Date ______
Page ______

Syntax Directed Translation

There are two notations for associating semantic rules with productions, which are:
- Syntax-directed definitions &
- Translation schema

Syntax-directed definition:

A syntax-directed definition is a context-free grammar together with semantic rules. In syntax-directed definition, attributes are associated with grammar symbol and semantic rules are associated with productions.

E.g.

1)

Production	Semantic Rules
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$

2) The syntax directed definition for a simple desk calculator:

new line or E return

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

Date ______
Page ______

Annotated parse tree:

A parse tree constructing for a given input string in which each node showing the values of attributes is called an annotated parse tree.

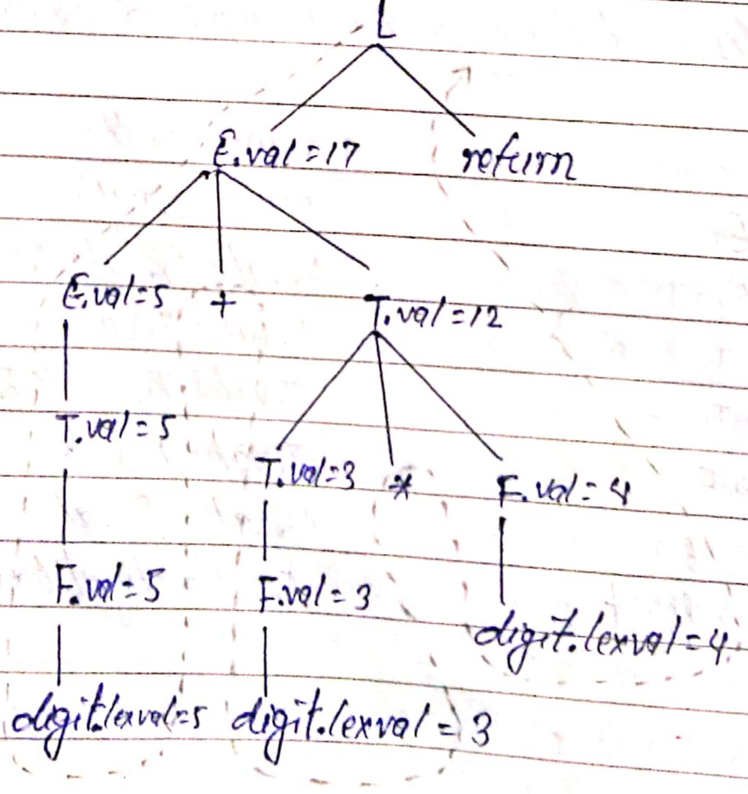
- The processing of computing the attributes values at the nodes is called annotating (or decorating) of the parse tree.

Example:

let's take a grammar;

$L \to E$ return	Print(E.val)
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T * F$	$T.val = T.val * F.val$
$T \to E$	$T.val = E.val$
$F \to (E)$	$F.val = E.val$
$F \to$ digit	$F.val = digit.lexval$

Now, the annotated parse tree for the input string $5+3*4$ is,



DFS → ---(line)
↳ depth 1st search.

Date
Page

Attribute Grammar in Yacc / Bison

$L \to E\ retn$

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid digit$

```
%token DIGIT
%%
L : E '\n'        {printf ("%d\n", $1);}
  ;
E : E'+'T         {$$ = $1+$3;}
  | T             {$$ = $1;}
  ;
T : T'*'F         {$$ = $1*$3;}
  | F             {$$ = $1;}
  ;
F : '('E')'       {$$ = $2;}
  | DIGIT         {$$ = $1;}
  ;
%%
```

<u>Inherited and synthesized Attributes</u> (Types of attributes)

* <u>Synthesized attributes:</u>

The attribute of node that are derived from its children nodes are called synthesized attributes.

E.g.

① $A \to BCD$, A be a parent node & B, C, D are children nodes

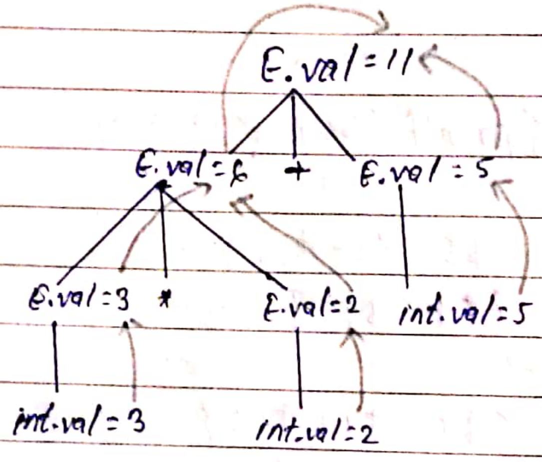
$A.s = B.s$
$A.s = C.s$
$A.s = D.s$
$\}$ Parent node A taking value from its children B, C, D.

$F \to 2 \quad \{F.val = 2\}$

Date
Page

(ii) $E \to E_1 * E_2 \quad \{E.val = E_1.val * E_2.val\}$
$E \to E_1 + E_2 \quad \{E.val = E_1.val + E_2.val\}$
$E \to int \quad \{E.val = int.val\}$

→ value owns from child to parent in the parse tree.
* For. E.g. for string $3*2+5$



* Inherited attributes:
The attributes of node that are derived from its parent or sibling nodes are called inherited attributes.
E.g.
(1) $A \to BCD$
$C.i = A.i$
$C.i = B.i$

(2)

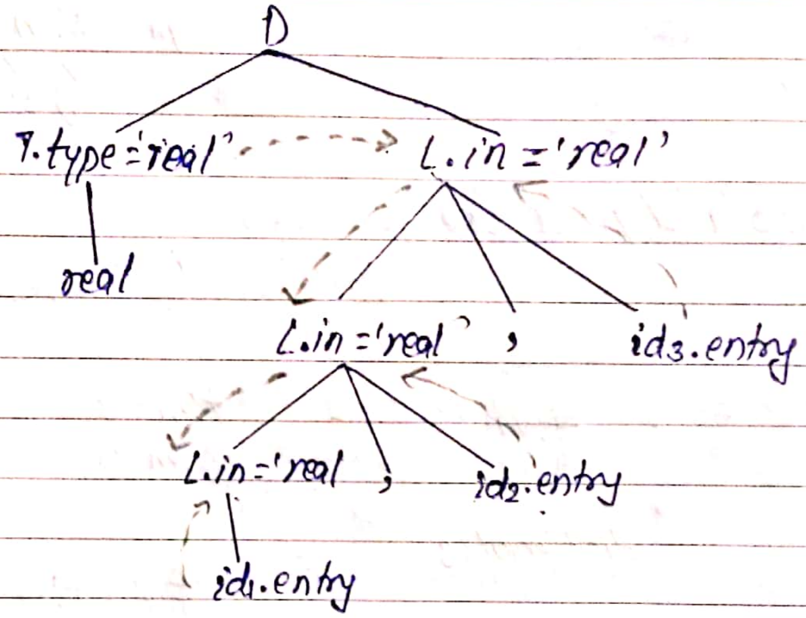
productions	semantic rules
$D \to TL$	$L.in = T.type$
$T \to int$	$T.type = integer$
$T \to real$	$T.type = real$
$L \to L_1, id$	$L_1.in = L.in$, addtype (id.entry, L.in)
$L \to id$	addtype (id.entry, L.in)

Symbol table entry for type info of id.

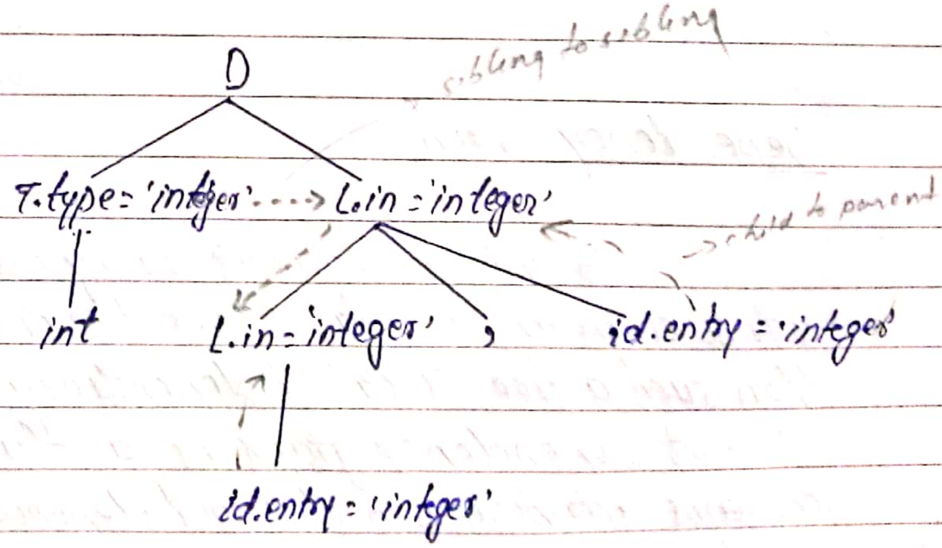
Symbol T is associated with a synthesized attribute type,
Symbol L is associated with an inherited attribute in.

→ Values flows into a node in the parse tree from parents and/or siblings.

Input: real id1, id2, id3



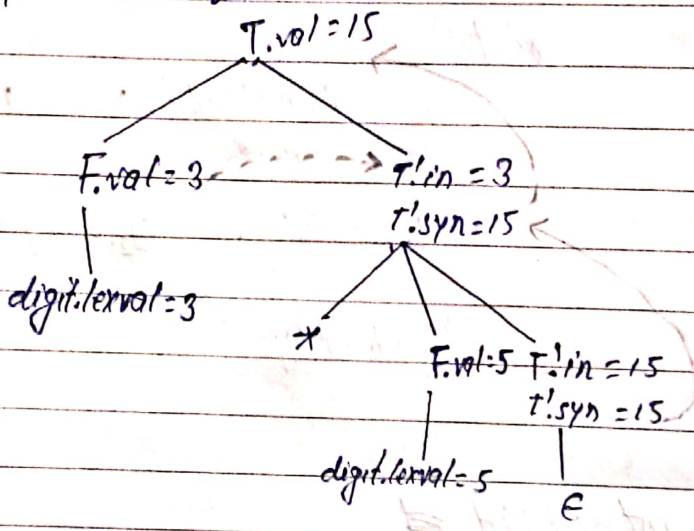
Input: int id, id



in → inherited att.
syn → synthesized att.

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

Annotated parse tree for 3*5



Dependency Graph

If interdependencies among the inherited and synthesized attributes in an annotated parse tree are specified by arrows then such a tree is called dependency graph.

· A dependency graph is a directed graph that contains attribute as nodes and dependencies across attributes as edge.

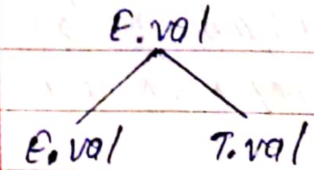
→ For synthesized attributes, each node has dependency to child nodes.

→ For Inherited attributes, any node have dependency to its sibling or parent nodes.

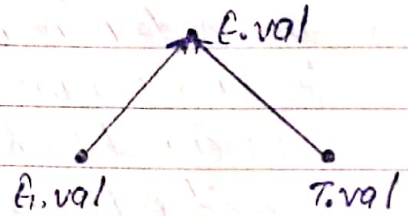
Date ____
Page ____

For e.g.

(1) $E \to E_1 + T$ $\qquad E.val = E_1.val + T.val$



dependency graph → $E.val = f(E_1.val, T.val)$



(2)

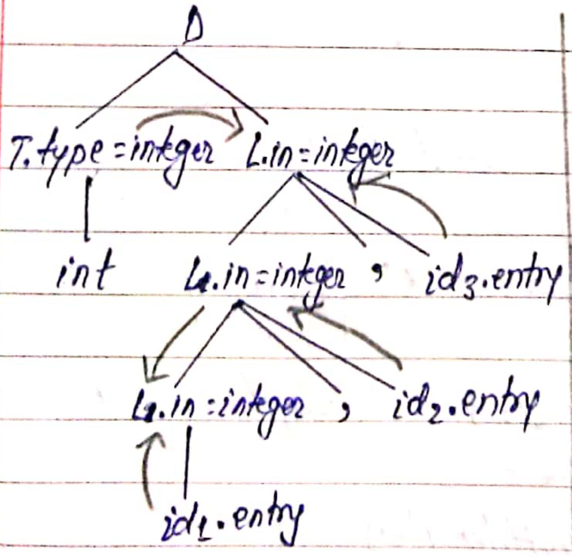
$D \to TL$	$L.in = T.type$
$T \to int$	$T.type = integer$
$T \to real$	$T.type = real$
$L \to L_1, id$	$L_1.in = L.in$, addtype(id.entry, L.in)
$L \to id$	addtype(id.entry, L.in)

Here type is synthesized and in is inherited attribute.

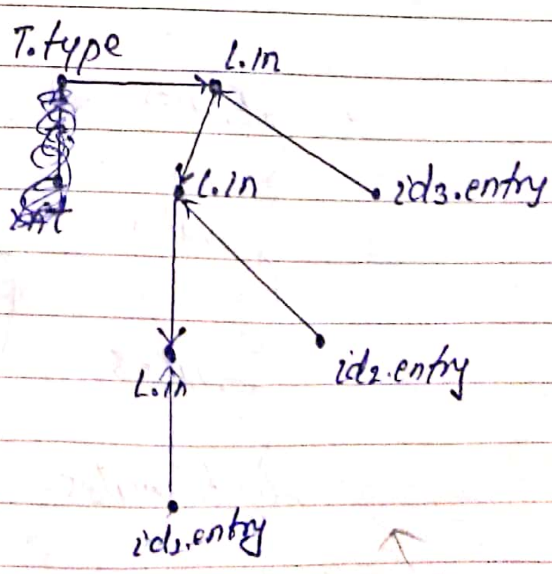
For int id_1, id_2, id_3

The annotated parse tree :

Tree बनाउँदा tree मा L1 पहिले L राख्ने



Dependency graph :



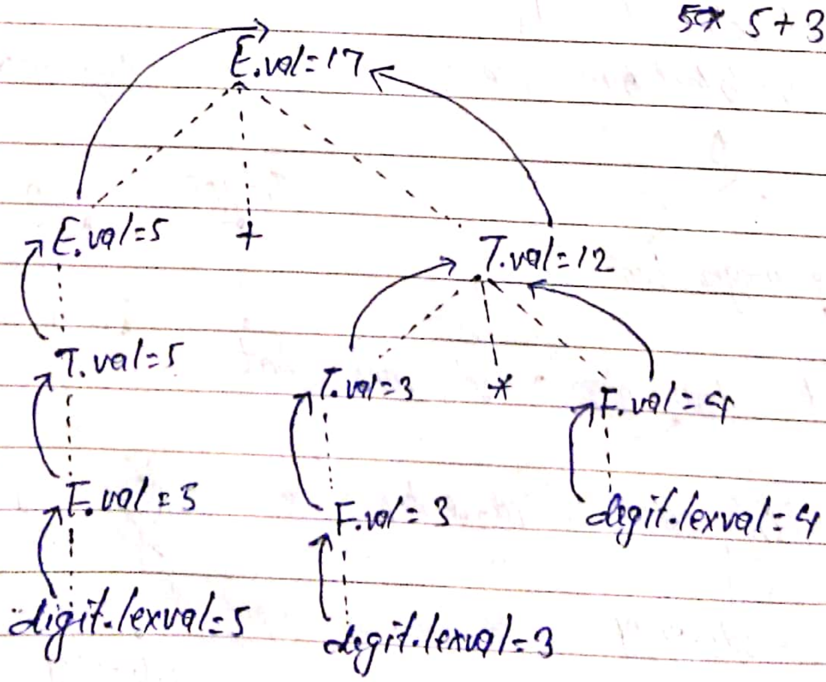
xx wrong मार्क गरे जस्तो लाग्छ ।

Algorithm for dependency graph:

for each node n in the parse tree do
  for each attribute a of the grammar symbol n do
    construct a node in the dependency graph for a.
for each node n in the parse tree do
  for each semantic rule of the form $b = f(c_1, c_2, \ldots c_k)$
  associated with the production used at n do
    for $i = 1$ to k do
      construct an edge from c_i to b.

Prod.	Semantic Rules
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow digit$	$F.val = digit.lexval$

~~5*~~ $5+3*4$



Here dotted line shows the parse tree and directed solid line shows the dependency graph.

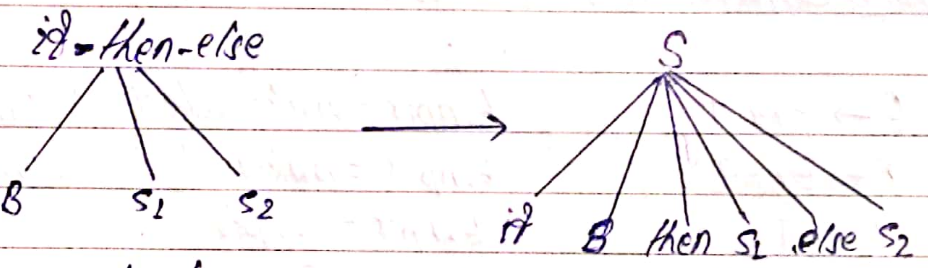
Date ______
Page ______

Construction of syntax tree

Syntax tree is an abstract representation of the language constructs. In the syntax tree, interior nodes are operators and leaves are operands. The syntax tree is used for syntax directed translation with the help of syntax directed definition:

e.g.

(i) $S \rightarrow$ if B then s_1 else s_2 is a production.



syntax tree — parse tree

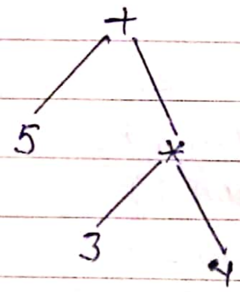
(ii) Arithmetic

$5+3*4$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid digit$



syntax tree

To construct syntax tree we need following functions:

(i) makenode (op, left, right) : It creates an operator node with ~~operator~~ label op. left & right points the address of left and right child.

(ii) makeleaf (id, entry) : It creates an identifier node with label id. Entry is a pointer to symbol table entry for that id.

(iii) makeleaf (num, val) : It creates a node with label num and val is the value of number.

Eg.

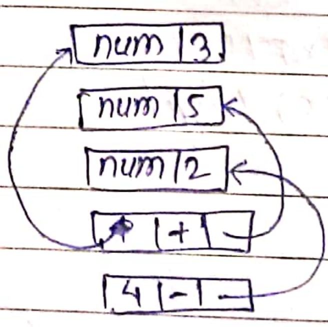
production	Semantic Rules
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to E_1 - T$	$E.val = E_1.val - T.val$
$E \to T$	$E.val = T.val$
$T \to (E)$	$T.val = E.val$
$T \to id$	$T.val = id.entry$
$T \to num$	$T.val = num.val$

Syntax directed def^n for construction of syntax tree

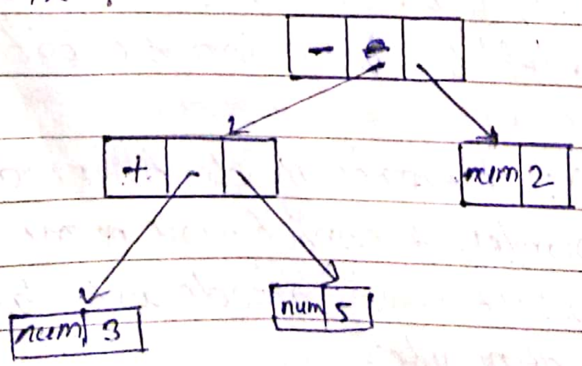
$E \to E_1 + T$	E.nptr = makenode ('+', E_1.nptr, T.nptr)
$E \to E_1 - T$	E.nptr = makenode ('-', E_1.nptr, T.nptr)
$E \to T$	E.nptr = T.nptr
$T \to (E)$	T.nptr = E.nptr
$T \to id$	T.nptr = makeleaf (id, id.entry)
$T \to num$	T.nptr = makeleaf (num, num.val)

expression : $3+5-2$

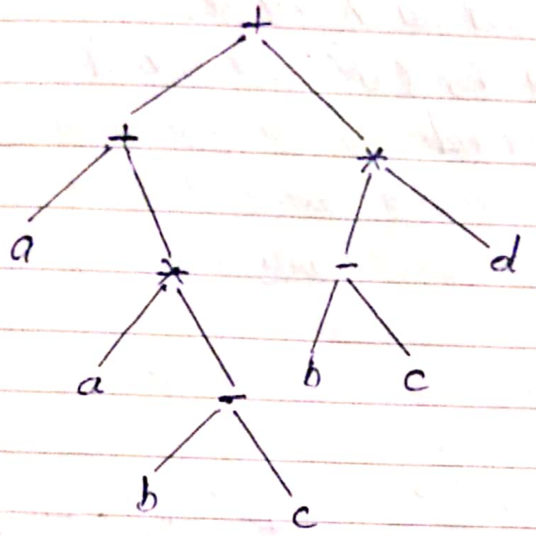
1. P_1 = makeleaf (num, 3) ;
2. P_2 = makeleaf (num, 5);
3. P_3 = makeleaf (num, 2);
4. P_4 = makenode ('+', P_1, P_2);
5. P_5 = makenode ('-', P_4, P_3);



syntax tree :



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



code:

~~1. makeleaf (a~~

1. P_1 = makeleaf (a, a.entry);
2. P_2 = makeleaf (a, a.entry);
3. P_3 = makeleaf (b, b.entry);
4. P_4 = makeleaf (c, c.entry);
5. P_5 = makeleaf (b, b.entry);
6. P_6 = makeleaf (c, c.entry);
7. P_7 = makeleaf (d, d.entry);
8. P_8 = makenode('-', P_3, P_4);
9. P_9 = makenode('*', P_2, P_8);
10. P_{10} = makenode('+', P_1, P_9);
11. P_{11} = makenode('-', P_5, P_6);
12. P_{11} = makenode('*', P_{11}, P_7);
13. P_{12} = makenode('+', P_{10}, P_{12});

Directed Acyclic Graph (DAG):

A DAG for an expression identifies common sub-expressions in the expression.

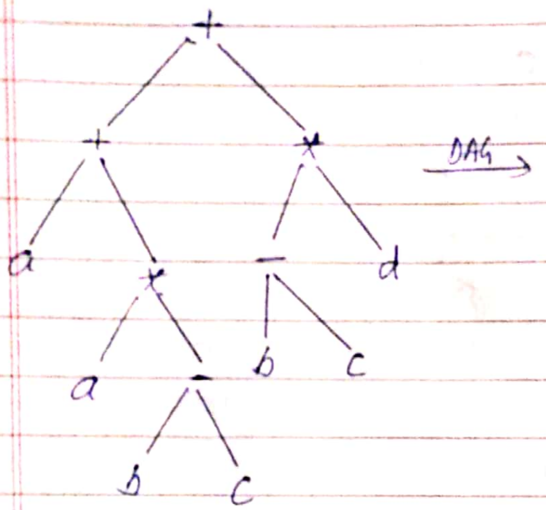
Date ____
Page ____

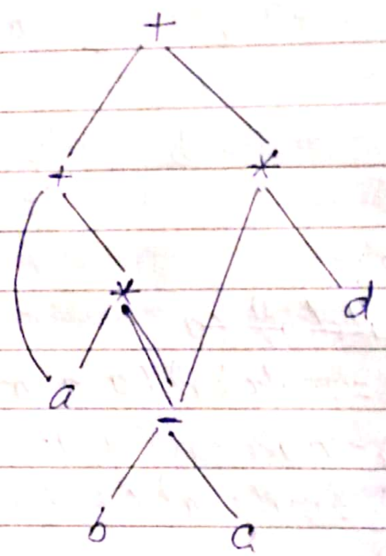
→ A directed acyclic graph (DAG) is an abstract syntax tree with unique node for each value.
interior node → operator, child → operands

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

Syntax tree:

DAG





sequence of instruction for DAG.

1. P_1 = makeleaf (a, a.entry);
2. P_2 = makeleaf (b, b.entry);
3. P_3 = makeleaf (c, c.entry);
4. P_4 = makeleaf (d, d.entry);
5. P_5 = makenod ('-', P_2, P_3);
6. P_6 = makenode ('*', P_1, P_5);
7. P_7 = makenode ('+', P_1, P_6);
8. P_8 = makenode ('*', P_5, P_4);
9. P_9 = makenod ('+', P_7, P_8);

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

Date
Page

S-attributed definition / grammar

A syntax directed definition that uses synthesized attributes exclusively is called an s-attributed definition. A parse tree of an s-attributed definition is annotated by evaluating the semantic rules for the attribute at each node in bottom-up manner.

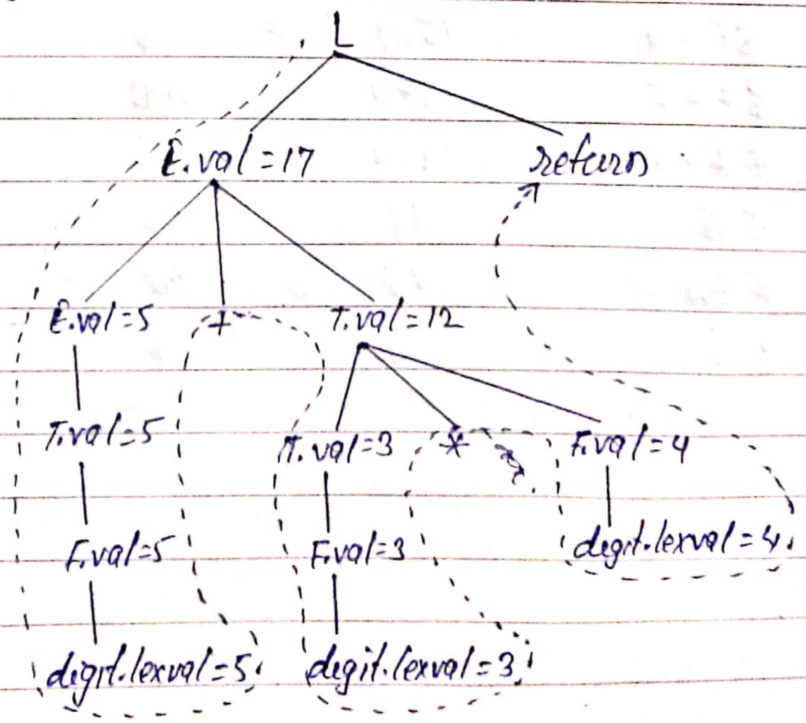
The evaluation of s-attributed definition is based on the depth first traversal of the annotated tree.

E.g.

production	semantic Rules
$L \rightarrow E$ return	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

@

Input string: 5+3*4 return



Date
Page

Bottom-up evaluation of s-attributed defn:

- For bottom-up evaluation of s-attribute definition we use LR parser.
- For bottom-up evaluation of s-attribute definition we put the value of synthesized attribute of the grammar symbol in the stack.

e.g.

For the string 3*5+4n (return) using above grammar

Stack	val	Input	Action
$	—	3*5+4n$	shift
$3	3	*5+4n$	Reduce F→digit
$F	3	*5+4n$	Reduce T→F
$T	3	*5+4n$	shift
$T*	3	5+4n$	shift
$T*5	3*5	+4n$	Reduce F→digit
$T*F	3*5	+4n$	Reduce T→T*F
$T	15	+4n$	~~shift~~ Red. E→T
$E	15	+4n$	shift
$E+	15	4n$	shift
$E+4	15+4	n$	Reduce F→digit
$E+F	15+4	n$	Reduce T→F
$E+T	15+4	n$	Reduce E→E+T
$E	19	n$	shift
$En	19	$	reduce L→En
$L	19	$	accept

$\boxed{\text{Left to right}}$ → attributes flow from left to right

L-attributed definitions

A syntax directed definition that uses both synthesized and inherited attribute but each inherited attribute is restricted to inherits from parent or left sibling only is called L-att. def^n.
mathematically,
A syntax-directed definition is L-attributed if each inherited attribute of X_j on the right side of $A \rightarrow X_1 X_2 \dots X_n$ depends only on
- the attributes of the symbols $X_1, X_2, \dots, X_{j-1}$
- the inherited attributes of A.

e.g.

$A \rightarrow XYZ$ $\{X.i = f_1(A.i),\ Y.i = f_2(Y.s),\ Z.i = f_3(Y.i)\}$ $\boxed{Y.i = f_4(Z.i)}$ → Not L-attributed

$A \rightarrow LM$ $\{L.i = f(A.i),\ \overset{\text{inherited}}{M.i = f(L.s)},\ \overset{\text{synthesized}}{A.S = f(M.S)}\}$

Evaluation of L-attributed definitions:

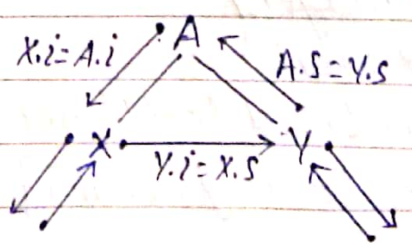
An inherited attribute can be evaluated in a left to right fashion using a depth first evaluation order.

Procedure:

```
dfvisit(n: node);        // depth-first evaluation
for each child m of n, from left to right do
        evaluate inherited attributes of m;
        dfvisit(m);
Evaluate synthesized attributes of n
```

e.g.

$A \rightarrow XY$



$X.i = A.i$
$Y.i = X.S$
$A.S = Y.S$

Translation scheme.

A translation scheme is a context-free grammar in which:
- attributes are associated with grammar symbols;
- semantic actions are inserted within the right sides of productions and are enclosed between braces { }.

E.g.

production	semantic Rules
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$

The translation scheme is

$T \rightarrow T_1 * F \ \{T.val = T_1.val * F.val\}$

If both synthesized and inherited attributes are involved:

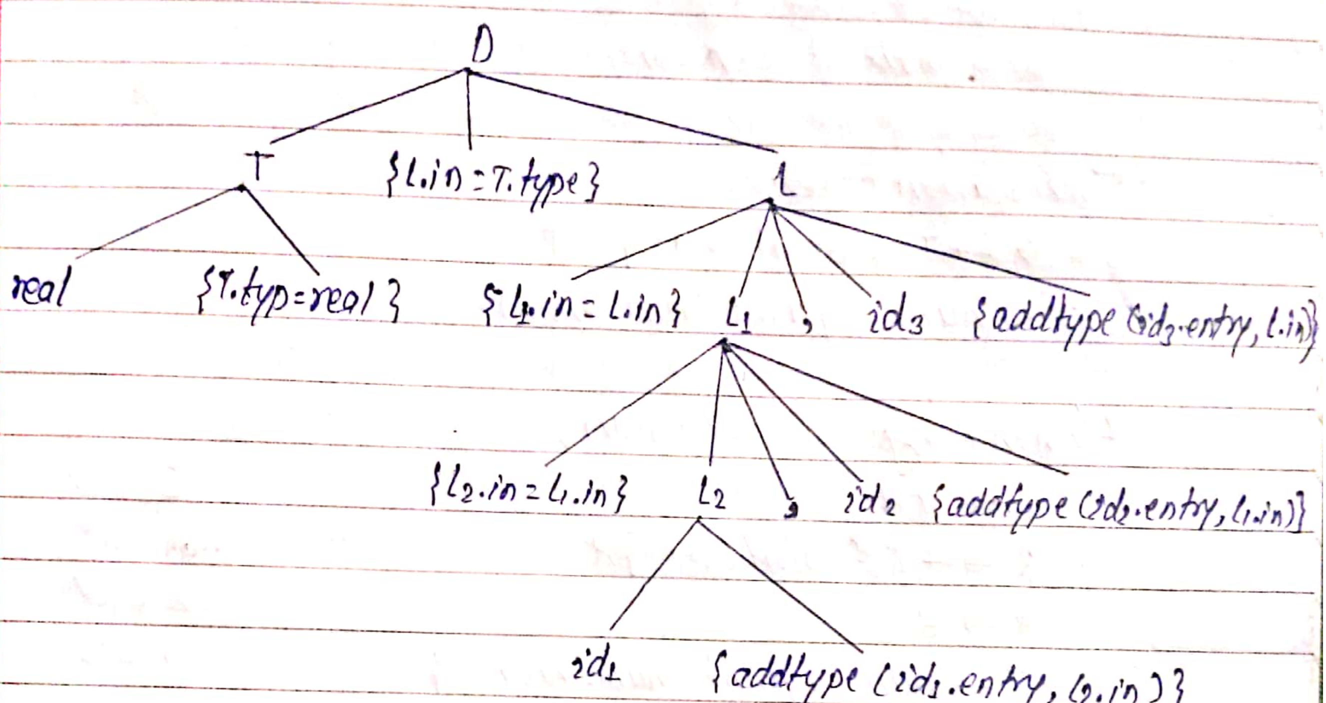
1. For synthesized attribute, for any production like
$A \rightarrow X_1 X_2 X_3 \dots X_n$
the translation actions are written at the right most part of the production.
i.e. $A \rightarrow X_1 X_2 X_3 \dots X_n \ \{ \dots \}$

2. ~~An~~ inherited attribute, for a symbol on the RHS of a production must be computed in an action before that symbol.

E.g.
Translation scheme for the L-attributed defn for "type declaration":

$D \rightarrow T \{L.in = T.type\} L$
$T \rightarrow int \{T.type = integer\}$
$T \rightarrow real \ \{T.type = real\}$
$L \rightarrow \{L_1.in = L.in\} L_1, id \ \{addtype(id.entry, L.in)\}$
$L \rightarrow id \ \{addtype(id.entry, L.in)\}$

Parse tree for input : real id1, id2, id3



Translation scheme that converts infix to postfix :

2+3*4

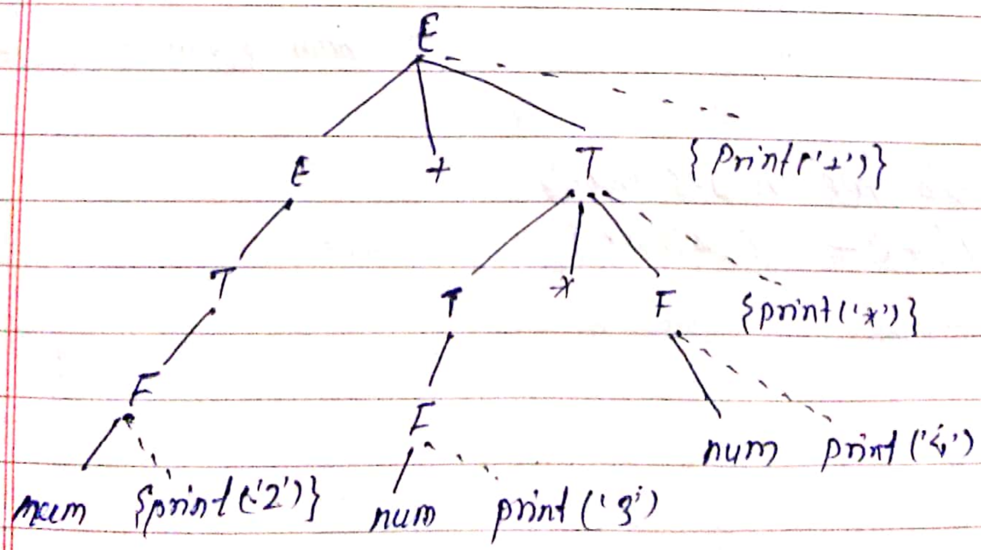
$E \rightarrow E + T$ {print('+');}

$E \rightarrow T$

$T \rightarrow T * F$ {print('*');}

$T \rightarrow F$

$F \rightarrow num$ {print (num.lexval);}



DFS

1st action → print('2')

2nd action → print('3')

3rd action → print('4')

4th action → print('*')

5th action → print('+')

234*+ (Postfix)

$\boxed{\text{If only '+' contains}}$

E.g $a+b+c \Rightarrow ab+c+$

infix postfix

Translation scheme:

$E \rightarrow E+T$ {print('+');} | T

$T \rightarrow num$ {print(num.lexval);}

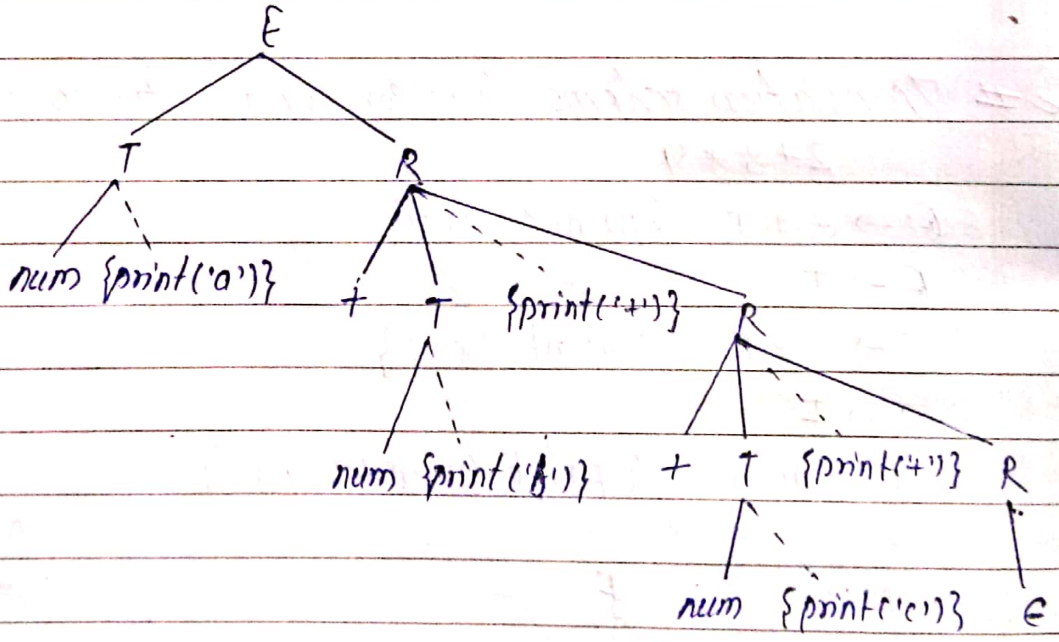
Eliminating left recursion.

$E \rightarrow TR$ ($E' = R'$)

$R \rightarrow +T$ {print('+')} R

$R \rightarrow \epsilon$

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



Traverse the tree in DFS order

$ab+c+$ (postfix)

Translation scheme Requirements:

If a translation scheme has to contain both synthesized and inherited attributes, we have to observe the following rules:

1. An inherited attribute of a symbol on the right side of a production must be computed in a semantic action before that symbol.
2. A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.
3. For synthesized attribute of non-terminal on LHS, the translation ~~sche~~ actions are written at the right most part of the production.

Top-Down Translation

- L-attributed definitions can be evaluated in a top-down fashion (predictive parsing) with translation scheme.
- The algorithm for elimination of left recursion is extended to evaluate action & attribute.

eliminating left Recursion from a Translation scheme

Consider a left recursive translation schema:

$A \rightarrow A_1 Y \quad \{ A.a = g(A_1.a, Y.y) \}$

$A \rightarrow X \quad \{ A.a = f(X.x) \}$

In this grammar each grammar symbol has synthesized attribute written using their corresponding lower case letter.

Removing left recursion

$A \rightarrow XR$ $\qquad\qquad$ $\boxed{A \rightarrow AY / X}$

$R \rightarrow YR \mid \epsilon$ $\qquad\qquad$ $A' = R$

Now,

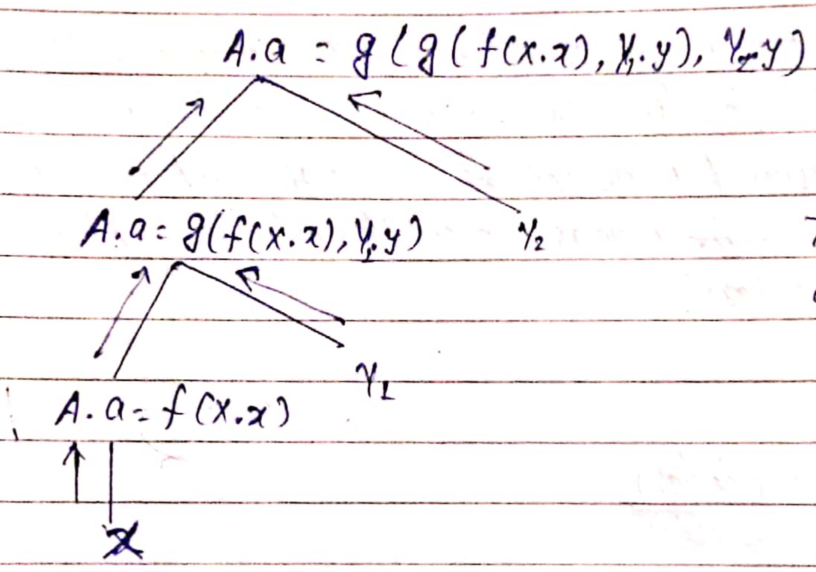
Taking semantic action for each symbol as follows.

$A \to x \{R.i = f(X.x)\}\ R\ \{A.a = R.s\}$

$R \to Y \{R_1.i = g(R.i, Y.y)\}\ R_1 \{R.s = R_1.s\}$

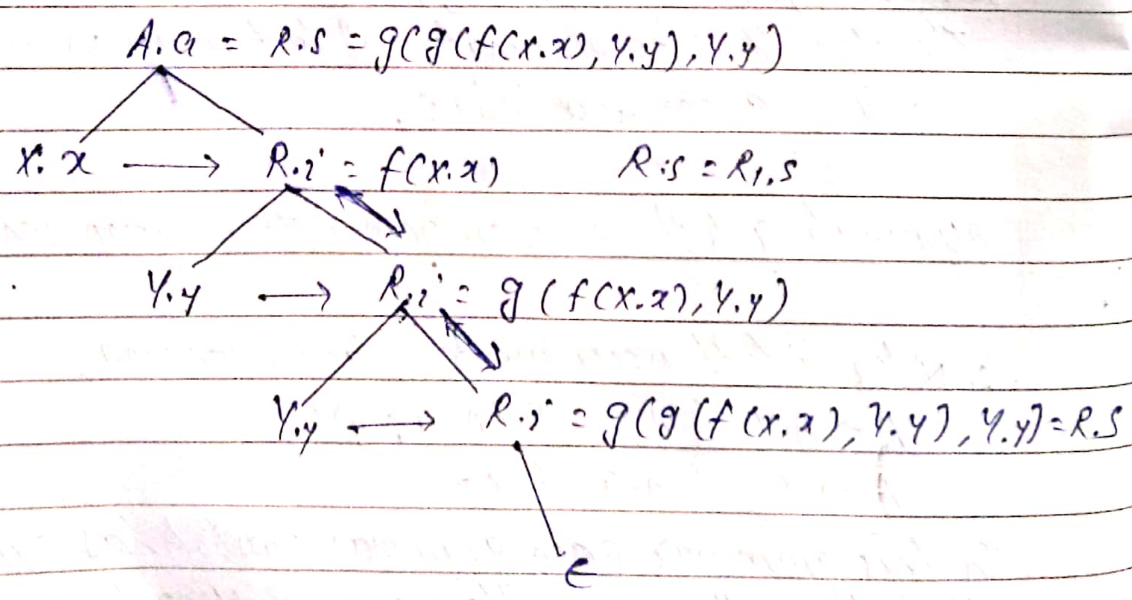
$R \to \epsilon \{R.s = R.i\}$

Evaluation of string xYY



These values are computed according to a left recursive grammar.

or



Date ______
Page ______

Q. Given grammar

$E \to E+E \,/\, E*E \,/\, (E) \,/\, id$

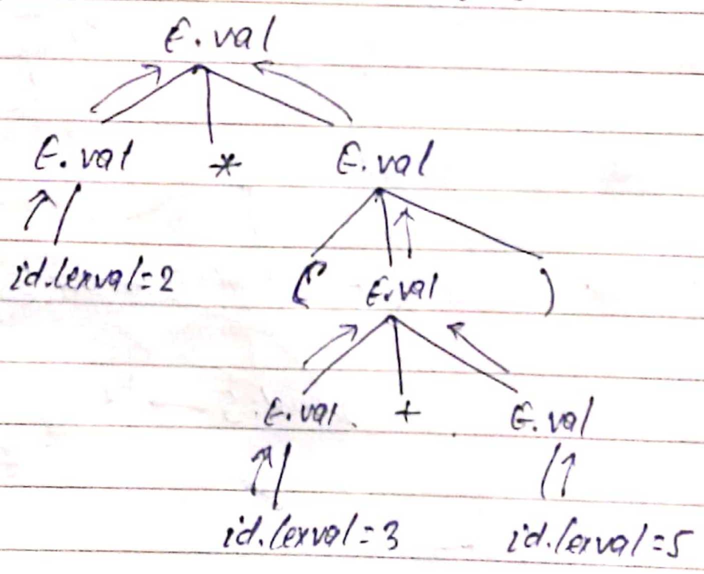
- Annotate the grammar with syntax directed defⁿ using synthesized attributes.
- Remove left recursion from grammar & rewrite the attributes correspondingly.

Solⁿ

First part:

Prod.	S.R.
$E \to E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E \to E_1 * E_2$	$E.val = E_1.val * E_2.val$
$E \to (E_1)$	$E.val = E_1.val$
$E \to id$	$E.val = id.lexval$

Annoted parse tree for $2*(3+5)$ is



Second part:

Removing left recursion as

$E \to (E)R \,/\, id\,R$

$R \to +ER_1 \,/\, *ER_1 \,/\, \epsilon$

Now add the attributes within this non-left recursive grammar as

S → synthesized a

$E \rightarrow (E) \{R.in = E_1.val\}\ R\ \{E.val = R.s\}$

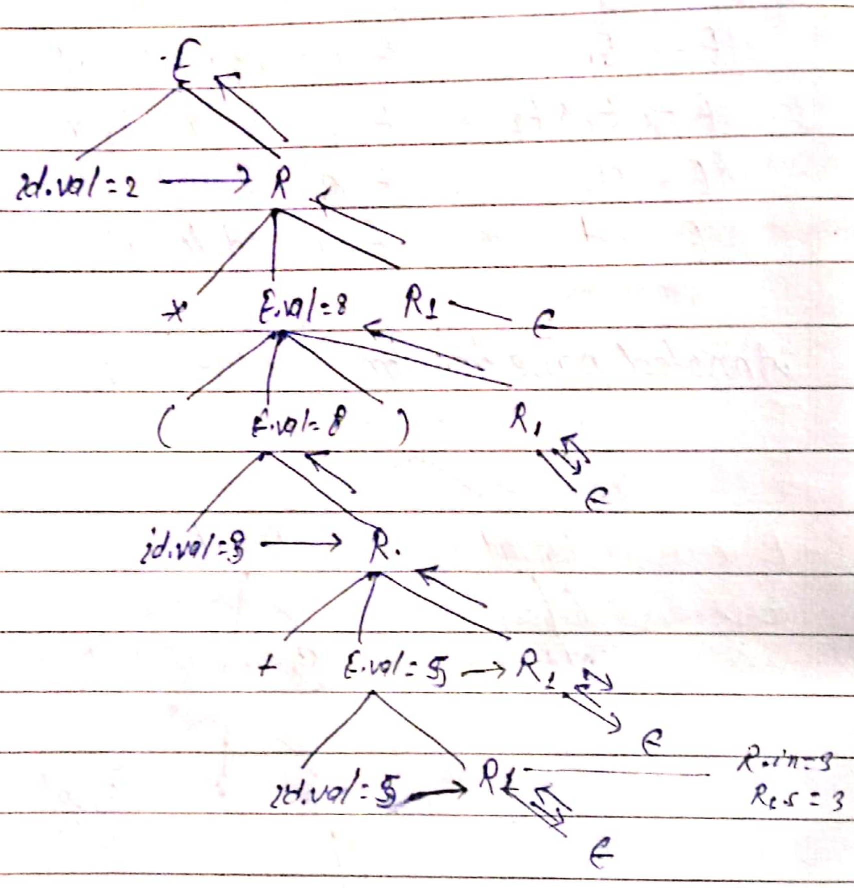
$E \rightarrow id\ \{R.in = id.lexval\}\ R\ \{E.val = R.s\}$

$R \rightarrow +E\ \{R_1.in = E.val + R.in\}\ R_1\ \{R.s = R_1.s\}$

$R \rightarrow *E\ \{R_1.in = E.val * R.in\}\ R_1\ \{R.s = R_1.s\}$

$R \rightarrow \epsilon\ \{R.s = R.in\}$

The annoaked parse tree for 2 * (3+5)



Type checking

- Type checking is the process of verifying that each operation executed in a program respects the type system of the language.
- This generally means that all operands in any expression are of appropriate types and number.
- Type checking is carried out in semantic phase.
- Type checking informⁿ added with the semantic rules.

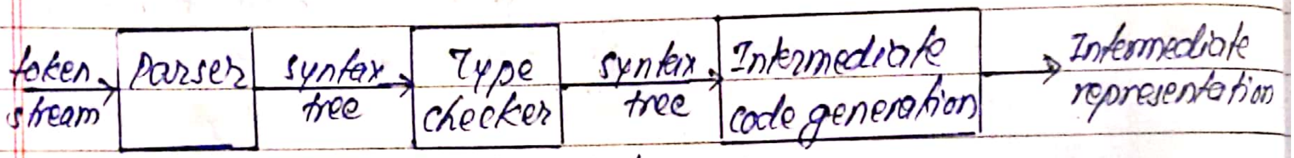


fig: position of type checker

Two types of type checking
- static type checking
- Dynamic type checking

static type checking - checks at compile time

static checking refers to the compile-time checking of programs in order to ensure that the semantic conditions of the language are being followed.
static checks include:

1. Type checks:
   Report an error if an operator is applied to incompatible operand.
   E.g. 2+2.5 = Error

Date
Page

E.g.

```
int op(int), op(float);      //overloading
int f(float);
int a, c[10], d;
d = c+d;      // Error : type mismatch
*d = a;       // Error : not a pointer type
a = op(d)     // OK : overloading
a = f(d)      // ok : Coersion of d to float
```

2. Flow-of-control ~~block~~ checks:

statements that results in a branch need to be terminated correctly. E.g. Break statement in C

```
myfunc (int a)
{
  cout << a;
  break;        // ERROR
  statements;
}
```

misplaced break
- break can't be used anywhere
- it can be only used in loop, switch case.

```
myfunc()
{ ...
  switch(a)
  { case 0:
      ....
      break;  //ok
    case 1:
      ....
      break;  //ok
    defult:
      ...
  }
}
```

```
myfunc(int a)
{
  while(a)
  {
    ...
    if (i>10)
      break;   // ok
  }
}
```

Date
Page

3. Uniqueness check:
In some context, an object must be defined exactly once.
e.g.

```
myfunc()
{
    int i, j, i ;     // ERROR : i is multiple
    ...
}
```

4. Name-related checks:
~~some tim~~ sometimes the same name may be appeared two or more times.

Dynamic checking

- It is done at run-time.
- Implemented by including type information for each data location at run time.
- Compiler generates some verification code to enforce programming ~~dynamic~~ language's dynamic semantics.

If a programming language has no any dynamic checking, it is called strongly typed language. (i.e. there will be no type errors during run-time).

→ Compiler generates code to do the checks at run-time.

Type system

A type system is a collection of rules for assigning type expression to the part of a program. A type checker implements a type system.
e.g.
if both operands of addition are of type integer, then result is of type integer.
- A sound type system eliminates run time type checking for type errors.

Type Expression

- A basic type is type expression. e.g. integer, real, char etc.
- A type name is type expression. e.g. variable name, function name, constant etc.
  if x is name of a variable, then x itself is also type expression.
- A type constructor applied to type expression is also type ~~constructor~~ expression. e.g.
  - array : If T is a type expression and I is a range of integer then array (I, T) is a type expression.
  - record : If $T_1, T_2, \ldots, T_n$ are type expressions and $f_1, f_2, \ldots f_n$ are field names, then record $((f_1, T_1), (f_2, T_2), \ldots (f_n, T_n))$ is a type expression.
  - If T_1 and T_2 are type expressions, then cartesian product $T_1 \times T_2$ is ~~type~~ expression product.
  - pointer : If T is a type expression, then pointer(T) is a type expression. e.g. pointer (int)

- functions: mapping a domain type D to a range type R.
  E.g. int → int represents the type of function which takes an int value as parameter and return type is also int.

Specification of a simple type checker

The specification of a type checker is defined by the translation scheme based on the syntax directed definitions associated with the type informations to each symbols.

E.g.

$P \rightarrow D ; E$

$D \rightarrow D ; D$

$D \rightarrow id : T$ $\{ addtype(id.entry, T.\overset{type}{\cancel{val}}) \}$

$T \rightarrow char$ $\{ T.type = char \}$

$T \rightarrow int$ $\{ T.type = int \}$

$T \rightarrow real$ $\{ T.type = real \}$

$T \rightarrow \uparrow T_1$ $T \rightarrow * T_1$ $\{ T.type = pointer(T_1.\overset{type}{\cancel{val}}) \}$

$T \rightarrow array\ [intnum]\ of\ T_1$ $\{ T.\overset{type}{\cancel{val}} = array(1..intnum.val, T_1.val) \}$

e.g.

a : int ; , b : char ;

Q. Consider the following grammar for arithmetic expression using an operator 'op' to integer or real number.

$E \rightarrow E_1\ op\ E_2 \mid num.num \mid num \mid id$

Give syntax directed defⁿ as translation scheme to determine the type expression when two integers are used in expⁿ, resulting type is integer otherwise real.

Solⁿ

Date
Page

specification of type checker for this problem:

$E \rightarrow id$ { E.type = lookup (id.entry) }
$E \rightarrow num$ { E.type = integer }
$E \rightarrow num.num$ { E.type = real }
$E \rightarrow E_1 \; op \; E_2$ { E.type = if (E_1.type = integer and E_2.type = integer)
then integer
else if (E_1.type = integer and E_2.type = real)
then real
else if (E_1.type = real and E_2.type = integer)
then real
else if (E_1.type = real and E_2.type = real)
then real
else typeerror()
}

Type checking for boolean expⁿ

$E \rightarrow true$ { E.type = boolean }
$E \rightarrow false$ { E.type = boolean }
$E \rightarrow literal$ { E.type = char }
$E \rightarrow num$ { E.type = integer }
$E \rightarrow id$ { E.type = lookup (id.entry) }
$E \rightarrow E_1 \; and \; E_2$ { E.type = if (E_1.type = boolean and E_2.type = boolean)
then boolean else typeerror(); }
$E \rightarrow E_1 \; or \; E_2$ { E.type = if (E_1.type = boolean and E_2.type = boolean)
then boolean else type-error(); }
$E \rightarrow E_1 + E_2$ { E.type = if (E_1.type = integer and E_2.type = integer) then
integer
else if (E_1.type = char and E_2.type = integer) then
integer

Scanned with CamScanner

else if (E_1.type = integer and E_2.type = char) then
integer
else if (E_1.type = char and E_2.type = char) then
integer
else type-error();
}

Type checking of expression:

\# $E \to$ literal { E.type = char }
$E \to$ num { E.type = integer }
$E \to$ id { E.type = lookup(id.entry) }
$E \to E_1$ mod E_2 { E.type = if (E_1.type = integer and E_2.type = integer) then integer else type_error(); }
$E \to E_1[E_2]$ { E.type = if (E_2.type = integer and E_1.type = array(s, t)) then t else type_error(); }
$E \to *E_1$ $E \to E_1\uparrow$ { E.type = if (E_1.type = pointer(t)) then t else type_error(); }

Type checking of statements

Assignment statement:

$S \to id = E$ { S.type = if (id.type = E.type) then void else type-error(); }

If then else statement:

$S \to$ if E then S_1 { S.type = if (E.type = boolean) then S_1.type else type_error(); }

While statement:

$S \to$ while E do S_1 { S.type = if (E.type = boolean) then S_1.type else type-error(); }

$S \to S_1; S_2$ $\{S.type = if\ (S_1.type = void\ and\ S_2.type = void)\ then\ void\ else\ type\text{-}error();\}$

Type checking of functions

$E \to E_1(E_2)$ $\{E.type = if(E_2.type = s\ and\ E_1.type = s \to t)\ then\ t\ else\ type\text{-}error();\}$

$T \to T_1 \to T_2$ $\{T.type = T_1.type \to T_2.type\}$

- Function whose domains are function from two characters and whose range is a pointer of integer

$T \to int\ \{T.type = int\}$
$T \to char\ \{T.type = char\}$
$T \to pointer[T_1]$ $\{T.type = pointer(T_1.type)\}$
$E \to E_1[E_2]$ $\{E.type = if\ (E_2.type = (char, char)\ and\ E_1.type = (char, char) \to pointer(int))\ then\ E_1.type\ else\ type\text{-}error();\}$

Type conversion and coercion

- Type conversion is explicit.

$c = a + int(b)$

1) Convert b into int
2) sum
3) Assign

Date
Page

- Type conversion which happen implicitly is called coercion. Compiler converts one type to another type automatically.

$$\overset{int}{c} = \overset{int}{a} + \overset{float}{b};$$

1) Conversion of 'a' from int to float
2) sum a+b both in float, result = float
3) convert result into int from float
4) Assign result to c.

Equivalence of Type Expression.

Two expressions are structurally equivalent if they are two expressions of same basic types or are formed by applying same constructor.

structural equivalence algorithm:

```
boolean sequival (s, t)
{
   if s and t are same basic types
        return true;
   else if s = array (s1, s2) and t = array (t1, t2) then
        return sequival (s1, t1) and sequival (s2, t2)
   else if if s = s1 x s2 and t = t1 x t2 then
        return sequival (s1, t1) and sequival (s2, t2)
   else if s = pointer (s1) and t = pointer (t1) then
        return sequival (s1, t1)
   else if s = s1 -> s2 and t = t1 -> t2 then
        return sequival (s1, t1) and sequival (s2, t2)
   else return false
}
```

Date
Page

E.g.
int a, b;
Here a and b are structurally equivalent.

Q. Write the type expressions for the following types:

a. An array of pointers to real's, where the array index range from 1 to 100.

Sol

~~$T \rightarrow real \quad \{T.type = real\}$~~

~~$E \rightarrow E_1[E_2] \quad \{E.type = if (E_2.type = int$~~

~~$E \rightarrow array [100, T] \quad \{if\ T.type = real\ the$~~

$T \rightarrow real \quad \{ T.type = real \}$

$E \rightarrow E_1[E_2] \quad \{$ if (E_2.type = int and E_1.type = array (1, 2, .. 100, E.type)) then E.type = real else E.type = type-error(); }

$E \rightarrow * E_1 \quad \{$ if (E_1.type = pointer(T.type) then E.type = T.type else E.type = type-error(); }