COMPILER DESIGN

M. Dheekshitha
CSE-B 3/4
17251A05B0

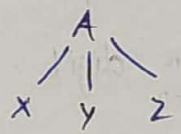
1. Define SDD. In what order does an evaluation of an SDD at the nodes of a parse tree is done with an example.

SDD or syntax directed definition is a kind of abstract definition and specification. It is an attribute grammer where each grammer production is associated with a set of production rules or semantic rules. These semantic rules compute the attribute values of the grammar symbols. Parse tree thus formed from the attribute values is called Annotated parse tree. There are 2 kinds of attributes for non-terminals.

1. Synthesized Attribute - This is associated with the production at N, which is a parse node for a non-terminal A where production must have A as it's head. A synthesized attribute is defined only in terms of attribute values at the children of N and itself.

Eg: $A \to XYZ$

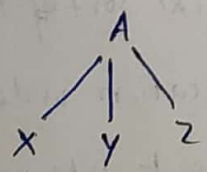
if $A.S = f(X.S \mid Y.S \mid Z.S)$



'.S' is called synthesized attribute

2. Inherited Attribute - This attribute have the value of a inherited attribute is computed from the values of attribute at the siblings and parent of that node.

Eg:



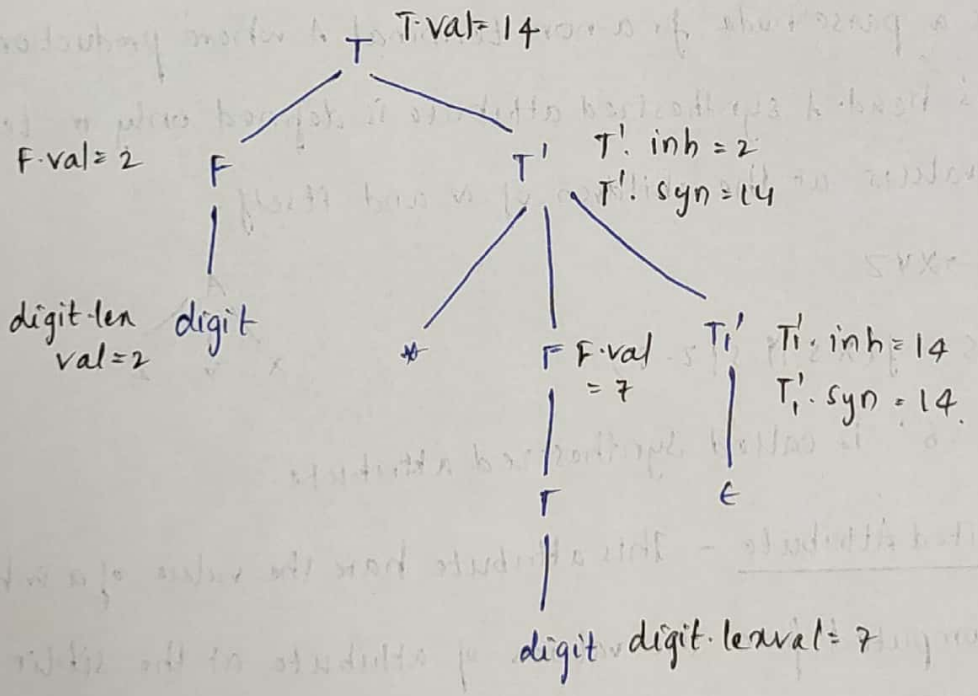
if $Y.i = f(X.i \mid Z.i \mid A.i)$

then '.i' is called inherited attribute

Eg: Let us take the example of 2 * 7

Production	Semantic rules
1. $T \to FT'$	$T'.inh = F.val$ $T.val = T'.syn$
2. $T' \to *FT_1'$	$T_1'.inh = T'.inh * F.val$ $T'.syn = T_1'.syn$
3. $T' \to \epsilon$	$T'.syn = T'.inh$
4. $F \to digit$	$F.val = digit.lexval$

It starts with production $T \to FT'$ where F generates the digit 2 and T' generates *.



2. Explain dependency graph in detail. Describe topological sort of the graph

1. There are many useful tools to determine an evaluation order for the attribute instances in a given parse tree. This helps us to determine values of attribute can be computed. It depicts the flow of information among the attribute instances in a particular parse tree an edge from one

attribute instance to another means value of first is needed to compute that of second.

Rules are -

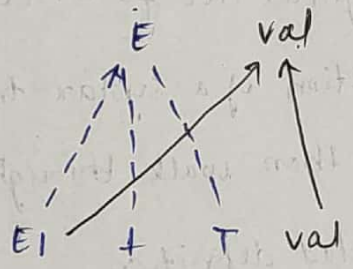
(1) For every node labelled by grammer symbol X, the dependency graph has a node for each of its attributes

(2) If a semantic rule for production 'p' defines the value of synthesized attributes A.b in terms of A.c, then there will be an edge

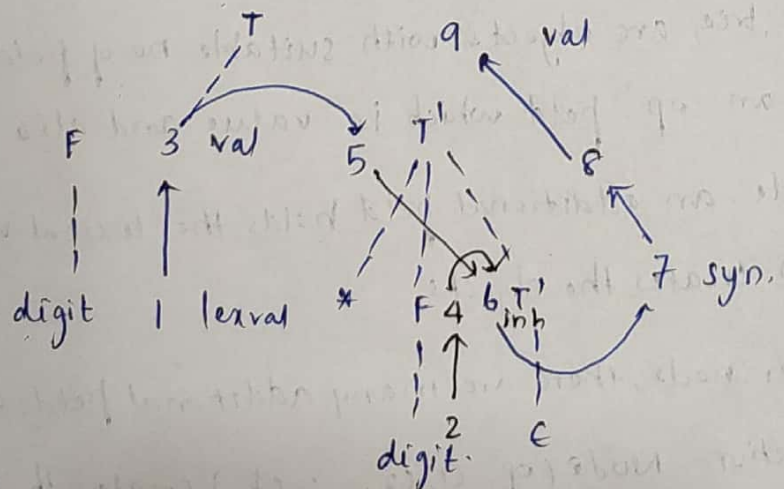
(3) If a semantic rule for production 'p' defines the value of an inherited attribute B.c in terms of X.a, then there will be an edge.

Eg:

Production	Rule
$E \rightarrow E1 + T$	$E.val = E1.val + T.val$



2) 3 * 5



Topological sort: If the dependency graph has an edge from node m to n, 'm' must be evaluated first and allowable sequences are $n_1, n_2, \ldots n_k$ such that if there is an edge of the dependency graph n_i to n_j such that $i<j$ such an order is called a topological sort.

If there is a cycle in the graph, there is no topological sort. If there are no cycles, there is atleast 1 topological sort.

For the second example in above,

1) 1, 2, 3, 4, 5, 6, 7, 8, 9
2) 1, 3, 5, 2, 4, 6, 7, 8, 9

are 2 topological sorts as there is no cycle with it.

3. What are different applications of syntax-Directed Translation? Write steps in the construction of syntax tree for a+(c-4) using a proper SDT.

SDT application is the construction of a syntax tree. Some compiler turn the input string into a tree. Some then walk through it. We consider 2 SDD's

- Top-Down parsing / L-attributed definition
- Bottom-up parsing / S-attributed definition

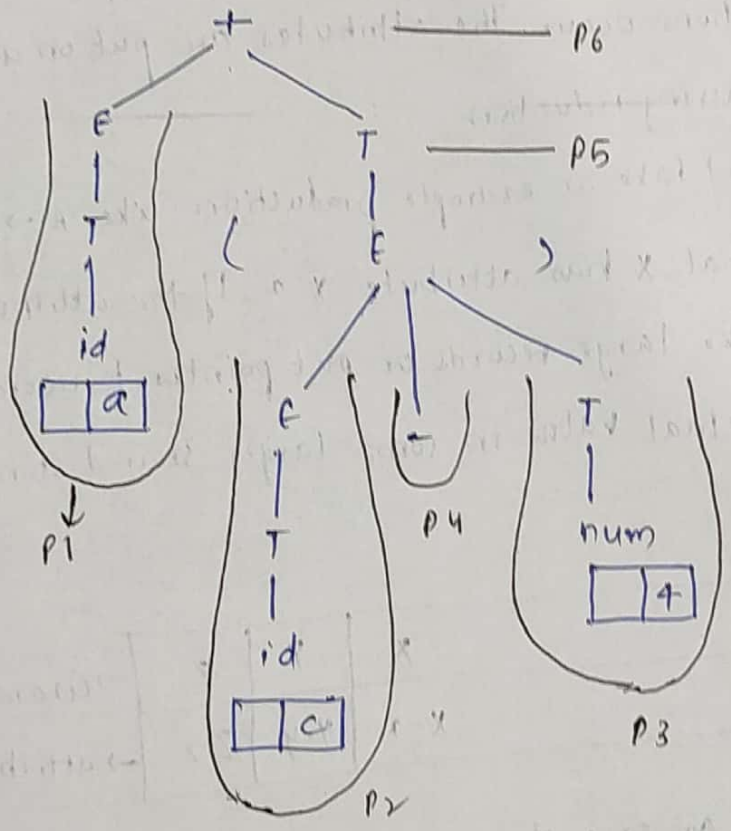
The nodes of a syntax tree are objects with suitable no of fields. Each object will have an op field which is value and also

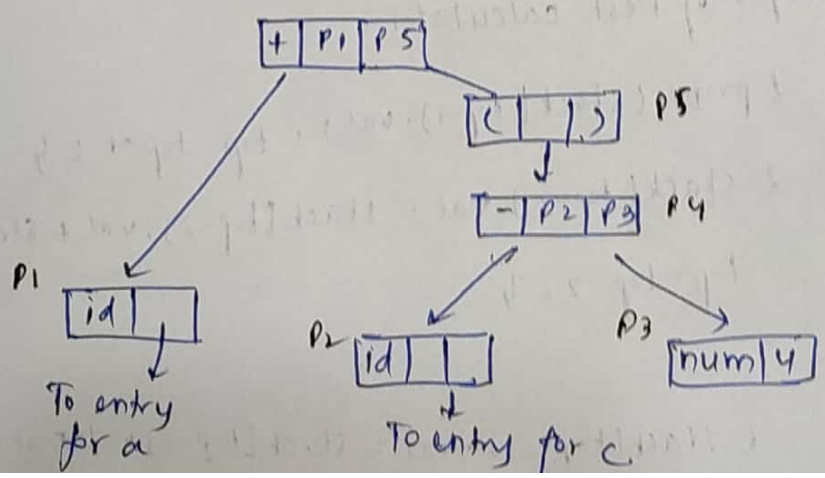
① If node is a leaf node, an additional field holds the lexical value. The function leaf(op, val) creates the objects.

② If node is an interior node, there are many additional fields as nodes have children. The function Node(op, c1, c2, ..., ck) creates the object.

For a+(c-4):

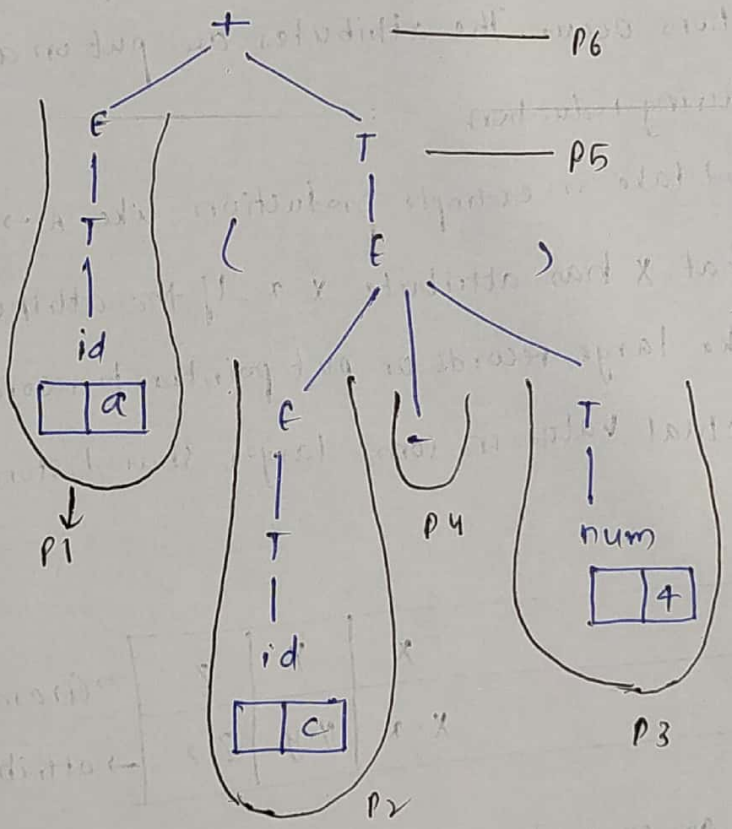
Grammer	Product Rules
$E \to E+T$	E.node = new Node('+', E.node, T.node)
$E \to E-T$	E.node = new Node('-', E.node, T.node)
$E \to T$	E.node = T.node
$T \to (E)$	T.node = E.node
$T \to id$	T.node = new Leaf(id, id.entry)
$T \to num$	T.node = new Leaf(num, num.val)

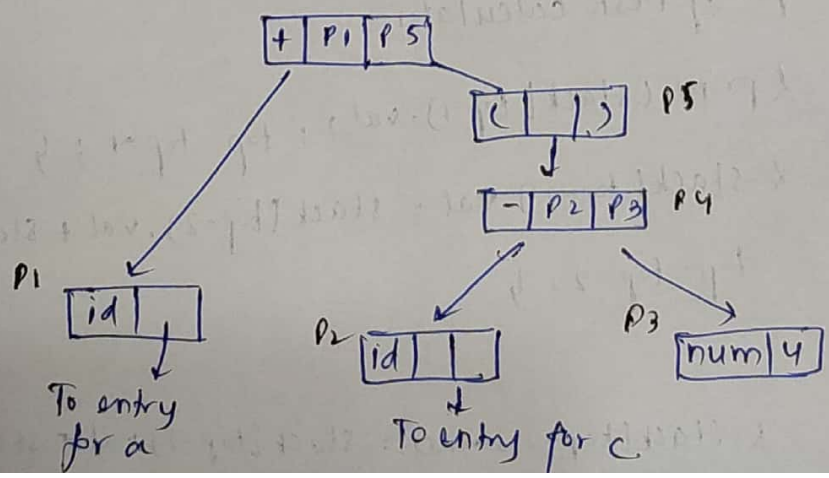




For a+(c-4):

Grammer	Product Rules
$E \to E+T$	E.node = new Node('+', E.node, T.node)
$E \to E-T$	E.node = new Node('-', E.node, T.node)
$E \to T$	E.node = T.node
$T \to (E)$	T.node = E.node
$T \to id$	T.node = new Leaf(id, id.entry)
$T \to num$	T.node = new Leaf(num, num.val)





top = top-2; }

T → F

F → (E) { stack[top-2].val = stack[top-1].val;
top = top-2; }

F → digit

For E → E1 + T, we will go 2 positions below the top and get the value of E1 and we find value of T at the top. The result E is placed at the top. For E → T, no action is necessary, because the length of stack T → F and F → digit doesnot change.

5 Write short notes on

① Eliminating left recursion from SDT's

When the grammar is a part of SDT, we need to worry about how actions are handled. Rules are:

① When transforming the grammar, treat the actions as if they were terminals. The actions are therefore executed in same order.

$$A \rightarrow A\alpha \mid \beta$$

that generate strings consisting of a β and any number of α's and replace them using a new non-terminal R

$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R \mid \epsilon$$

Eg: E → E1 + T { print ('+'); }
E → T

Now, α = + T { print ('+'); }

Applying left recursion, we get E → TR
R → + T { print ('+'); } R
R → ϵ

② SDT's for L-attributed definition in the case of WHILE statement.

Rules:

a) Embed the action that computes the inherited attribute for a non-terminal A immediately before that occurance of A in the body of production

b) Place the actions that compute a synthesized attribute for the head of the production at the end of the body of the production

SDD for while statements:

S → While (C) S1

L1 : newL()

L2 : newL()

S1.next = L1

C.false = S.next

C.true = L2

S.code = label||L1||C.code||label||L2||S1.code

Now, to convert to SDT, we have to handle L1 and L2 as they are not attributes:

S → while ({L1=newL(); L2=newL(); C.false = S.next; C.true = L2;}

C) {S1.next=L1;}

S1 {S.code = label||L1||C.code||label||L2||S1.code;}

---

③ SDT's with Actions Inside Productions

An action can be placed at any position within the body of a production.

For eg: B → X {a;} Y, the action here is done after we have recognized X.

If parse is:

bottom-up: a is performed on the occurence of X on top of stack.

top-down: we perform a just before we attempt to expand the occurance

of Y or check for Y on the input.

Eg: Postfix SDT's, L-attributed definition of SDT's.

If we take the example of an extreme SDT,

1) $L \rightarrow En$
2) $E \rightarrow E\{print('+');\}E1+T$
3) $E \rightarrow T$
4) $T \rightarrow T\{print('*');\}T1*F$
5) $T \rightarrow F$
6) $F \rightarrow (E)$
7) $E \rightarrow digit\{print(digit.lexval);\}$

Not possible to implement the SDT because critical actions like + or * must be performed long before it knows whether these symbols will appear in the input.

Any SDT can be implemented as:

a) Ignoring the actions, parse the input and produce parse tree
b) Then, examine each interior node N, say one for production $A \rightarrow \alpha$. Add additional children to N for the actions in α
c) Perform a preorder traversal of the tree.