Module 3 – Semantic Analysis

Syntax directed definitions

- Syntax directed definition is a generalization of context free grammar in which each grammar symbol has an associated set of attributes.
- The attributes can be a number, type, memory location, return type etc….
- Types of attributes are:
    1. Synthesized attribute
    2. Inherited attribute

E.Value
Memory location
Return Type
Type

Syntax Directed Translation Scheme

- The Syntax directed translation scheme is a context free grammar
- It is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the production.
- The position at which the action to be executed is shown by enclosed between braces.

Synthesized attributes

- Value of synthesized attribute at a node can be computed from the value of attributes at the children of that node in the parse tree.
- A syntax directed definition that uses synthesized attribute exclusively is said to be S-attribute definition.
- Example: Syntax directed definition of simple desk calculator

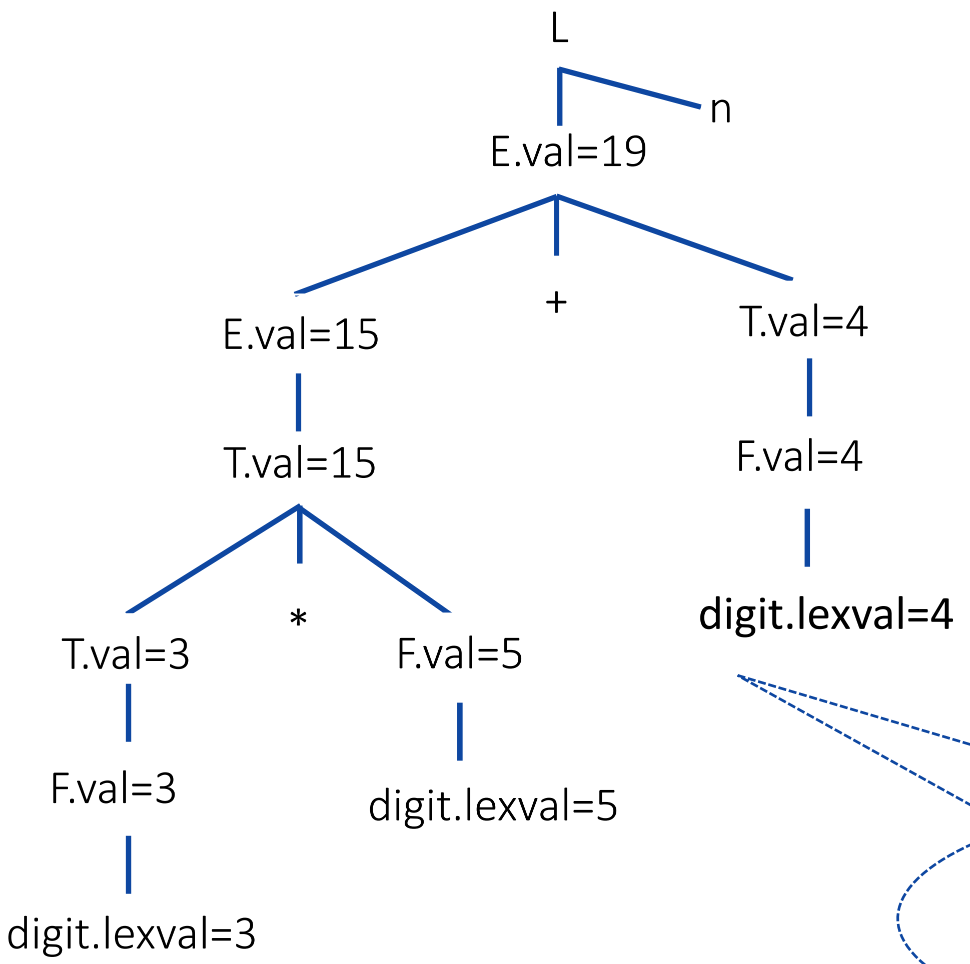
Production	Semantic rules
$L \rightarrow E_n$	
$E \rightarrow E_1+T$	
$E \rightarrow T$	
$T \rightarrow T_1*F$	
$T \rightarrow F$	
$F \rightarrow (E)$	
$F \rightarrow digit$	

Applications of SDT

- Executing Arithmetic Expression
- Conversion from Infix to Postfix
- Conversion from Infix to Prefix
- Conversion from Binary to Decimal
- Counting No. of Reductions
- Creating Syntax Tree
- Generating Intermediate Code
- Type Checking
- Storing Type information into Symbol Table.

Example: Synthesized attributes

String: 3*5+4n;



Annotated parse tree for 3*5+4n

The process of computing the attribute values at the node is called annotating or decorating the parse tree

parse tree showing the value of the attributes at each node is called Annotated parse tree

Production	Semantic rules
L → E$_n$	Print (E.val)
E → E$_1$+T	E.Val = E$_1$.val + T.val
E → T	E.Val = T.val
T → T$_1$*F	T.Val = T$_1$.val * F.val
T → F	T.Val = F.val
F → (E)	F.Val = E.val
F → digit	F.Val = digit . lexval

Exercise

Draw Annotated Parse tree for following:

1. 7+3*2n
2. (3+4)*(5+6)n

Syntax directed definition to translates arithmetic expressions from infix to prefix notation

Production	Semantic rules
L→E	
E→E+T	
E→E-T	
E→T	
T→T*F	
T→T/F	
T→F	
F→F^P	
F→P	
P→(E)	
P→digit	

Inherited attribute

- An inherited value at a node in a parse tree is computed from the value of attributes at the parent and/or siblings of the node.

Production	Semantic rules
D → T L	
T → int	
T → real	
L → L_1 , id	
L → id	

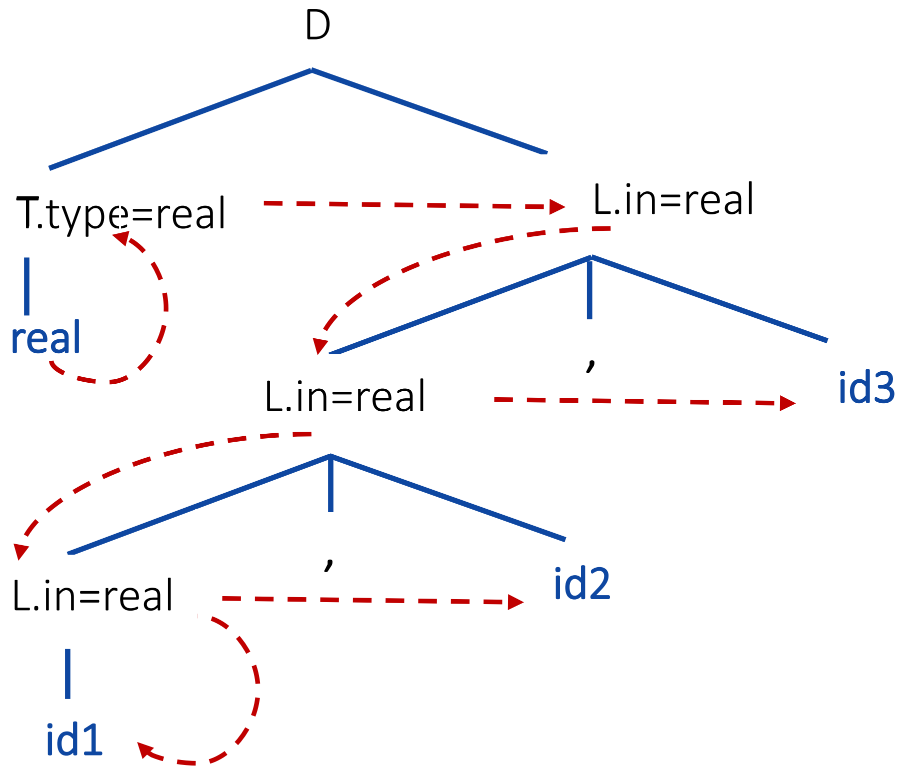
Syntax directed definition with inherited attribute L.in

- Symbol T is associated with a synthesized attribute type.
- Symbol L is associated with an inherited attribute in.

Example: Inherited attribute

Example: Pass data types to all identifier
real id1,id2,id3

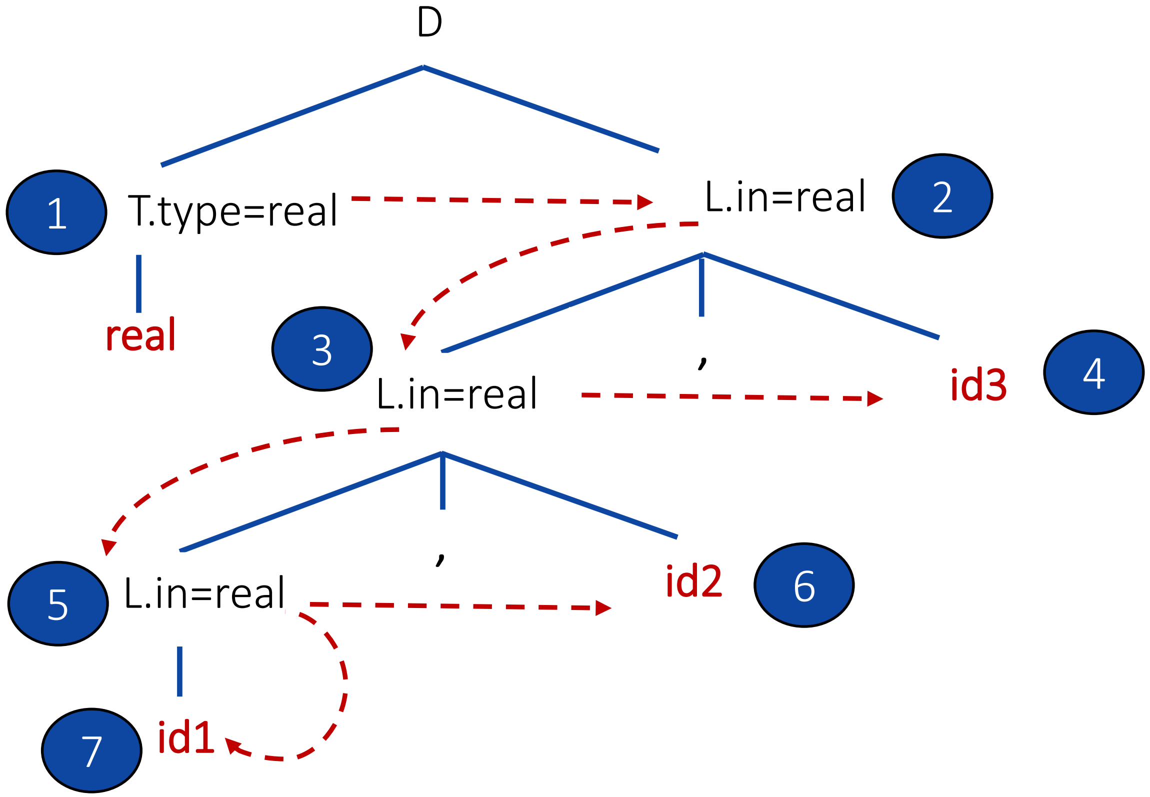
Production	Semantic rules
D → T L	L.in = T.type
T → int	T.type = integer
T → real	T.type = real
L → L_1 , id	L_1.in = L.in, addtype(id.entry,L.in)
L → id	addtype(id.entry,L.in)



D → T L
L → L_1, id

Evaluation order

- A topological sort of a directed acyclic graph is any ordering $m_1, m_2, \ldots \ldots \ldots, m_k$ of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.
- If $m_i \rightarrow m_j$ is an edge from m_i to m_j then m_i appears before m_j in the ordering.



Construction of syntax tree

- Following functions are used to create the nodes of the syntax tree.
  1. **Mknode (op,left,right):** creates an operator node with label op and two fields containing pointers 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 for the identifier.
  3. **Mkleaf (num, val):** creates a number node with label num and a field containing val, the value of the number.

Construction of syntax tree for expressions

Example: construct syntax tree for a-4+c

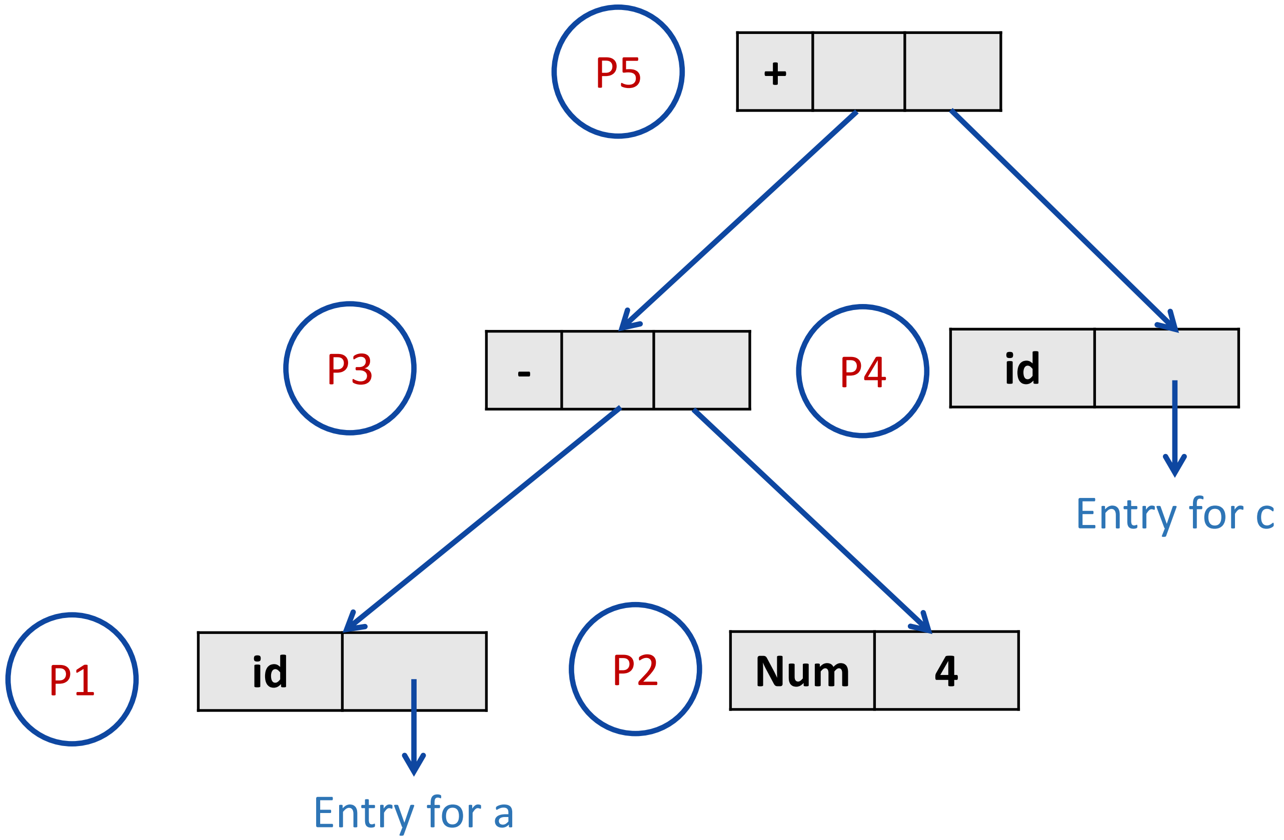
P1: mkleaf(id, entry for a);

P2: mkleaf(num, 4);

P3: mknode('-',p1,p2);

P4: mkleaf(id, entry for c);

P5: mknode('+',p3,p4);



Bottom up evaluation of S-attributed definitions

- S-attributed definition is one such class of syntax directed definition with synthesized attributes only.
- Synthesized attributes can be evaluated using bottom up parser only.

Synthesized attributes on the parser stack

- Consider the production A→XYZ and associated semantic action is A.a=f(X.x, Y.y, Z.z)

	State	Value
top-2 →	X	$X.x$
top-1 →	Y	$Y.y$
top →	Z	$Z.z$

Before reduction

	State	Value
top →	A	$A.a$

After reduction

Bottom up evaluation of S-attributed definitions

Production	Semantic rules
L → E$_n$	Print (val[top])
E → E$_1$+T	val[top]=val[top-2] + val[top]
E → T	
T → T$_1$*F	val[top]=val[top-2] * val[top]
T → F	
F → (E)	val[top]=val[top-2] - val[top]
F → digit	

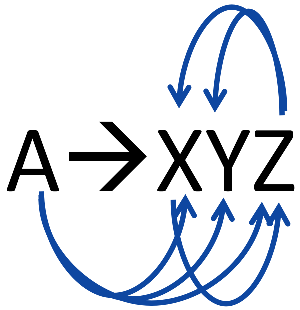
Implementation of a desk calculator with bottom up parser

Input	State	Val	Production Used
3*5**n**	-	-	

Move made by translator

L-Attributed definitions

- A syntax directed definition is L-attributed if each inherited attribute of X_j, $1 <= j <= n$, on the right side of $A \rightarrow X_1, X_2 \ldots X_n$ depends only on:
    1. The attributes of the symbols $X_1, X_2, \ldots X_{j-1}$ to the left of X_j in the production and
    2. The inherited attribute of A.
- Example:



L-Attributed ✓

Not L-Attributed ✗

Production	Semantic Rules
A→ LM	L.i:=l(A.i) M.i=m(L.s) A.s=f(M.s)
A→ QR	R.i=r(A.i) Q.i=q(R.s) A.s=f(Q.s)

- Above syntax directed definition is *not L-attributed* because the inherited attribute Q.i of the grammar symbol Q depends on the attribute R.s of

Bottom up evaluation of S-attributed definitions

- Translation scheme is a context free grammar in which attributes are associated with the grammar symbols and semantic actions enclosed between braces { } are inserted within the right sides of productions.
- Attributes are used to evaluate the expression along the process of parsing.
- During the process of parsing the evaluation of attribute takes place by consulting the semantic action enclosed in { }.
- A translation scheme generates the output by executing the semantic actions in an ordered manner.
- This process uses the depth first traversal.

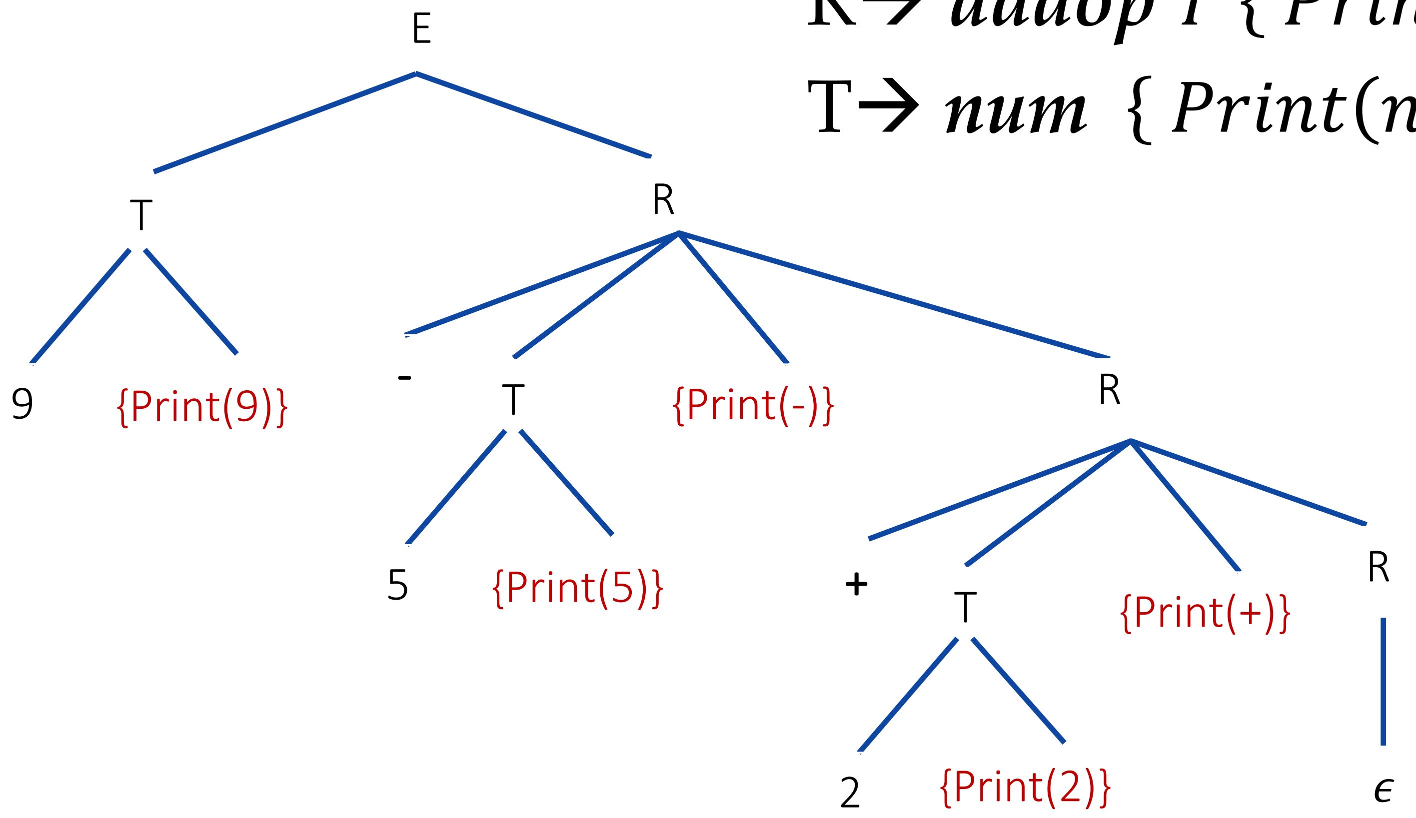
Example: Translation scheme (Infix to postfix notation)

String: 9-5+2

E→TR

R→ $\boldsymbol{addop}\ T\ \{\ Print(addop.lexeme)\}$ R1 | ϵ

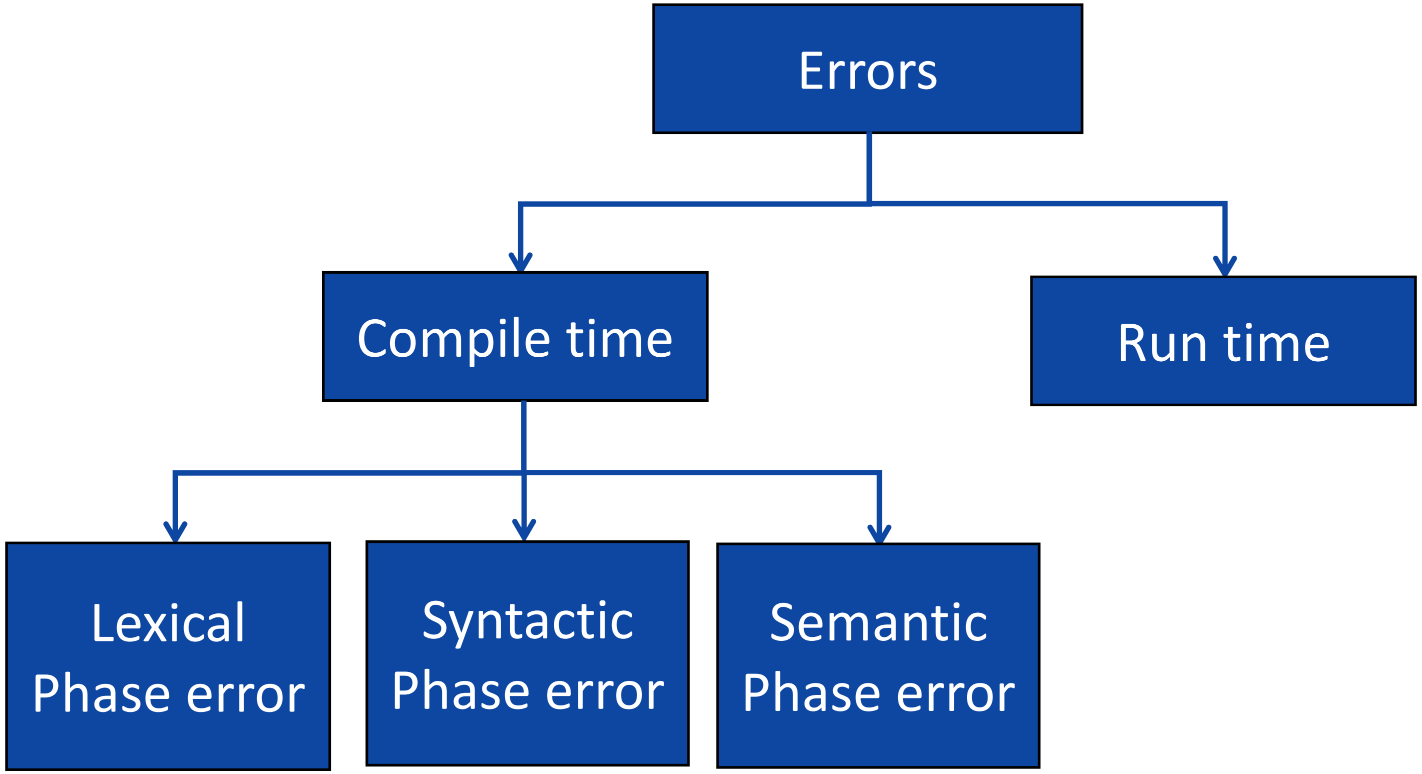
T→ $\boldsymbol{num}\ \ \{\ Print(num.val)\}$



Now, Perform Depth first traversal Postfix=95-2+

Types of errors

Types of Errors



Lexical error

- Lexical errors can be detected during lexical analysis phase.
- Typical lexical phase errors are:
  1. Spelling errors
  2. Exceeding length of identifier or numeric constants
  3. Appearance of illegal characters
- Example:

```
fi ( )
{
}
```

- In above code 'fi' cannot be recognized as a misspelling of keyword *if* rather lexical analyzer will understand that it is an identifier and will return it as valid identifier.
- Thus misspelling causes errors in token formation.

Syntax error

- Syntax error appear during syntax analysis phase of compiler.
- Typical syntax phase errors are:
    1. Errors in structure
    2. Missing operators
    3. Unbalanced parenthesis
- The parser demands for tokens from lexical analyzer and if the tokens do not satisfy the grammatical rules of programming language then the syntactical errors get raised.
- Example:

printf("Hello World !!!") ← **Error: Semicolon missing**

Semantic error

- Semantic error detected during semantic analysis phase.
- Typical semantic phase errors are:
  1. Incompatible types of operands
  2. Undeclared variable
  3. Not matching of actual argument with formal argument
- Example:

  id1=id2+id3*60 (Note: id1, id2, id3 are real)

  (Directly we can not perform multiplication due to incompatible types of variables)

Error recovery strategies (Ad-Hoc & systematic methods)

Error recovery strategies (Ad-Hoc & systematic methods)

- There are mainly four error recovery strategies:
  1. Panic mode
  2. Phrase level recovery
  3. Error production
  4. Global generation

Panic mode

- In this method on discovering error, the parser discards input symbol one at a time. This process is continued until one of a designated set of synchronizing tokens is found.
- Synchronizing tokens are delimiters such as semicolon or end.
- These tokens indicate an end of the statement.
- If there is less number of errors in the same statement then this strategy is best choice.

```
fi ( )
{
}
```

← Scan entire line otherwise scanner will return fi as valid identifier

Phrase level recovery

- In this method, on discovering an error parser performs local correction on remaining input.
- The local correction can be replacing comma by semicolon, deletion of semicolons or inserting missing semicolon.
- This type of local correction is decided by compiler designer.
- This method is used in many error-repairing compilers.

Error production

- If we have good knowledge of common errors that might be encountered, then we can augment the grammar for the corresponding language with error productions that generate the erroneous constructs.
- Then we use the grammar augmented by these error production to construct a parser.
- If error production is used then, during parsing we can generate appropriate error message and parsing can be continued.

Global correction

- Given an incorrect input string x and grammar G, the algorithm will find a parse tree for a related string y, such that number of insertions, deletions and changes of token require to transform x into y is as small as possible.
- Such methods increase time and space requirements at parsing time.
- Global correction is thus simply a theoretical concept.