

SYNTAX - DIRECTED DEFINITION (SDD)

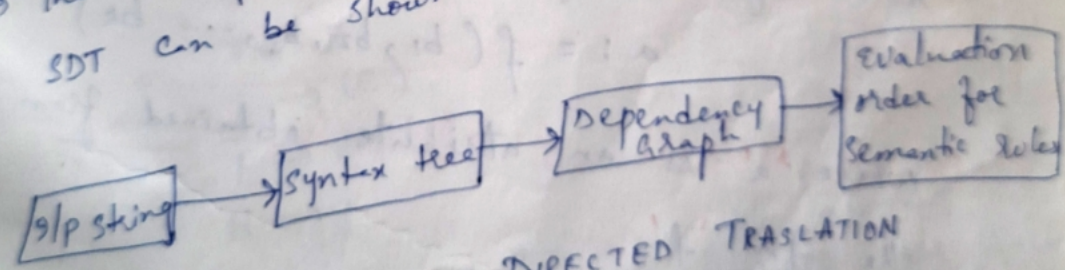
→ while doing the static analysis of the lang we use Syntax-Directed Definitions. that means an Augmented CFG is generated.

→ In other words the set of attributes are associated with each terminal & non-terminal symbols.

→ the attribute can be a string, a no, a type, a memory location or anything else.

→ SDD is a kind of abstract specification.

→ The conceptual view of Syntax Directed Translation SDD can be shown as



SYNTAX - DIRECTED TRANSLATION

→ The implementation need not have to follow all the steps.

→ In single pass implementation semantic rules can be evaluated during parsing w/o explicitly constructing a parse tree, or dependency graph.

→ In such semantic evaluation, at the nodes of the syntax tree, values of the attribute are defined for the given s/p string.

Such a parse tree containing the values of attributes at each node is called an Annotated or Decorated parse tree.

SDD Definition

SDD is a generalization of CFG in which each grammar production

$$X \rightarrow \alpha$$

is associated with a set of semantic rules of the form

$$a := f(b_1, b_2, b_3, \dots, b_k)$$

where 'a' is an attribute obtained from the fn 'f'.

Then there are two types of Attributes:

→ SYNTHESIZED ATTRIBUTE

→ INHERITED "

SYNTHESIZED ATTRIBUTE (S-attribute)

→ The attribute 'a' is called synthesized attribute of 'X' and $b_1, b_2, \dots, b_k$ are attributes belonging to the production symbols.

→ Synthesized Attribute is also called as S-attribute.

→ The value of S-attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

eg:- Consider the CFG as

$S \rightarrow EN$

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

$N \rightarrow ;$

Start $\rightarrow$ print

non-terminal $\rightarrow$.val

digit $\rightarrow$.lexval

The SDD can be written for the above grammar by using semantic actions for each production.

Start symbol has to print the value
Non-terminal Associate with .val
terminal " with .lexval

; ignored by lex analyzer.

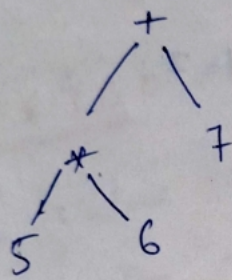
Production rule	Semantic Actions
$S \rightarrow EN$	$\text{print}(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} := E_1.\text{val} + T.\text{val}$
$E \rightarrow E_1 - T$	$E.\text{val} := E_1.\text{val} - T.\text{val}$
$E \rightarrow T$	$E.\text{val} := T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} := T_1.\text{val} * F.\text{val}$
$T \rightarrow T_1 / F$	$T.\text{val} := T_1.\text{val} / F.\text{val}$
$T \rightarrow F$	$T.\text{val} := F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} := E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} := \text{digit}.\text{lexval}$
$N \rightarrow ;$	can be ignored by Lexical Analyzer as ; is terminating symbol.

STEPS TO COMPUTE S-attribute definition

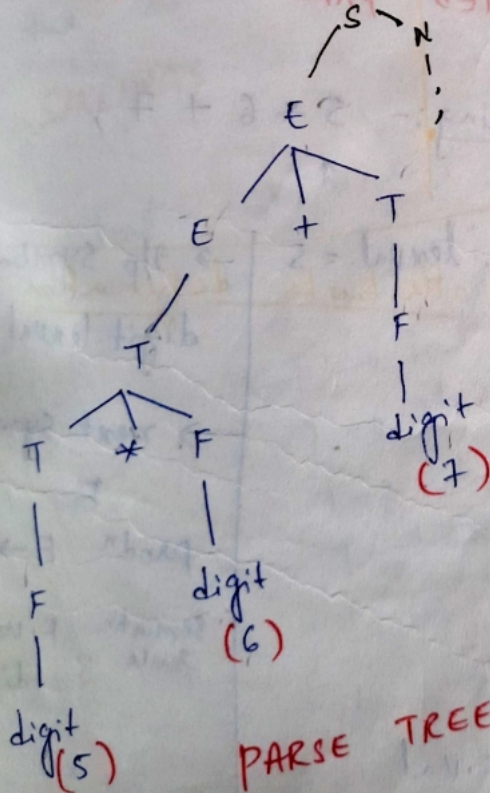
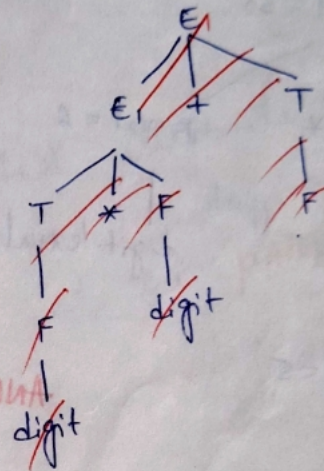
1. Write the SDD using the appropriate semantic actions for corresponding production rule of the given grammar.
2. The Annotated parse tree is generated & attribute values are computed.
the computation is done in Bottom-up manner.

3. the value obtained at the root node is supposed to be the final o/p.

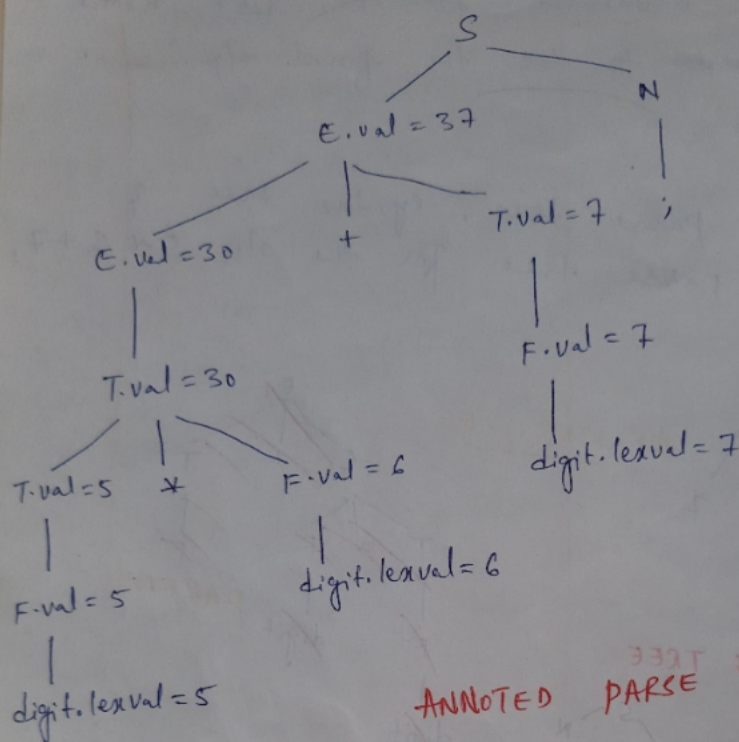
Ex:- Construct parse tree, syntax tree & Annotated parse tree for the exp $5 * 6 + 7$;
 postfix



SYNTAX TREE



PARSE TREE



bottom
up
fashion

ANNOTATED PARSE TREE

Computation

9/p String :- 5 * 6 + 7 ;

→ the 9/p is 5, digit.lexval = 5

→ $F \rightarrow \text{digit}$
 $F.val := \text{digit.lexval}$
 $:= 5$

→ next symbol T
 production $T \rightarrow F$
 semantic rule $T.val := F.val$
 $:= 5$

→ 9/p symbol 6
 digit.lexval = 6

→ next symbol F
 prodⁿ $F \rightarrow \text{digit}$
 semantic rule $F.val := \text{digit.lexval}$

→ next symbol T
production $T \rightarrow T * F$

semantic Action $T.val := T.val * F.val$

$= 5 * 6$
 $= 30$

→ proceed same upto S.

✓ INHERITED

ATTRIBUTE

→ the value of inherited attribute at a node in a parse tree is defined using the attribute values at the **PARENT** or **SIBLINGS**.

int a, b, c;

$S \rightarrow TL;$

$T \rightarrow \text{int}$

$T \rightarrow \text{float}$

$T \rightarrow \text{char}$

$T \rightarrow \text{double}$

$L \rightarrow \text{id}$

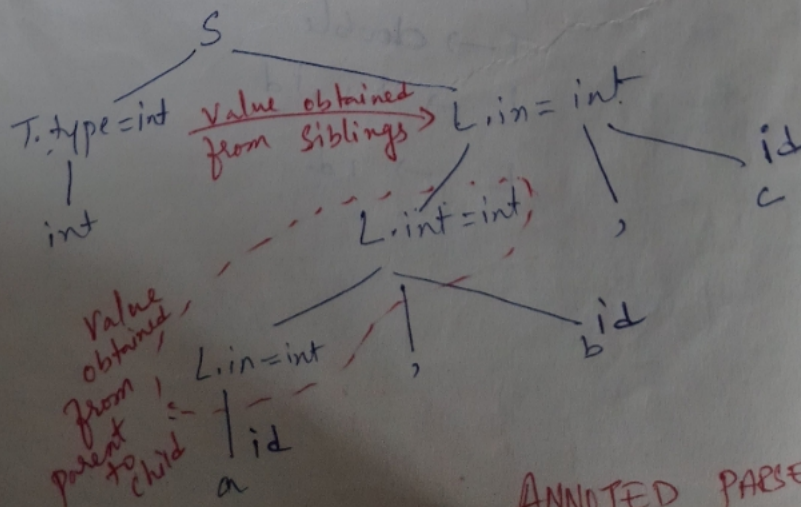
$L \rightarrow \text{id}$

STEPS

- ### STEPS
1. Construct the SDD using semantic action
 2. Annotate the parse tree with inherited attributes by processing in top-down fashion.

Production Rule	Semantic Actions
$S \rightarrow TL ;$	$L.in := T.type$
$T \rightarrow int$	$T.type := integer$
$T \rightarrow char$	$T.type := char$
$T \rightarrow float$	$T.type := float$
$T \rightarrow double$	$T.type := double$
$E \rightarrow L, id$	$L_1.in := L.in$
$L \rightarrow id$	Enter-type (id.entry, L.in)
	Enter-type (id.entry, L.in)

SDD

 $\nexists a, b, c;$ 

ANNOTATED PARSE TREE

$S \rightarrow TL$

T value as int or float or char or double
L value obtained from T.type (siblings).

DEPENDENCY GRAPH

→ the directed graph that represents the interdependencies b/w synthesized & inherited attributes at nodes in the parse tree is called DEPENDENCY GRAPH.

→ For the rule $X \rightarrow YZ$
the semantic action is $X.x := f(Y.y, Z.z)$

then
Synthesized attribute is $X.x$ &
 $X.x$ depends upon attributes
 $Y.y$ & $Z.z$

Ex:- Design the dependency graph for the following grammar

$$E \rightarrow E_1 + E_2$$

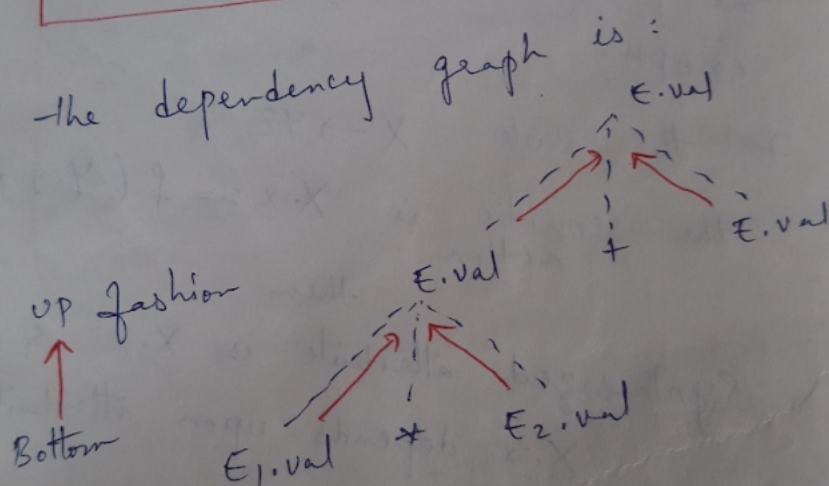
$$E \rightarrow E_1 * E_2$$

Q1

The semantic role for the grammar is

Production rule	Semantic role
$E \rightarrow E_1 + E_2$	$E.val := E_1.val + E_2.val$
$E \rightarrow E_1 * E_2$	$E.val := E_1.val * E_2.val$

the dependency graph is :



→ Synthesized attributes can be represented by .val

→ Dependencies among the nodes is given by SOLID ARROWS

→ Parse tree represented by DOTTED LINES

CONSTRUCTION OF SYNTAX TREES

→ the Syntax tree is an abstract repr of the lang constructs.

→ the Syntax trees are used to write the translation routines using SDD.

ex:-

$$E \rightarrow E_1 + T$$

$$E \rightarrow E_1 - T$$

$$E \rightarrow E_1 * T$$

$$E \rightarrow T$$

$$T \rightarrow id$$

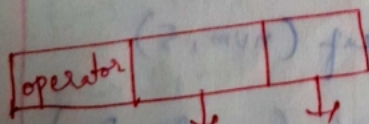
$$T \rightarrow num$$

→ Constructing Syntax tree for an Expr means translation of Expr into postfix form.

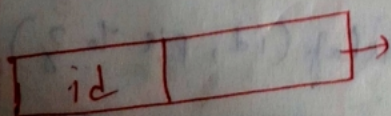
→ the nodes for each operator & operand is

Created. It can be implemented as a record

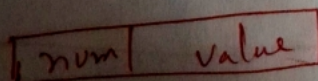
→ Each node can be implemented as a record with multiple fields.



$mknode(op, left, right)$



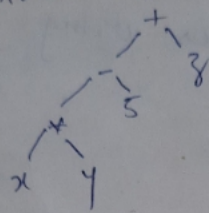
$mkleaf(id, entry)$



$mkleaf(num, val)$

Construct the syntax tree for the expr
 $x * y - 5 + z$ for the above
 grammar.

Sol step 1:- Convert the expr into postfix



$xy * 5 - z +$

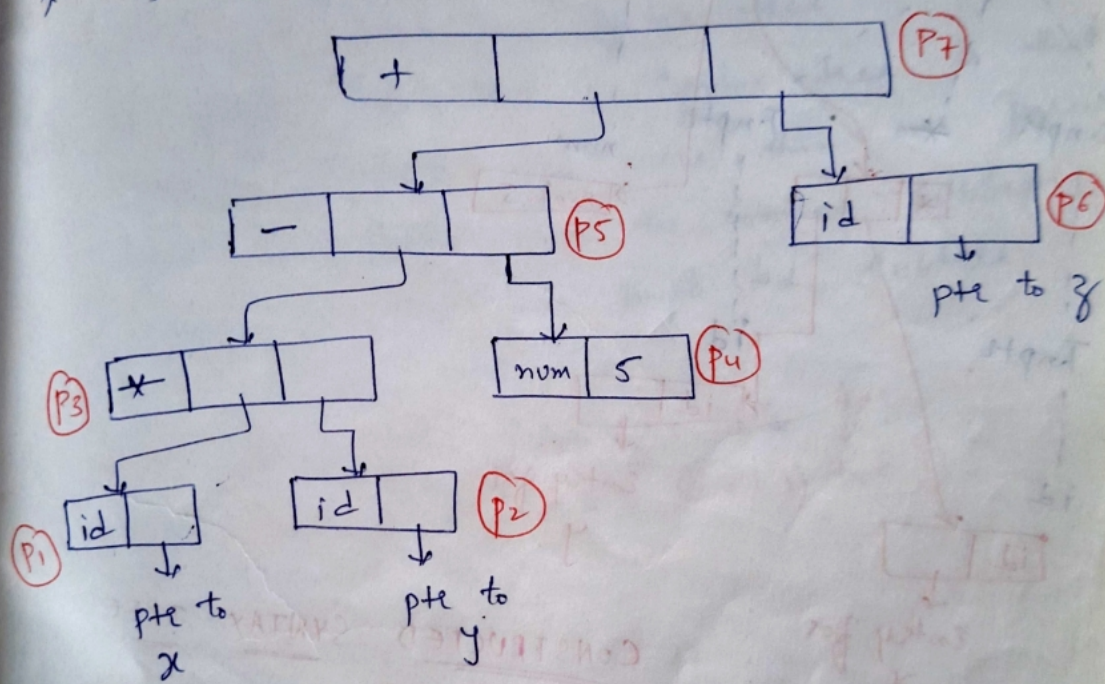
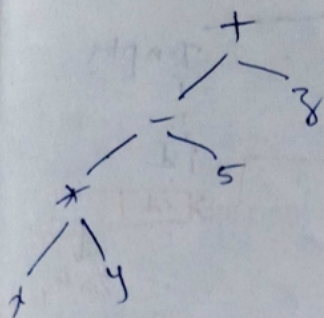
step 2:- make use of the fns
 $mknode()$, $mkleaf(id, ptr)$ & $mkleaf(num, val)$

step 3:- the sequence of calls given.

POSTFIX EXPRESSION $xy * 5 - z +$

Symbol	operation
x	$P_1 = mkleaf(id, ptr \text{ to } x)$
y	$P_2 = mkleaf(id, ptr \text{ to } y)$
*	$P_3 = mknode(*, P_1, P_2)$
5	$P_4 = mkleaf(num, 5)$
-	$P_5 = mknode(-, P_3, P_4)$
z	$P_6 = mkleaf(id, ptr \text{ to } z)$
+	$P_7 = mknode(+, P_5, P_6)$

CONSIDER THE STRING $x * y - 5 + z$
 DRAW THE SYNTAX TREE.



SYNTAX TREE

SDD for ABOVE GRAMMAR

Production Rule	Semantic operation
$E \rightarrow E_1 + T$	$E.npte := mknode(+, E_1.npte, T.npte)$
$E \rightarrow E_1 - T$	$E.npte := mknode(-, E_1.npte, T.npte)$
$E \rightarrow E_1 * T$	$E.npte := mknode(*, E_1.npte, T.npte)$
$E \rightarrow T$	$E.npte := T.npte$
$T \rightarrow id$	$T.npte := mkleaf(id, id.pt-entry)$
$T \rightarrow num$	$T.npte := mkleaf(num, num.val)$

DIR



CONSTRUCTED SYNTAX TREE

DIRECTED ACYCLIC GRAPH FOR EXPRESSION (DAG)

→ the Directed Acyclic Graph usually referred as DAG is a directed graph drawn by identifying the common subexpr.

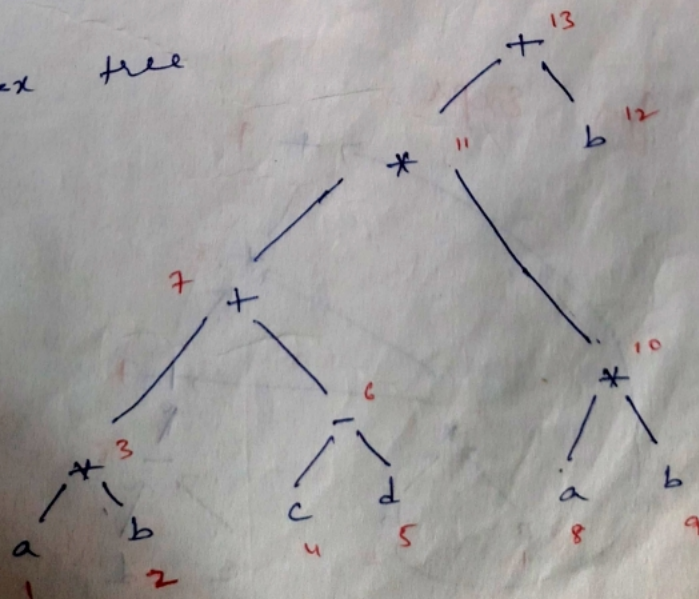
→ DAG is same as syntax tree

→ the diff b/w DAG & Syntax tree is that common subexpr has more than one parent and in syntax tree the common subexpr would be repr as duplicated subtree.

Ex:- DRAW the syntax tree & DAG for the expr $((a * b) + (c - d)) * (a * b) + b$

Sol → the postorder traversal is $ab * cd - + ab * b +$

→ Syntax tree



Sequence of operations for syntax tree is:

$$P_1 = \text{mkleaf}(\text{id}, a)$$

$$P_2 = \text{mkleaf}(\text{id}, b)$$

$$P_3 = \text{mknode}(*, P_1, P_2) \rightarrow (a * b)$$

$$P_4 = \text{mkleaf}(\text{id}, c)$$

$$P_5 = \text{mkleaf}(\text{id}, d)$$

$$P_6 = \text{mknode}(-, P_4, P_5) \rightarrow (c - d)$$

$$P_7 = \text{mknode}(+, P_3, P_6) \rightarrow (a * b) + (c - d)$$

$$P_8 = \text{mkleaf}(\text{id}, a)$$

$$P_9 = \text{mkleaf}(\text{id}, b)$$

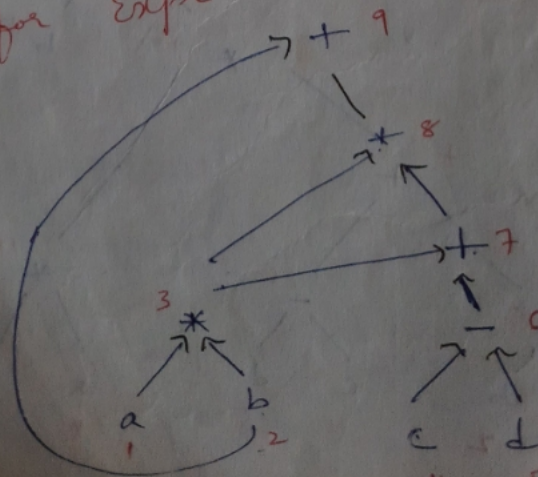
$$P_{10} = \text{mknode}(*, P_8, P_9) \rightarrow (a * b)$$

$$P_{11} = \text{mknode}(*, P_7, P_{10}) \rightarrow ((a * b) + (c - d)) * (a * b)$$

$$P_{12} = \text{mkleaf}(\text{id}, b)$$

$$P_{13} = \text{mknode}(+, P_{11}, P_{12}) \rightarrow (((a * b) + (c - d)) * (a * b)) + b$$

DAG for Expr



sequence of operations for DAG

$$p_1 = \text{mkleaf}(\text{id}, a)$$

$$p_2 = \text{mkleaf}(\text{id}, b)$$

$$p_3 = \text{mknode}(*, p_1, p_2) \rightarrow (a * b)$$

$$p_4 = \text{mkleaf}(\text{id}, c)$$

$$p_5 = \text{mkleaf}(\text{id}, d)$$

$$p_6 = \text{mknode}(-, p_4, p_5) \rightarrow (c - d)$$

$$p_7 = \text{mknode}(+, p_3, p_6) \rightarrow ((a * b) + (c - d))$$

$$p_8 = \text{mknode}(*, p_7, p_3) \rightarrow (((a * b) + (c - d)) * (a * b))$$

$$p_9 = \text{mknode}(+, p_8, p_2) \rightarrow$$

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

Bottom-up evaluation of S-attribute Definition

- S attribute definition is one such class of SDD with synthesized attributes only.
- the purpose of stack is to keep track of values of the synthesized attributes associated with the grammar symbol on its stack. this stack is commonly known as **PARSER STACK**

Eg:- $X \rightarrow ABC$

	States	Value
top-2	A	A.a
top-1	B	B.b
top	C	C.c

Before Reduction

⇒

States	Value
X	X.x

After reduction

pushed in reverse order

Eg:- For the following grammar construct the SDD & generate the code fragment (Translator) using S-attribute definition.

$S \rightarrow EN$

$E \rightarrow E+T$

$E \rightarrow E-T$

$E \rightarrow \text{~~EA~~ T}$

$T \rightarrow T * F$

$T \rightarrow T / F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$ $N \rightarrow ;$

g/p string:-

$2 * 3 + 4;$

Sol The SDD can be written as

production	Semantic Actions
$S \rightarrow EN$	print (E.val)
$E \rightarrow E + T$	$E.val := E.val + T.val$
$E \rightarrow E - T$	$E.val := E.val - T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T * F$	$T.val := T.val * F.val$
$T \rightarrow T / F$	$T.val := T.val / F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$ () ignored by lexical analyzer
$F \rightarrow \text{digit}$	$F.val := \text{digit.lexval}$
$N \rightarrow ;$	; ignored by lexical analyzer

To Evaluate the attributes the CODE FRAGMENT can be generated by using PARSER STACK.

production	Code Fragment
$S \rightarrow EN$	print (value[top])
$E \rightarrow E + T$	$value[top] := value[top-2] + value[top]$
$E \rightarrow E - T$	$value[top] := value[top-2] - value[top]$
$E \rightarrow T$	no code fragment, as T tends to further grammar production
$T \rightarrow T * F$	$value[top] := value[top-2] * value[top]$

$T \rightarrow T / F$

$\text{value}[\text{top}] := \text{value}[\text{top}-2] / \text{value}[\text{top}]$

$T \rightarrow F$

no code fragment, as F tends to further grammar production

$F \rightarrow (E)$

$\text{value}[\text{top}] := \text{value}[\text{top}-1]$

$()$ ignored

$F \rightarrow \text{digit}$

no code fragment, digit value directly read by lex analyzer

$N \rightarrow ;$

no code fragment ; ignored

usual pd example ()

the sequence of moves made by the parser for the i/p $2 * 3 + 4 ;$

i/p string	state	value	production
$2 * 3 + 4 ;$	-	-	-
$* 3 + 4 ;$	2	2	-
$* 3 + 4 ;$	F	2	$F \rightarrow \text{digit}$
$* 3 + 4 ;$	T	2	$T \rightarrow F$
$3 + 4 ;$	$T *$	2	-
$+ 4 ;$	$T * 3$	2, 3	$F \rightarrow \text{digit}$
$+ 4 ;$	$T * F$	2, 3	$T \rightarrow T * F$
$+ 4 ;$	T	6	-
$+ 4 ;$	E	6	$E \rightarrow T$

4;	E+	6	-
;	E+4	6,4	F → digit
;	E+F	6,4	T → F
;	E+T	6,4	E → E+T
;	E	10	-
-	E;	10	N → ;
-	EN	10	S → EN
-	S	10	-

SYMBOL TABLE ORGANIZATION ✓

DATA STRUCTURE OF SYMBOL TABLE ✓ type check

BLOCK STRUCTURE & NON BLOCK STRUCTURE

Symbol TablesDefinition:-

- Symbol table is a DS used by compiler to keep track of semantics of variables. That means symbol table stores the inf about scope & binding inf about names.

- Symbol table is built in lexical & syntax analysis phases.
- Semantic analysis phase refers symbol table for type conflict issue.
- Code generation refers symbol table knowing how much run-time space is allocated?
- L-value & R-value
 left right
 $a := i + 1$
 l-value r-value

Symbol - Table Entries

- to achieve compile time efficiency compiler makes use of symbol table.
- it associates lexical names with their attributes.
- the items to be stored in symbol table are:

Var names

Constants

procedure names

fn names

literal constants & strings

Compiler generated temporaries

labels in source lang

- Compiler uses following types of info from Symbol table

data type

declaring procedures

if struct or record then ptr to

struct table

for parameters, whether parameter

passing is by value or by reference.

no. & type of arguments passed

to the fn.

Base addr.

Organization of Symbol Table

The organization of symbol tables for non-block structured lang is by ordered or unordered manner.

Ordered Symbol Table

- In ordered symbol table the entries of var is made in alphabetical manner.
- The searching of ordered symbol table can be done using LINEAR & BINARY SEARCH.

Adv

- The searching of particular var is efficient.
- The relationship of particular var with another can be established very easily.

Disadv

- Insertion of ele in ordered list is costly if there are large no. of entries in symbol table.

Unordered Symbol Table

- As variable is encountered, then its entry is made in Symbol Table.
- these entries are not in sorted manner.
- before inserting any var, a lookup is made to check whether it is already present in the symbol table or not.

Adv

Insertion of var in Symbol table is very efficient.

Disadv

For large table size, this method turns to be inefficient before insertion a lookup is made.

Name Representation

- (a) Fixed-length name
- (b) Variable-length name
- (a) Size fixed for all names.
if name small then wastage of space
- (b) the name can be stored with the help of starting index & length of each name.

DATA STRUCTURE OF SYMBOL TABLE

LIST

self-organizing list/linked list

Binary trees

Hash tables

LIST DS for Symbol Table

- linear list is a simplest kind of mechanism to implement the symbol table.
- In this method an array is used to store names and associated info. (8)
- new names can be added in the order as they arrive.
- the pointer "Available" is maintained at the end of all stored records.

Name1	info1
Name2	info2
Name3	info3
⋮	⋮
Name n	info n

AVAILABLE →
(start of empty slot)

- to retrieve the info about some name we start from beginning of array and go on searching up to available pointer.

if we reach at pointer available w/o³ finding a name we get an error "Use of Undeclared name".

→ while inserting a new name we should ensure that it should not be already there. If it is there another error occurs. i.e. "Multiple Defined Name".

→ The advantages of list organization is that it takes min amount of space.

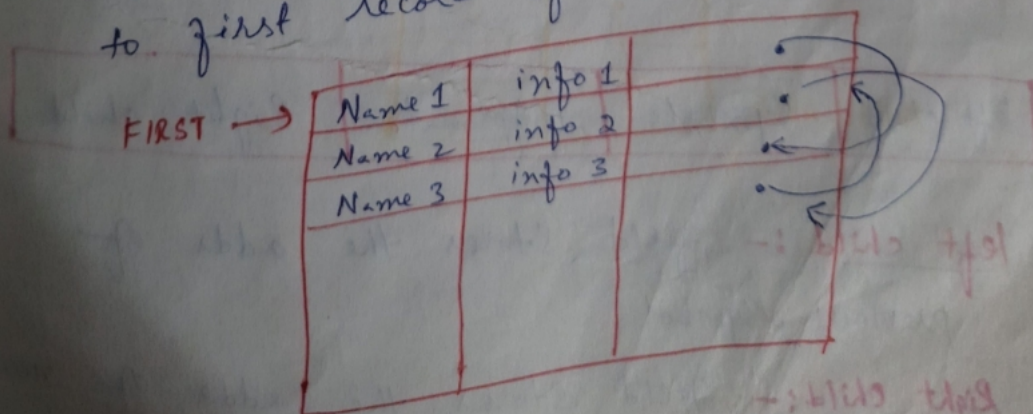
SELF - ORGANIZING LIST / linked list ⁽⁹⁾

This symbol table implementation is using linked list.

A link field is added to Each record.

→ we search the records in the order pointed by the link of link field.

→ A pointer "FIRST" is maintained to point to first record of the symbol table.



The reference to these names can be
Name 3, Name 1, Name 4, Name 2.

→ When the name is referenced or created
it is moved to the front of the list.

→ the most frequently referred names will tend
to be front of the list.

Hence Access time to most frequently
referred names will be the least.

Advantages of List

1. The list organization takes less space.
2. Insertion of new symbol is easy.

BINARY TREES

→ when the organization symbol table is by
means of binary tree,
the node structure will be as follows:

left child	Symbols	information	Right child
------------	---------	-------------	-------------

left child :- field stores the add of
previous symbol

Right child :- field stores the add of next
symbol.

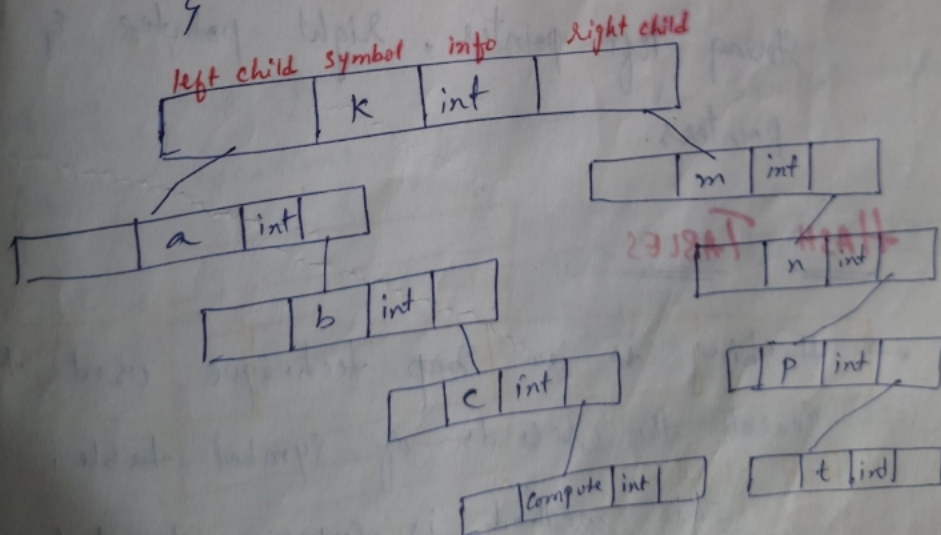
Symbol:- field is used to store the name of the symbol.

information:- field is used to give inf about the symbol.

eg:- Create a tree structure using symbol table organization.

```
int m, n, p;  
int compute (int a, int b, int c)  
{  
    t = a + b * c;  
    return (t);  
}
```

```
main()  
{  
    int k;  
    k = compute (10, 20, 30);  
}
```



Tree structure organization

→ the binary tree structure is basically a Binary Search tree in which the value of left node is always less than the value of parent node.

Similarly, the value of right node is always more or greater than the parent node.

Adv

1. Insertion of any symbol is efficient.
2. Any symbol can be searched efficiently using binary search method.

(12)

Disadv

1. this structure consumes lot of space in storing left pointer, right pointer & null pointers.

HASH TABLES

→ Hashing is an imp technique used to search the records of symbol table.

this method is superior to list organization

→ In hashing scheme two tables are maintained a Hash table & Symbol table.

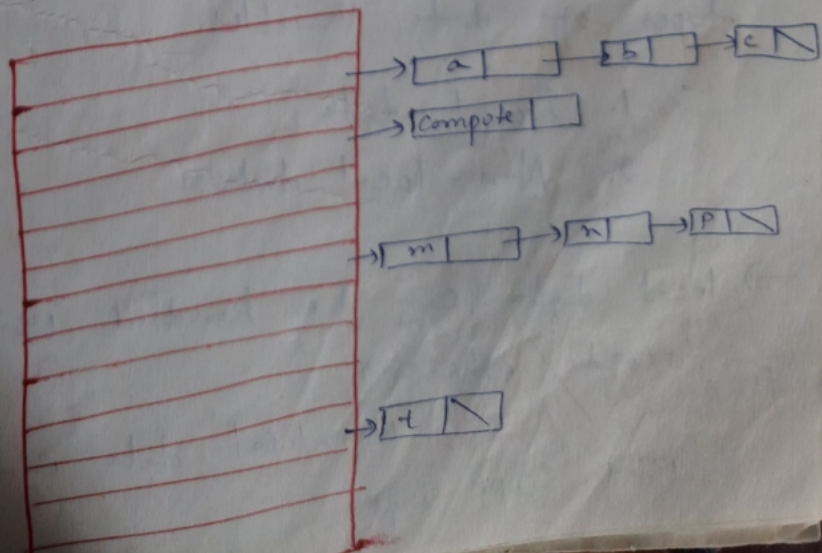
→ The Hash table consists of k entries from $0, 1$ to $k-1$.

these entries are basically pointers to Symbol table pointing to the names of Symbol table.

→ to determine whether the "Name" is in Symbol table, we use a hash fn " h " such that $h(\text{name})$ will result any integer b/w 0 to $k-1$, (13)
we can search any name

by position = $h(\text{name})$
→ using this position we can obtain the exact locations of names in Symbol table.

Previous
ex code



→ The hash fn should result in Uniform distribution of names in Symbol table.

→ The hash fn should be such that there will be min no. of Collision.

→ Various Collision resolution techniques are open-addressing, chaining, rehashing.

→ The advantages of hashing is quick Search is possible and the disadvantage is that hashing is complicated to implement.

Some xtra space is required obtaining Scope of var is very difficult

BLOCK STRUCTURE & NON-BLOCK STRUCTURE

The Storage allocation can be done for two types of data variables. ①

1. Local data
2. Non-local data.

→ local data can be handled using Activation record whereas

Non-local data can be handled using Scope info.

→ the block structured storage allocation can be done using static scope or lexical scope.

→ the Non-block structured storage allocation can be done using Dynamic Scope.

(2)

LOCAL DATA

→ the local data can be accessed with the help of activation record.

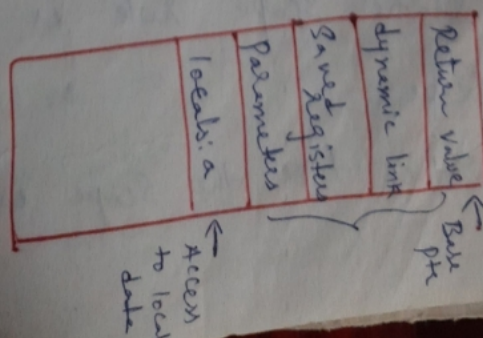
→ the offset relative to base pointer of an activation record points to local data variables within an activation record.

Hence

Reference to any variable x in procedure =
Base ptr pointing to start of procedure
+ offset of var x from base ptr.

Eg. - procedure A
int a;

the contents of stack along with base ptr & offset x shown as:



Access to Non-local Names

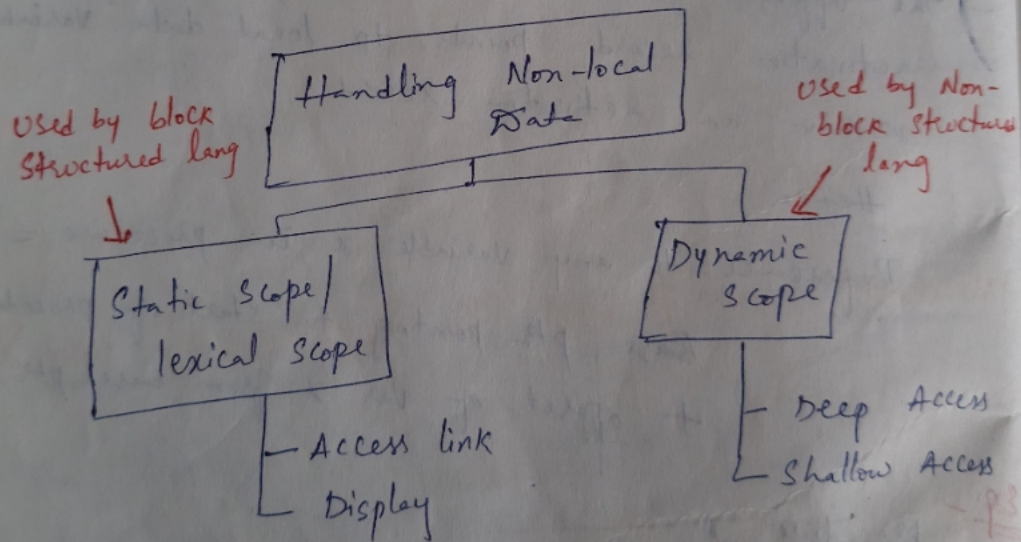
→ A procedure may sometimes refer to variables which are not local to it. Such variables are called as Non-local variables.

→ For the Non-local names there are two types of scope rules that can be defined:

Static

(2)

Dynamic



Static Scope rule

→ The Static Scope rule is also called as lexical scope.

→ In this type the scope is determined by

Examining the program text.

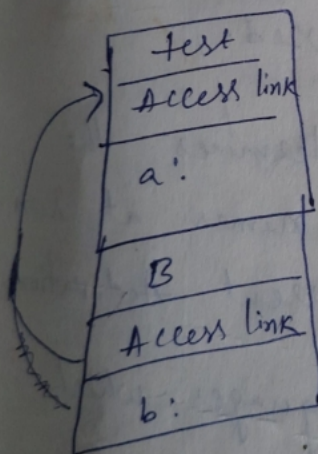
→ PASCAL, C, ADA are the lang that use the 'Static Scope rule.

→ these lang are also called as Block structured languages.

Access Link

→ the Implementation of lexical scope can be obtained by using pointer to each Activation Record.

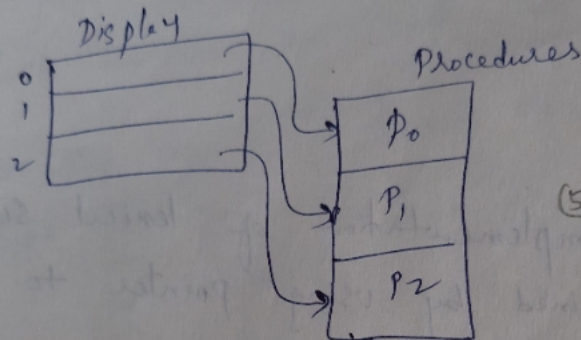
→ these pointers are called Access links.



Display

→ It is expensive to traverse down access link every time when a particular non-local variable is accessed.

→ to speed-up the access to non-locals can be achieved by maintaining an array of pointers called Display.



Dynamic Scope Rule

→ for non-block structured lang this dynamic scope allocation rules are used.

→ the dynamic scope rule determines the scope of declaration of the names at run-time by considering the current activation.

→ LISP & SNOBOL are the languages which use dynamic scope rule.

Storage Allocation for Non-Block structured languages.

Dynamic Scoping

→ In Dynamic Scoping a use of non-local variable refers to the non-local data declared in most recently called, still active procedure.

→ therefore each time new bindings are set up for local names called procedure.

→ In dynamic scoping symbol tables can be required at run-time. (6)

Deep Access
Shallow Access

Deep Access

→ the idea is keep a stack of Active Variables, use ctrl links instead of Access links & when v want to find a var then search the stack from top-to-bottom looking for most recent Activation Record that contains the space for desired var.

→ In this method a symbol table needs to be used at run-time.

Shallow Access

→ the idea is to keep a central storage with one slot for every variable name.

→ if the names are not created at run-time then that storage layout can be fixed at compile time otherwise when new activation of procedure occurs, then that procedure changes the storage entries for its locals at Entry & Exit. ⑦

Deep Access

- takes longer time to access non-locals

- no overhead

- needs a symbol table at run-time

Shallow Access

- Fast access

- overhead of handling procedure Entry & Exit

- symbol table not needed.

PDF Created Using



Camera Scanner

Easily Scan documents & Generate PDF



<https://play.google.com/store/apps/details?id=photo.pdf.maker>