* Syntax directed definition- (SDD)

SDD = Grammer + Semantic Rules

Production	Semantic Rule
$E \rightarrow E+T$	E.val = E.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T * F$	T.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow$ digit	F.val = digit.val

- Syntax directed Translation
  - ↳ method that is utilized in semantic and language Translation from source to target language through semantic Actions and attributes attach to grammer.

Attributes are associated with grammer symbols and Semantic Rules are associated with productions.

Attributes may be number, strings, reference, datatypes etc.

- Annotated Parse Trees
  - ↳ A Parse Tree Containing the values of attributes at each node for given input string is Called annotated / decorated Parse ~~Tree~~ Tree.

- Types of Attributes - (2)

(A) Synthesized Attribute (S-Attribute)

↳ Attributes which derive their values from their children nodes

$E \rightarrow E_1 + T$: E.val = E_1.val + T.val

↳ This derive from E_1.val and T.val

⇒ A SDD that uses only synthesize attributes is Called S-Attributed SDD

$A \rightarrow BCD$

$A.i = B.i$

$A.i = C.i$

$A.i = D.i$

- Semantic Actions are always placed at right end of the production. (Postfix SDD)
- Attributes are evaluated with Bup. (Bottom-up)

(B) Inherited Attributes -

Attributes which derive their values from their parent or sibling nodes

$A \rightarrow BCD$

$C.i = A.i$

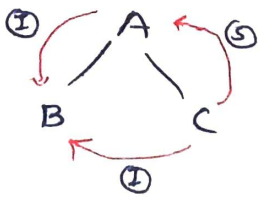
$C.i = D.i$

$C.i = B.i$

A SDD that uses both synthesized and inherited Attributes is called as L-attributed SDD but each inherited attribute is restricted to inherit from parent or left siblings only.

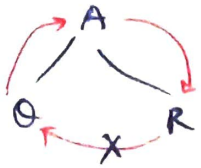
- Semantic Actions are placed Anywhere on RHS of the Production.
- Attributes are evaluated by traveling Parse Tree depth first, left to right.

Exa - 1) $A \rightarrow BC \{B.i = A.i, C.i = B.i, A.i = C.i\}$



$\Rightarrow$ L-Attributed SDD

2) $A \rightarrow \Theta R \{R.i = A.i, \Theta.i = R.i, A.i = \Theta.i\}$



$\Rightarrow$ Not L-Attributed SDD

• Binary to Decimal :

$S \rightarrow L \quad \{ S.dv = L.dv \}$

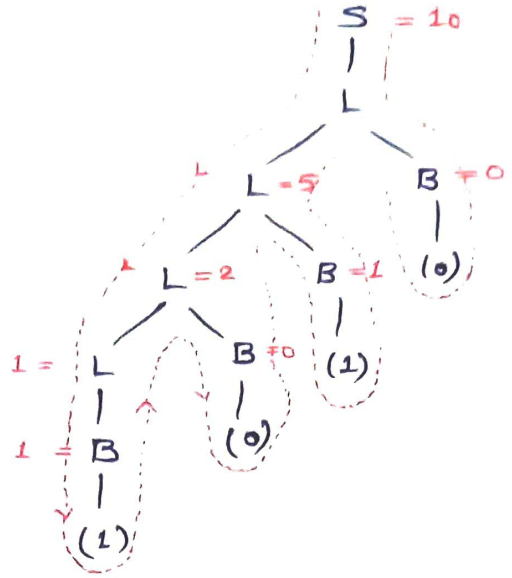
$L \rightarrow LB \quad \{ L.dv = 2 * L.dv + B.dv \}$

$L \rightarrow B \quad \{ L.dv = B.dv \}$

$B \rightarrow 0 \quad \{ B.dv = 0 \}$

$B \rightarrow 1 \quad \{ B.dv = 1 \}$

Let string = 1010



✱ Construct Syntax Tree from SDD -

↳ Unlike Parse Trees, syntax trees are abstract. operators are interior nodes, and operands are leaves.

$E \rightarrow E + T \quad \{ E.nptr = mKnode(E.nptr, +, T.nptr) \}$

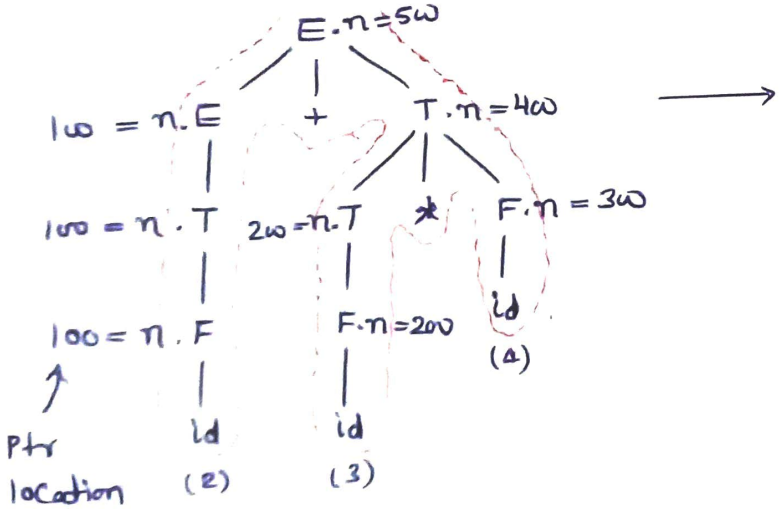
$E \rightarrow T \quad \{ E.nptr = T.nptr \}$

$T \rightarrow T * F \quad \{ T.nptr = mKnode(T.nptr, *, F.nptr) \}$

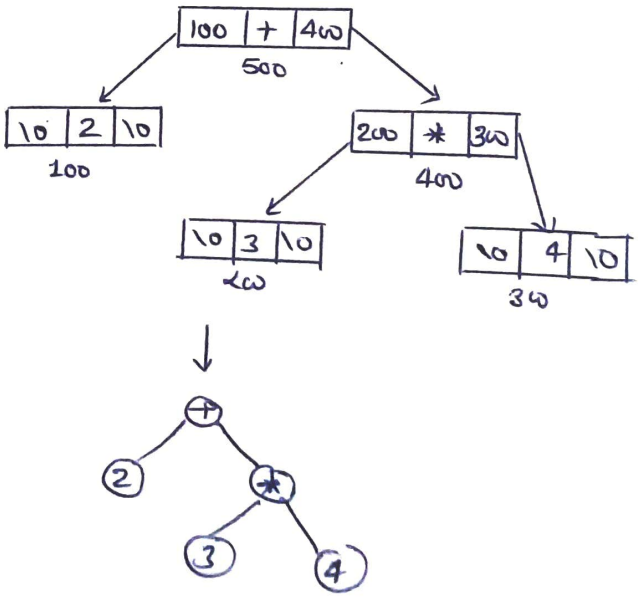
$T \rightarrow F \quad \{ T.nptr = F.nptr \}$

$F \rightarrow id \quad \{ F.nptr = mKnode(NULL, id, NULL) \}$

string = 2 + 3 * 4



Parse Tree



Syntax Tree

Parse Tree

1.) Parse Tree is used as intermediate ~~Code~~ representation during the Compilation process.

2) More detailed and longer

3) It includes extra information like Comments and White spaces

4.) Not in human readable

Syntax Tree

1) Syntax Tree is used as the final representation for generating machine code or intermediate code.

2) Simple and more abstract

3) It only includes necessary information for Code generation

4) Can be read and understood by humans.

* SDD to Store type Information into Symbol table

$D \longrightarrow D_1, id$

↳ To distinguise from D we express it by D_1

$D \longrightarrow D_1, id$ { Addtype (id, D_1.type); D.type = D_1.type }

$D \longrightarrow T\ id$ { Add type (id, T.type); D.type = T.type }

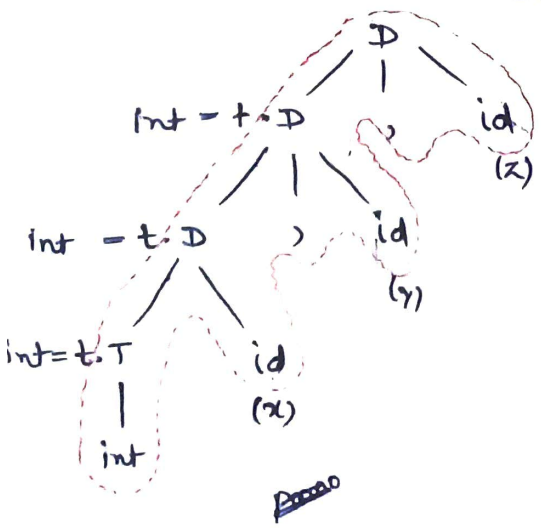
$T \longrightarrow int$ { T.type = int }

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

$T \longrightarrow float$ { T.type = float }

Variable Name	Variable TYPE

Str → ~~X,Y,Z~~
int x,y,z ;



V.N.	V.T.
x	int
y	int
z	int

Note-

- Evaluation methods of Parse Tree

once a parse Tree is built, Compiller must decide in what order to Calculate these attributes -

(A) Dynamic Dependence Based methods-

(i) Build the Parse Trees

(ii) Build the dependence graph (directed)

(iii) Topological Sort

(iv) Evaluate the attributes in this specific topological order.

(B) Rule Based methods-

(i) The Compiler analyzes the grammer rules at Compiler generation time.

(ii) It determines fixed (static ordering) for evaluation.

(iii) determined nodes in pre determined order.

(c) Oblivious method-

↳ This methods ignores the specific structure of the rules or the tree during evaluation.

- It uses a Convenient order (chosen at design time), such as multiple passes (e.g. Left to Right)

* Type checking - (2) → 1) Static checking : done at Compile time
Catch error before program runs

2) Dynamic checking : happens at runtime

- flow of Control checks- This ensures that "path" the program takes is valid and Context sensitive

- visibility and Scope
- order of operation : Variable must be declared before it is used.
- return type : should be valid return type

- Uniqueness Check -
  ↳ Identifiers should be used once within their Context.
  - single definition
  - ✓ distinct labels in switch-Case

Parser ⟶ Type checker ⟶ Intermediate Code Generation

* Type systems - Collection of rules that assigns type to different Parts of a program (like Variables, functions)

types → integer, boolean, Character,

- Languages Can be divided into 3 Categories based on how they handled types-

A) Untyped - The language has no Concept of types.
Data is just bits and programmer handles this
Exa - Assembly.

B) Statically typed - Types are checked during Compile time.
Errors are Caught before the program runs
Exa - C++, Java, Pascal.

C) Dynamically Typed - These are checked during runtime.
You Can change the type of Variable while the program is executing.
Exa - Javascript, Python

- Static semantic checking - (during Compilation)

  1) Type Checks : Report if an operator applied to incompatible operands. E.g. ⇒ (int + String)

  2) flow of Control checks : Ensuring statements that cause Control to leave a block (like break, return) are used in the right place.

  3) Uniqueness check

  4) Name related check : identifiers are ~~declared~~ declared before they are used.

- Dynamic checks - (~~during~~ While the program running)

  1) Array Bound check

  2) Null Pointer dereference.

* Type Expressions -

  → A Type Expression represents the type of language Construct

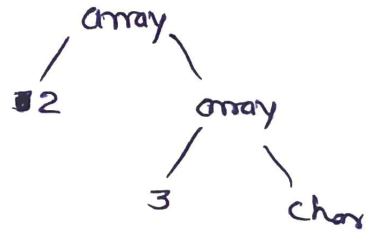
  → We Can build Complex types using Type Constructor
  ↳ Array, Pointer, Products, functions, Struct/Record

- Arrays - store multiple elements of the same type :

  Type Expr = array (size, type)

  int arr[10] : array (10, integer)

  Char matrix[2][3] : array (2, array (3, Char))



- **Product types** – (Cartesian Product)

  $T_1 \times T_2$ (This Combines multiple Values togeather)

```
struct Point {
    int x;
    int y;
}
```

→ (TE = Integer x Integer)

- **Pointers** – Stores memory Address of another type

  Type Expr = Pointer (type)

  char *name : Pointer (char)

  int **x ; : Pointer (Pointer (integer))

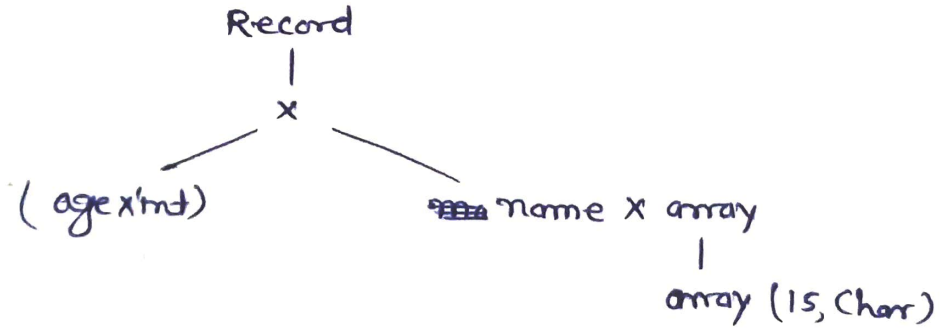
Pointer
|
Pointer
|
Integer

- **Records / Struct** –

```
struct Student {
    int age;
    char name[15];
}
```

⟹

Record (
(age x integer)
x
(name x array (15, Char))
)



- **Functions** – maps : Input → output

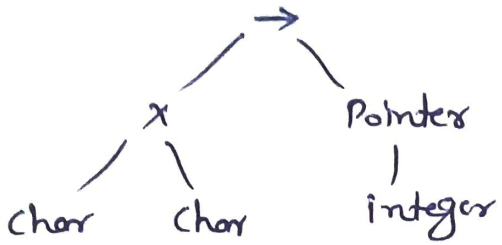
  D → R

int square (int x) : integer → integer

int add (int a, int b) : Int x Int → int

int * foo (char a, char b) ;

⇒ char x char → Pointer (int)



→ With the help of Grammer Rules and Semantic Actions Compiler builds Symbol table.

* Type Conversion-

$E = E_1 + E_2$

5 + 10 (integer + integer) ✓ valid

2.5 + 1.2 (real + real) ✓

5 + 2.3 (integer + real)

Compiler Cannot directly add different types so it performs type Conversion. (Coercion)

- Compiler automatically Converts one type to another.

Smaller → larger precision

5 + 2.3 (5 → 5.0) This is implicit / automatic Coercion.

5.0 + 2.3 (real + real)

if programmer manually Converts → Explicit Conversion

float x = (float) 10 ;

E_1	E_2	result	Action
int	int	int	None
~~float~~ real	real	real	None
int	real	real	int → real
real	int	real	int → real

Note - Array Access type Checking -

Expr - $E_1[E_2]$

1) index must be integer

arr[3] ✓ arr[2.5) ✗

2) Array Element type

$E_1.type = array(n, T)$

$E_1[E_2].type = T$

- Name Equivalance - Two Variables are Considered to have the Same type only if they were declared using the exact Same type name.

```
Struct Student1 {
    int roll;
    Char name[30];
}
```

```
Struct Student2 {
    int roll;
    Char name[30];
}
```

x, y = student1() z = student2()

x, y are Name Equivalence

- Structural Equivalence - internal structure is identical, regardless of their names

x, z → Structural Equivalence

* SDD to generate Three Address Code -

$S \rightarrow id = E$ { gen (id.name = E.place); }

$E \rightarrow E_1 + T$ { E.place = new temp();
gen (E.place = E_1.place + T.place); }

$E \rightarrow T$ { E.place = T.place }

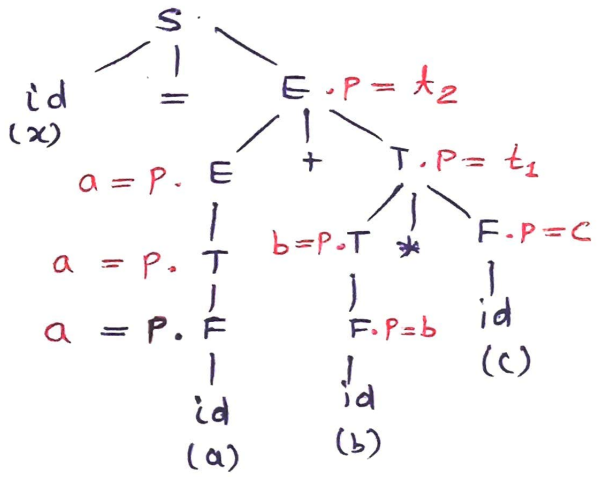
$T \rightarrow T_1 * F$ { T.place = new temp();
gen (T.place = T_1.place * F.place); }

$T \rightarrow F$ { T.place = F.place }

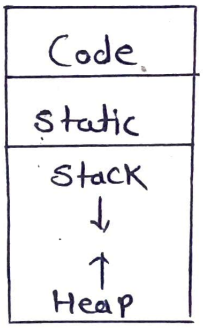
$F \rightarrow id$ { F.place = id.name }

Let x = a + b * c

$$\begin{aligned} t_1 &= b * c \\ t_2 &= a + t_1 \\ x &= t_2 \end{aligned}$$



* Run time Allocation -



- Static Allocation -

1) Memory Allocation is done at Compile time
2) Binding do not change at run time
3) Recursion is not supported
4) Size of data objects must be Known at Compile time
5) Dynamic data structure not supported

- Stack Allocation -

1) Recursion supported
2) Local Variable belongs to new activation record
3) Dynamic Data structure not supported

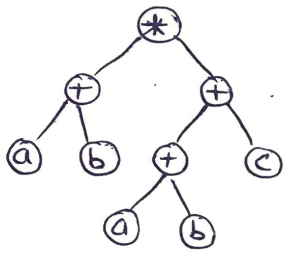
- Heap Allocation-

1) Allocation and Deallocation will be done at anytime based on User requirement
2) Recursion supported
3) Dynamic data structure supported.

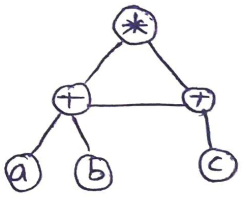
* Intermediate Code generation-

Expr = (a+b) * (a+b+c)

- Linear form
  - Postfix Notation: ab+ab+c+*
  - Three Address Code:
    $t_1 = a+b$
    $t_2 = a+b$
    $t_3 = t_2+c$
    $t_4 = t_1 * t_3$
- Tree form
  - Syntax Tree



  - DAG



- Types of Three Address Code-

1) x = y op z
2) x = op z
3) x = y
4) if (x rel_op y) goto L
5) goto L
6) A[i] = x
   y = A[i]
7) x = * p
   y = &x