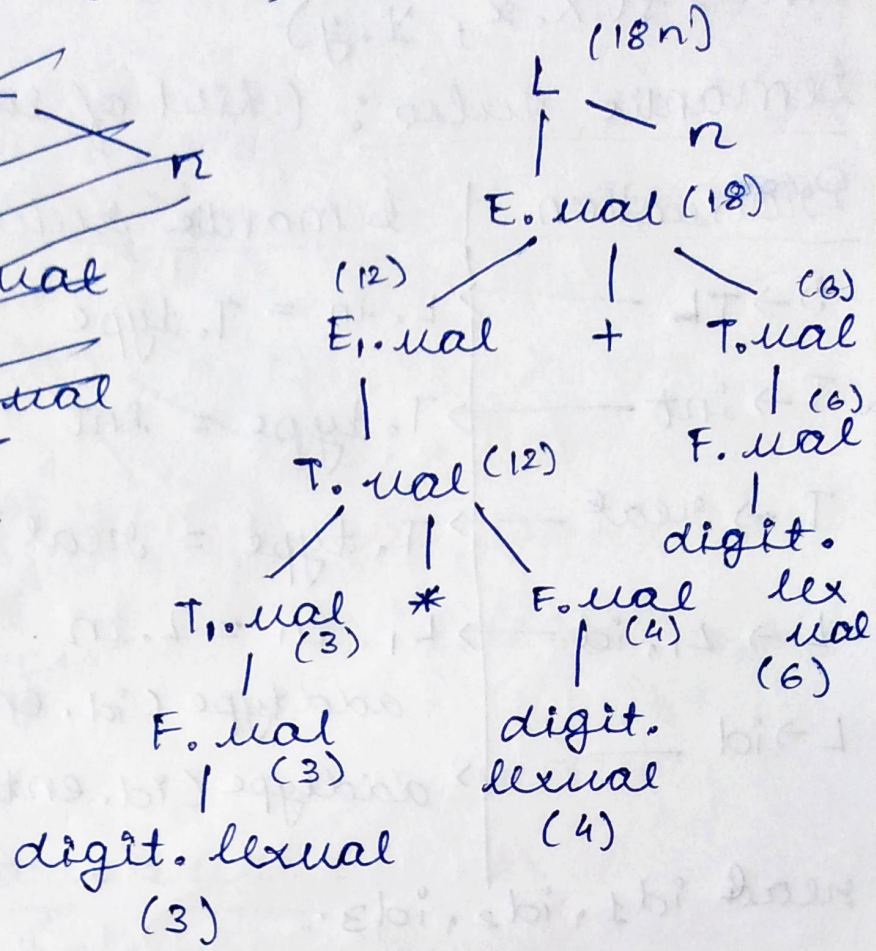
↓ all digit T

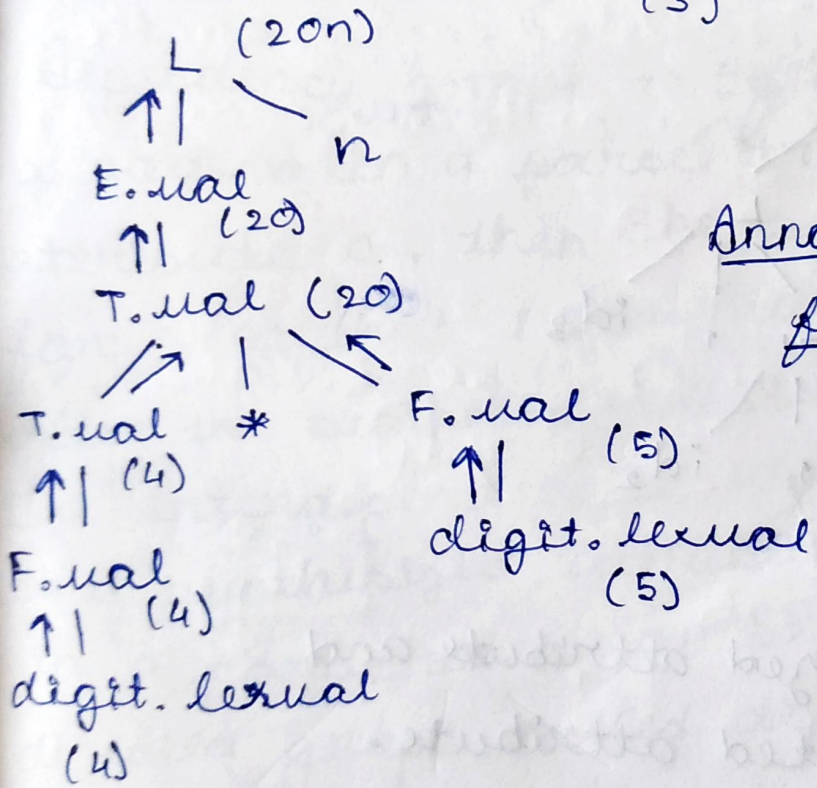
$3 * 4 + 6n$



Annotated parse tree →







Annotated parse tree for 4*5n

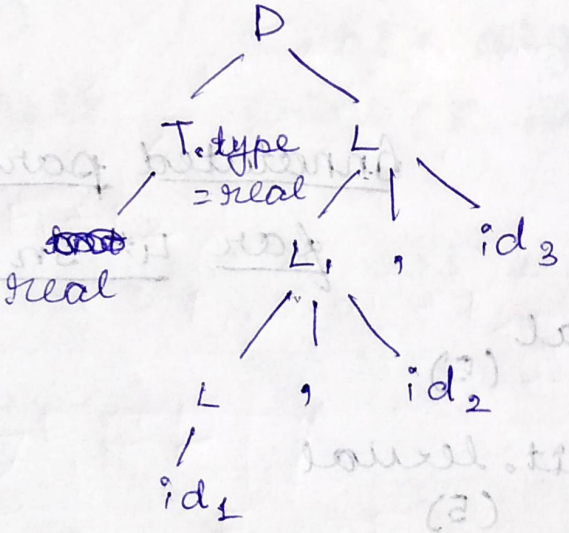
14/10/22

$A.a = f(X.x, Y.y)$

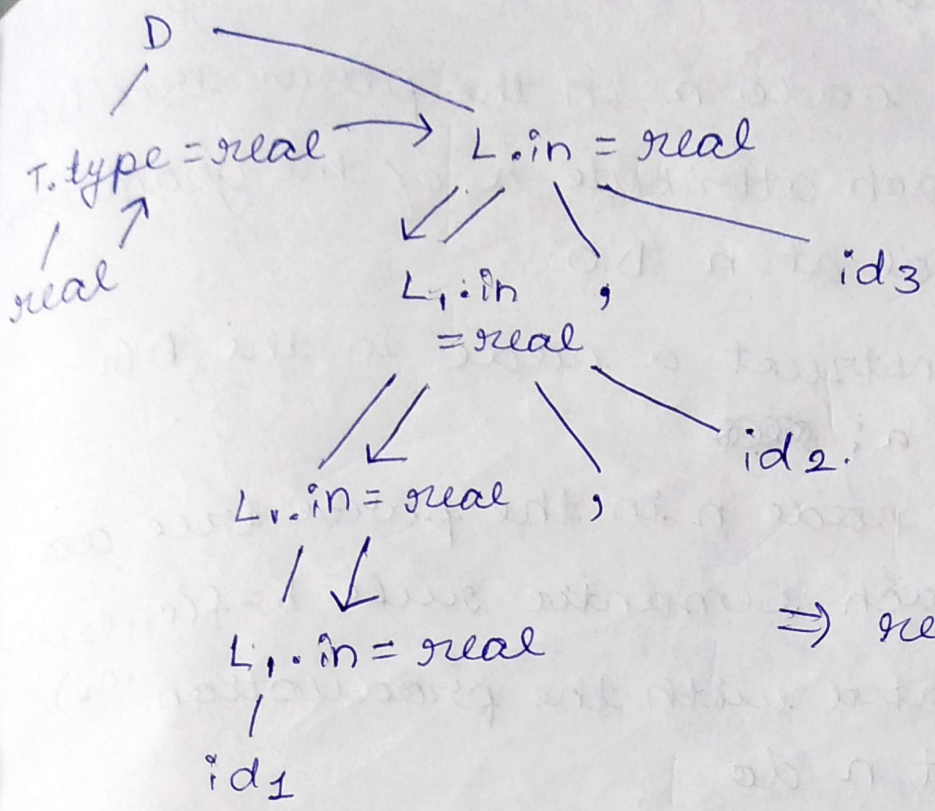
Semantic rules: (List of identifiers)

Production	Semantic Rules
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow int$	$T.type = int$
$T \rightarrow real$	$T.type = real$
$L \rightarrow L_1, id$	$L_1.in = L.in$. addtype(id.entry, L.in)
$L \rightarrow id$	addtype(id.entry, L.in)

read id_1, id_2, id_3.



real is synthesized attribute and in is inherited attribute.



$\Rightarrow$ real id_1, id_2, id_3.

Annotated parse tree

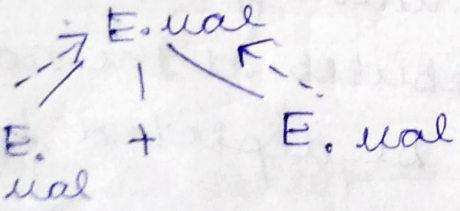
Dependency graph → If an attribute b at a node n in a parse tree depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c. The interdependencies among the inherited and synthesized attributes at the nodes in a parse tree can be depicted by a directed graph called dependency graph.

The DG is constructed as:

1. For each node n in the parse tree DO
2. For each attribute a of the grammar symbol at n DO
3. Construct a node in the DG for a;
4. For each node n in the parse tree do
5. For each semantic rule $b = f(c_1, c_2, \ldots, c_k)$ associated with the production used at n do
6. for i = 1 to K do
7. Construct an edge from the node for c_i to the node for b.

eg. $E \rightarrow E_1 + E_2$

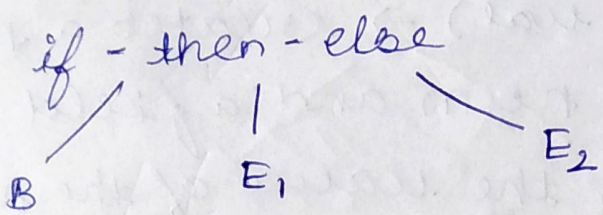
sem. rule : $E_1.val + E_2.val = E.val$



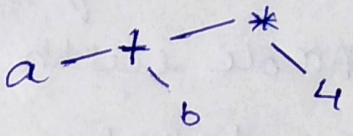
construction of Syntax Tree →

A ~~synta~~ syntax tree is a condensed form of parse tree useful for representing language constructs.

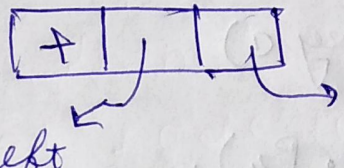
if B then E_1 else E_2



a + b * 4



Each node in a syntax tree can be implemented as a record with several fields. In the node for an operator, one field identifies the operator and the remaining fields contains pointers to the nodes for the operator.



We use the following func. to create the nodes of syntax tree for expression with binary operators. Each func. returns a pointer to a newly created node.

1) mknode(op, left, right) → creates an operator node with level op and two fields containing pointer to left and right.

2) mkleaf (id, entry) → creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry to the identifier.

3) mkleaf (num, val) → creates a number node with label num and a field containing val, the value of the number.

eg. a-6+c

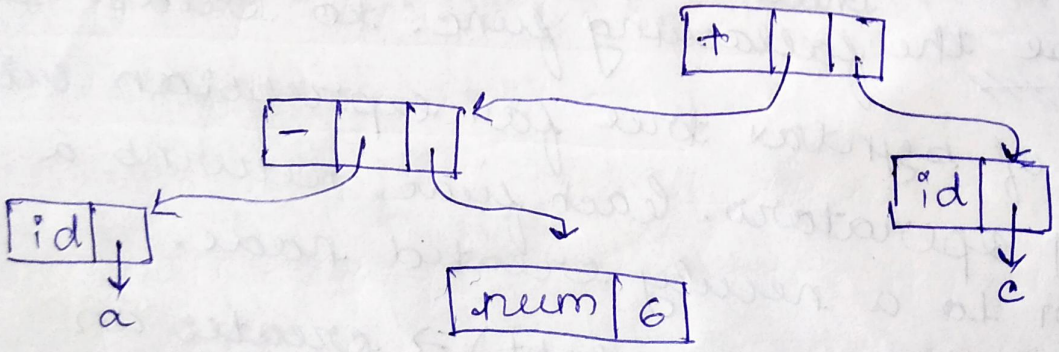
P_1 = mkleaf (id, entry a)

P_3 = mknode (−, P_1, P_2).

P_2 = mkleaf (num, 6)

P_4 = mkleaf (id, entry c)

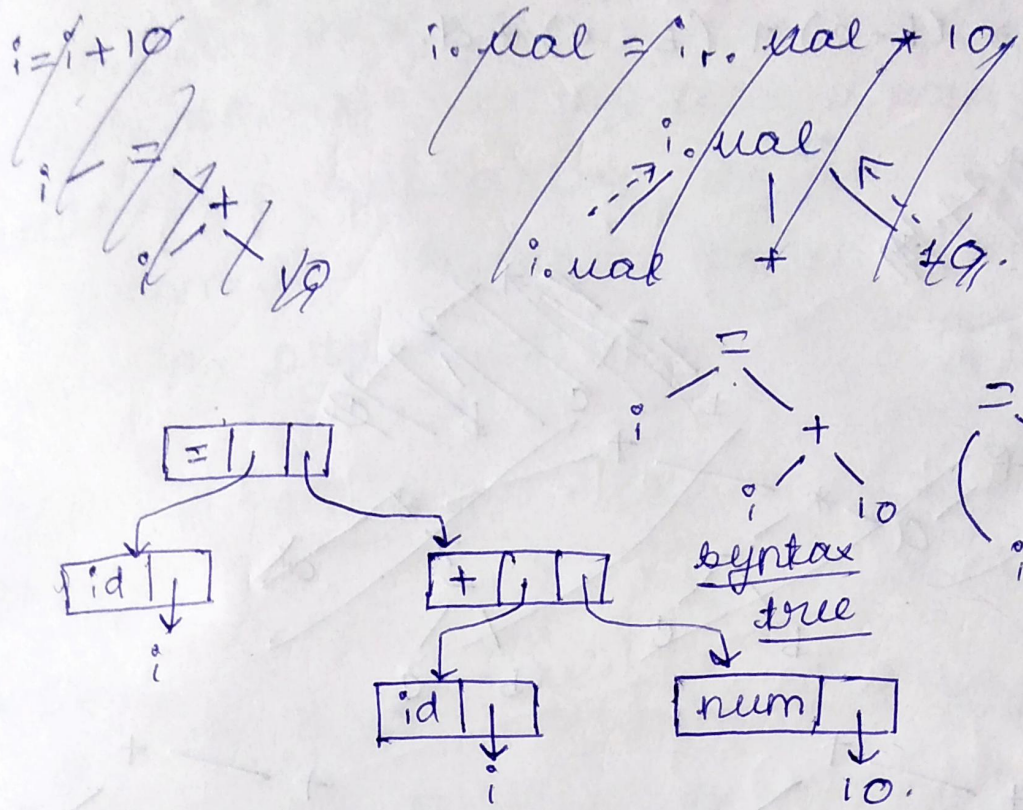
P_5 = mknode (+, P_3, P_4).



Syntax tree

17/10/22

Directed acyclic graph (DAG).

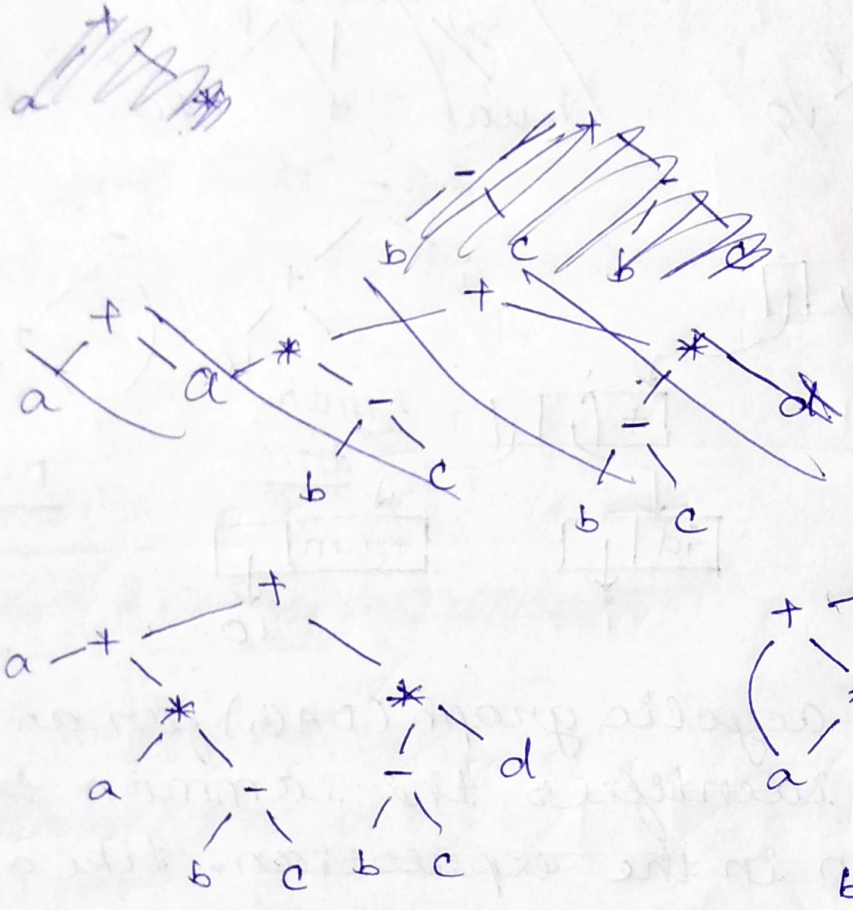


A directed acyclic graph (DAG) for an expression identifies the common sub-expression in the expression. Like a syntax tree, a DAG has a node for every sub-expression of the expression; and interior node represents an operator and its children represents its operators.

The difference is that a node in a DAG representing a common sub-expression has more than one parent in a syntax tree.

the common sub-expression would be represented as a duplicate sub tree.

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



DAG

Bottom-up evaluation of S-attribute definition

Synthesize attributes can be evaluated by a bottom-up parser as the input is being parsed. The parser can keep the

values of the synthesized attributes associated with the grammar symbol on its stack. Whenever a reduction is made, values of the new synthesized attributes are computed from the attributes appearing on the stack for the grammar symbol on the RHS of the reducing production.

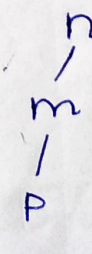
L → En
E → E, + T
E → T T → T, * F
T → F F → (E)
F → dig

Input	State	Val	Production used
3* 5+4n	—	—	
* 5+4n	3	3	
* 5+4n	F	3	F → digit
*5 + 4n	T	3	T → F.
5 + 4n	T*	3-	
+ 4n	T * 5	3-5	
+ 4n	T * F	3-5	F → digit
+ 4n	T	15	T → T * F.
4n	T+	15-	
n	T + 4	15_4	

n	T + F	15 _ 4	F → digit.
n	T + T	15 _ 4	T → F
n	E + T	15 - 4	E → T.
n	E	19	E → E + T
-	E n	19 -	
-	L	19 -	L → E n

Depth-First Evaluation Order for Attributes in a Parse tree → 18/10/22

```
dfvisit (n : node)
begin
    for each child m of n, from left to right DO
    begin
        Evaluate inherited attributes of m;
        dfvisit (m)
    END
    Evaluate synthesized attributes of n
END
```



L-Attributed definitions

A syntax directed definition is L-attributed if each inherited attribute of X_j, $1 \leq j \leq n$ on the right side of production $A \rightarrow X_1 X_2 \ldots X_n$, depends only on

i) the attributes of the symbol $X_1, X_2, \ldots X_{j-1}$ to the left of X_j on the production.

ii) the inherited attributes of A.

Translation scheme

A translation scheme is a CFG in which attributes are associated with the grammar symbol and semantic actions enclosed ~~within~~ between braces { } are inserted within the right side of the production.

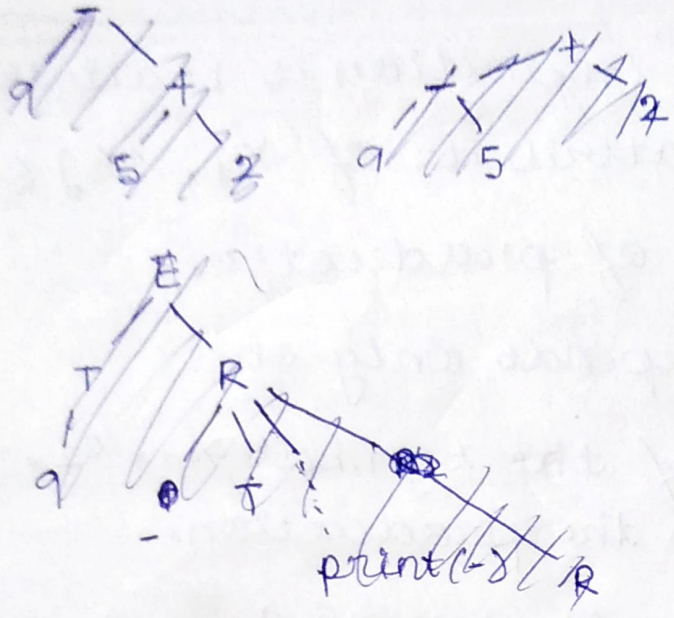
$E \rightarrow TR$

$R \rightarrow addop\ T\ \{ print(addop.lexeme) \} R_1 \mid \in$

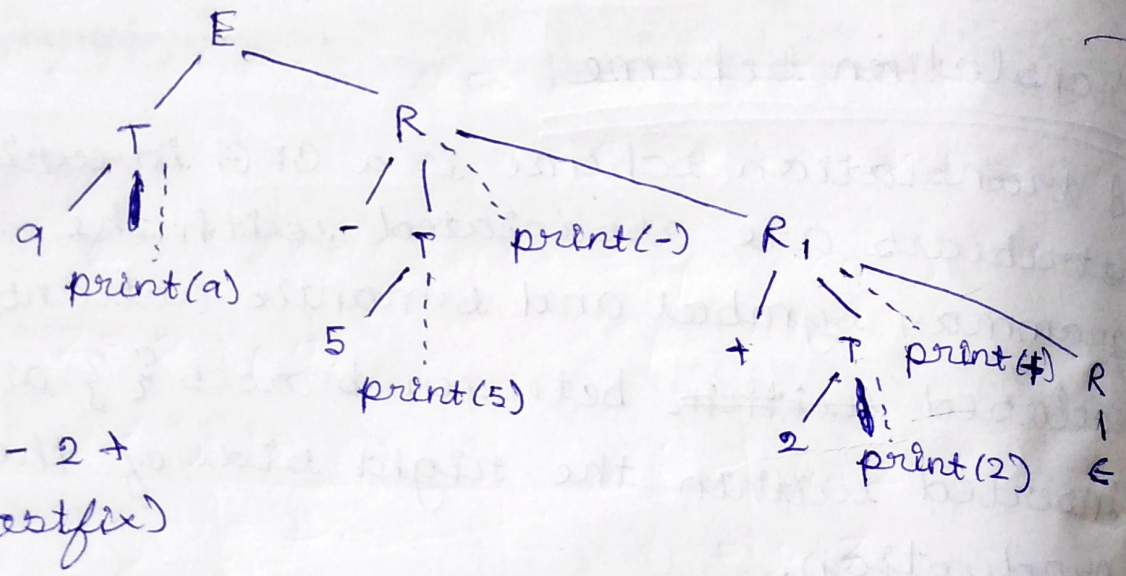
$T \rightarrow num\ \{ print(num.val) \}$.

$9-5+2$ $\quad 9+(-5)+2$.

Show action for infix to postfix using parse tree.



Infix input →
9 - 5 + 2



9 5 - 2 +
(postfix)

Bottom-up evaluation of Inherited Attributes

real p, q, r.

- D D → TL.

D → TL
T → int
T → real
L → L1 , id
L → id

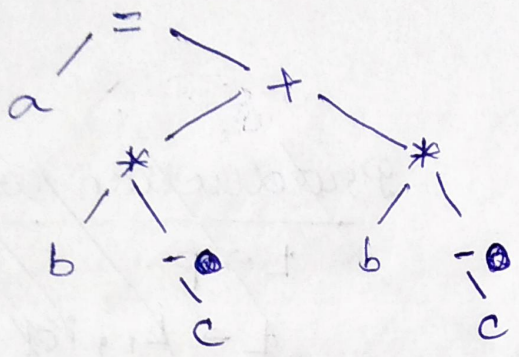
Input	State	Production used
real p, q, r	-	-
real L, q, r	real	T → real
real L, r	T	L → id
	TL	-
	TL,	-
	TL, q	L → L1 , id
	TL	-
	TL,	-
	TL, r	L → L1 , id
	TL	

Input	State	Production used
real p, q, r	-	-
p, q, r	real	T → real
p, q, r	T	L → id
, q, r	Tp	-
, q, r	TL	-
q, r	TL,	L → L1 , id
, r	TL, q	-
, r	TL	L → L1 , id
r	TL,	-
-	TL, r	L → L1 , id
-	TL	

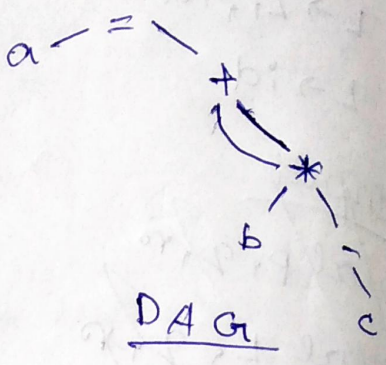
Intermediate Code Generation

21/10/22

$a = b * -c + b * -c$



Syntax tree



DAG

Postfix notation: ~~b - c * b - c * + =~~

$a\ b\ c\ -\ *\ b\ c\ -\ *\ +\ =$

Three address code (max. 3 operands, 2 operators.)

$z = x\ op\ y.$

eg. if ( ) then
goto L.

$a = x + y + z$

$t_1 = y + z.$

$a = x + t_1$

⇓

3-address code.

~~t_1 t_2~~

$t_1 = -c$

$t_2 = t_1 * b$

$t_3 = t_2 + t_1$

$a = t_3.$

~~Three address code.~~

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$a = t_2 + t_4$

~~a=t5~~

~~syntax tree~~

$t_1 = -c$

$t_2 = b * t_1$

$a = t_2 + t_2$

~~a=t5~~

DAG.

ICG (3-address code)

Types of 3-address code →

i) $Z = x$ op y / $x = y$ op z.

ii) $x =$ op y

iii) $x = y$

iv) goto L

v) if x relational operator y goto L
(relop)

vi) $x = y[i]$ / $x[i] = y$

vii) $x = \&y$; $x = *y$.

Implementation of 3 address statements →

A 3-address statement is an abstract form of intermediate code. In a compiler these statements can be implemented as records with fields

for the operators and operands.
3 such representations are ~~quadraples~~ quadruples, triples and indirect triples.

Quadruples → It is a record structure with 4 fields: (operator) op, arg1, arg2, result

a = b*-c + b*-c

Implementing using quadruple.

The op field contains an internal code for the operator. The three address statement x = y op z is represented by placing y in arg1, z in arg2 and x in result.

	op	arg1	arg2	result
(0)	-	c		t_1
(1)	*	b	t_1	t_2
(2)	-	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	a

	op	arg 1	arg 2
(0)	-	c	
(1)	*	b	(0)
(2)	-	c	
(3)	*	b	(2)
(4)	+	(3)	(1)

Triple representation

28/10/22

Indirect triple → Another implementation of 3-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.

	Statement
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15).

(10)
(11)
(12)
(13)
(14)
(15)

Basic Block and Flow Graph

A graph representation of 3-address statement is called a flow graph. It is useful for understanding code generation algorithm.

A basic block is a sequence of consecutive 3-address statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Partition-into-basic-blocks

Inputs – a sequence of 3-address statements.

Outputs → a list of basic blocks with each 3-address statement in exactly one block.

Method

1) We first determine the set of leaders, the first statement of basic block. The rules we are the following.

i) The first statement is a leader.

ii) Any statement that is the target of a conditional or unconditional goto is a leader.

iii) Any statement that immediately follows a goto or conditional goto statement is a leader.

2) For each leader its basic block consists of the leader and all statements upto but not including the next leader for the end of the program.

When each basic block is connected it forms a flow graph.

eg: 1) prod = 0

2) i = 1

iii) $t_1 = 4 * i$

iv) $t_2 = a[t_1]$.

v) $t_3 = 4 * i$

vi) $t_4 = b[t_3]$

vii) $t_5 = t_2 * t_4$

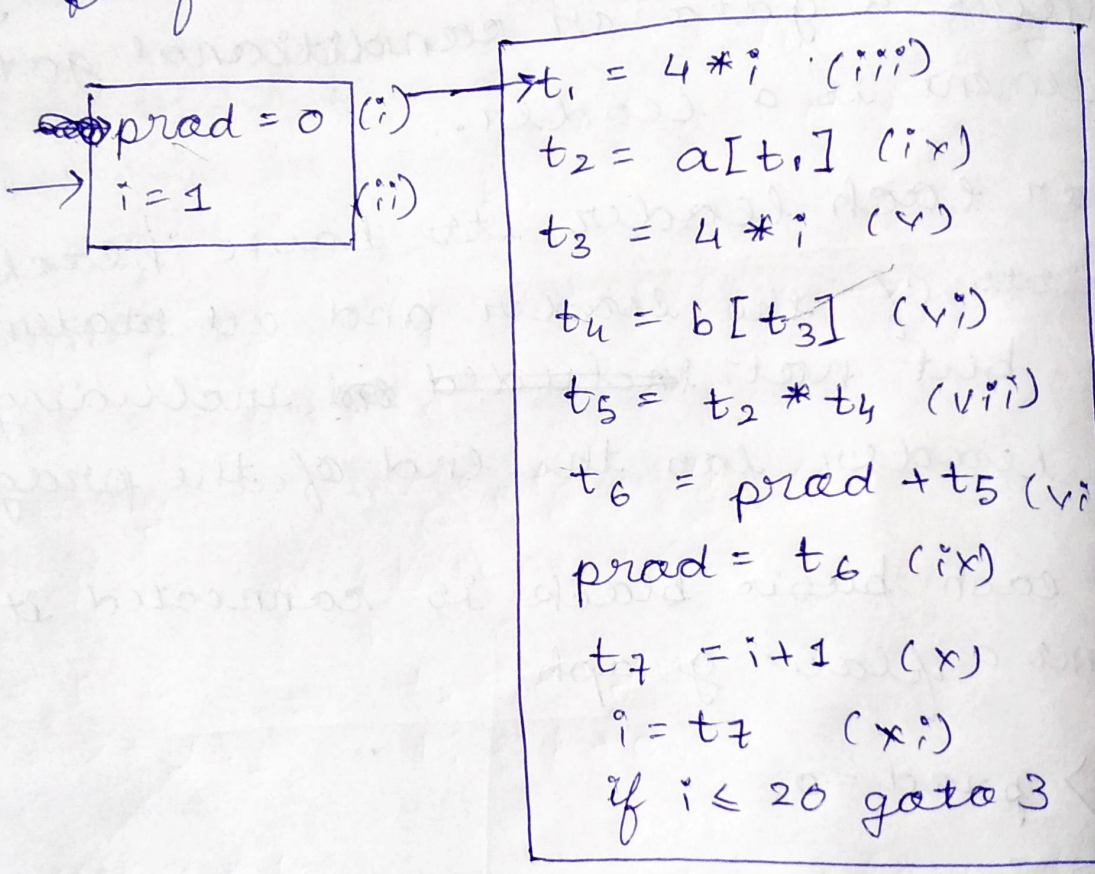
viii) $t_6 = prod + t_5$.

ix) $prod = t_6$

x) $t_7 = i + 1$

xi) $i = t_7$

xii) if $i \leq 20$ goto 3.



Transformation on basic blocks 1/11/22

A basic block computes a set of expressions. These expressions are the values of the names-life or exit from the block. Two bb are said to be equivalent if they compute the same set of expressions. A number of transformation can be applied to a bb without changing the set of expressions computed by the block. Many of these transformations are useful for improving the quality of code that will be generated from a bb. There are 2 important classes of local transformation that can be applied to bb. These are the structure preserving transformation and algebraic transformation.

1) Structure preserving transformation →
The primary SPT on bb are →

i) common subexpression elimination
ii) Dead code elimination
iii) Renaming of temporary variables
iv) Interchange of two independent adjacent statements.

```
a = b + c
b = a - d
c = b + c
d = a - d
```

Eliminating the sub-common expression,

```
a = b + c
b = a - d
c = b + c
d = b
```

Dead code elimination → Remove any expression that in not used further in the basic block.

Renaming of temp. variables →

$t = b + c \Rightarrow u = b + c$

Renaming variable t with u.

Interchange of statements →

$t_1 = b + c \quad \Rightarrow \quad t_2 = x + y$

$t_2 = x + y \qquad\qquad t_1 = b + c$

eg. $x = x * 1$, $x = x + 0$ ⤵ algebraic transformation.

$x = y ** 2$

$x = y * y$.

DAG representation of basic blocks

(1) $t_1 = 4 * i$

(2) $t_2 = a[t_1]$

(3) $t_3 = 4 * i$

(4) $t_4 = b[t_3]$

(5) $t_5 = t_2 * t_4$

(6) $t_6 = prod + t_5$

(7) $prod = t_6$

(8) $t_7 = i + 1$

(9) $i = t_7$

(10) if $i \leq 20$ goto (1)

DAGs are useful data structures for implementing transformation on basic blocks. A DAG gives a picture of how the value computed by each statement in basic block is used in subsequent statements of the block. Constructing a DAG from 3-address statements is a good way of determining common subexpression within a block, determining which names are used inside the block but evaluated outside the block, and determining which statements of the

block would have their computed value used outside the block.

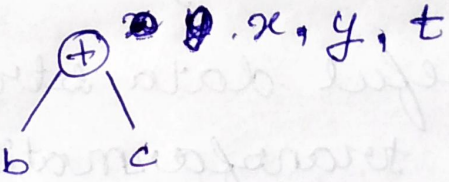
A DAG for a basic block is a directed acyclic graph with the following label of nodes —

i) leaves are labelled by unique identifiers, variable names or constants.

ii) interior nodes are labelled by an operator symbol.

iii) nodes are also optionally given a sequence of identifier for labels.

$x = b + c$
$y = x$
$t = y$



eg: