Syntax Directed Translation

- Grammar + Semantic Rules = Syntax Directed Translation (SDT)
- Attach just about anything to a prod. rule. This will be executed when it is REDUCED.
- Traverse the parse tree from top to bottom, left to right & upon REDUCE execute attached action(s)

Eg. ① $E \to E + T \mid$ { E.val = E.val + T.val }

T { E.val = T.val }

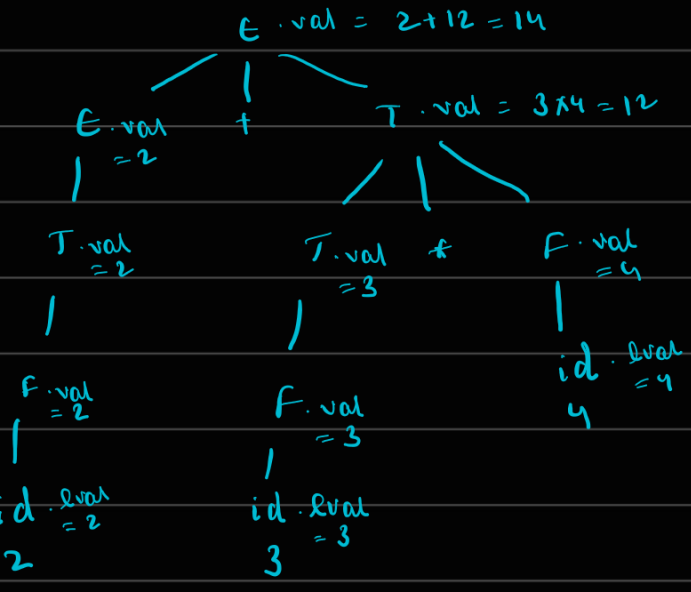
$T \to T * F \mid$ { T.val = T.val * F.val }

F { T.val = F.val }

$F \to id$ { F.val = id.lval }

Input : 2 + 3 * 4

Parse Tree :

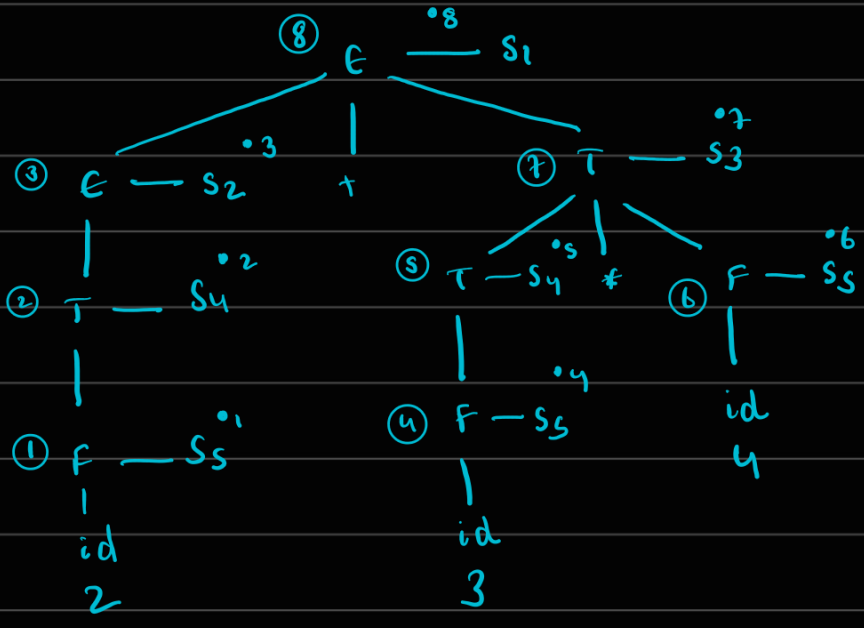


② Infix to Postfix :

$E \rightarrow E+T$	{ print("+") }	S_1
T	{ } (no action)	S_2
$T \rightarrow T*F$	{ print("*") }	S_3
F	{ } (no action)	S_4
$F \rightarrow id$	{ print(id.lval) }	S_5

Input : 2 + 3 * 4

Parse Tree :



Top-down parser evaluation: (top-down, left-to-right)

follow •

$S_5 \rightarrow S_4 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4 \rightarrow S_5 \rightarrow S_3 \rightarrow S_1$

Output : 2 3 4 * +

Bottom-up parser evaluation: (on REDUCE, execute action)

follow •

$S_5 \rightarrow S_4 \rightarrow S_2 \rightarrow S_5 \rightarrow S_4 \rightarrow S_5 \rightarrow S_3 \rightarrow S_1$

Output : 2 3 4 * +

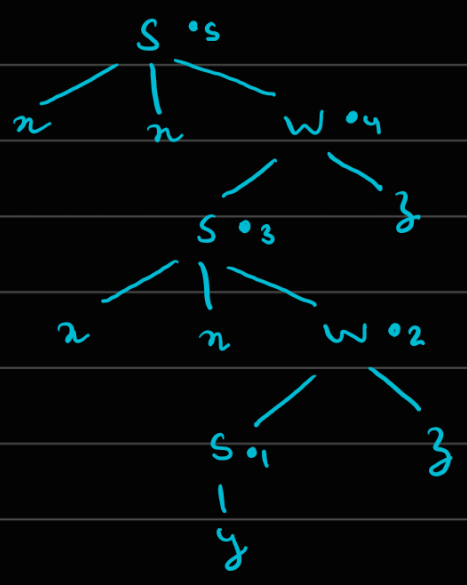
③ $S \rightarrow xnW$ | { print(1) }

y { print(2) }

$W \rightarrow Sz$ { print(3) }

Input : $xxxxyzz$

Parse Tree :



Output : 2 3 1 3 1

④ $E \rightarrow E \# T$ | { E.val = E.val * T.val }

T { E.val = T.val }

$T \rightarrow T \& F$ | { T.val = T.val + F.val }

F { T.val = F.val }

$F \rightarrow num$ { F.val = num.lval }

⇒ Note that '+' has more precedence than '*' and '+' and '*' are left associative as the corresponding prod. rules are left recursive

Input : 2 # 3 & 5 # 6 & 4

⇒ 2 * 3 + 5 * 6 + 4

⇒ 2 * (3+5) * (6+4)

⇒ 2 * 8 * 10 = 16 * 10 = 160

- SDT to build abstract syntax tree (AST) :

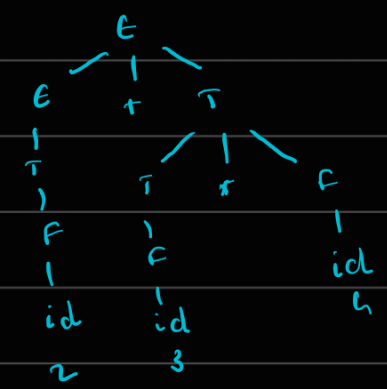
  - Concrete syntax tree (CST) : has more info in each node
  - Abstract syntax tree (AST) : is a simplified CST.

for eg. $E \to E+T \mid T$; $T \to T*F \mid F$; $F \to id$ and

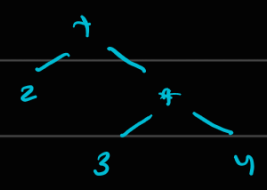
input : 2 + 3 * 4

aka annotated parse tree →

CST :



AST :



The SDT to make this conversion :

$E \to E+T \mid$	$\{ E.nptr = mknode(E.nptr, "+", T.nptr) \}$
T	$\{ E.nptr = T.nptr \}$
$T \to T*F \mid$	$\{ T.nptr = mknode(T.nptr, "*", F.ptr) \}$
F	$\{ T.nptr = F.nptr \}$
$F \to id$	$\{ F.nptr = mknode(null, id.name, null) \}$

[nptr → are pointers]

Notice that it has 3 child ptrs. This can be used to build intermediate code as well

Type Checking

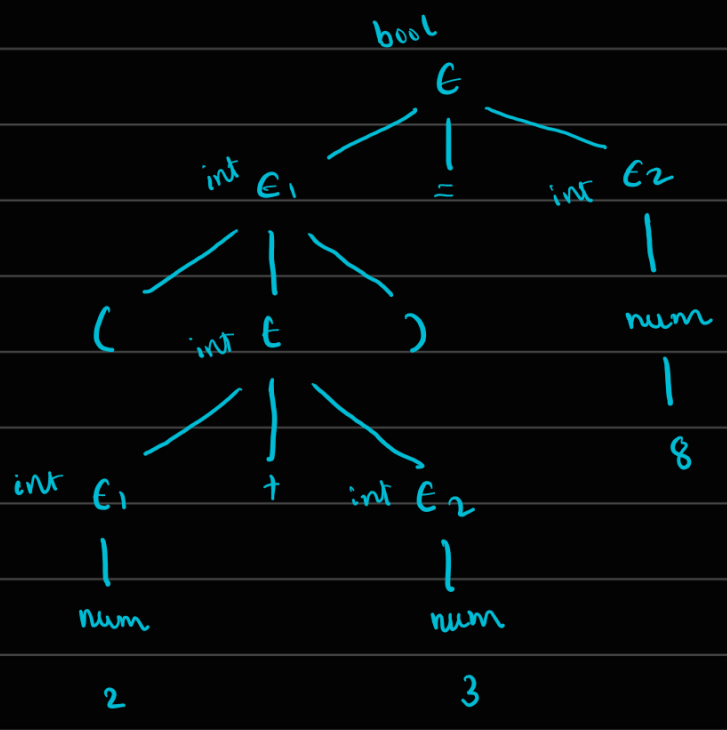
- Use SDTs for type checking. For eg.

Prod. Rule	Semantic Rule Attached
$E \rightarrow E_1 + E_2$	if ((E_1.type == E_2.type) && (E_1.type == int)) then E.type = int else then ERROR
$E_1 = E_2$	if ((E_1.type == E_2.type) && (E_1.type == int || E_1.type == bool)) then E.type = bool else then ERROR
(E_1)	E.type = E_1.type
num	E.type = int
true	E.type = bool
false	E.type = bool

[Obviously, here E_1 and E_2 are E. The subscripts are added for the purpose of differentiating b/w LHS and terms of RHS]

Input : (2+3) == 8

Parse Tree :



3 - Address Code Generation

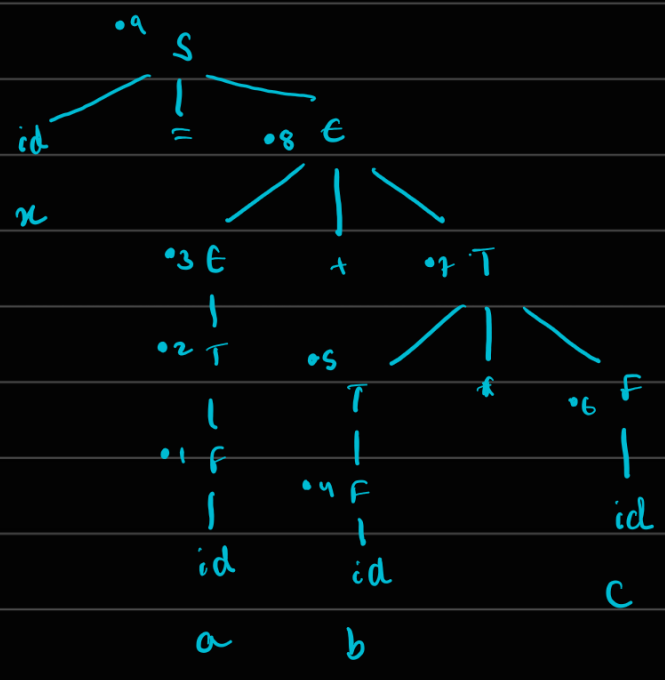
- Use SDTs to generate 3-address code. For eg.

$S \rightarrow id = E$	{ gen (id.name = E.place) }
$E \rightarrow E_1 + T$ |	{ E.place = new temp() ; gen (E.place = E_1.place + T.place) }
T	{ E.place = T.place }
$T \rightarrow T_1 * F$ |	{ T.place = new temp() ; gen (T.place = T_1.place * F.place) }
F	{ T.place = F.place }
$f \rightarrow$ id	{ F.place = id.name }

[place means placeholder. Again, E_1 is E and T_1 is T
Subscript is used for differentiation]

Input : $x = a + b * c$

Parse Tree :



•1 F.place = id.name = a

•2 T.place = F.place = a

•3 E.place = T.place = a

•4 F.place = id.name = b

•5 T.place = F.place = b

•6 F.place = id.name = c

•7 T.place = new Temp() = t_1

gen (T.place = T_1.place * F.place)

⇒ $t_1 = b * c$

•8. E.place = new Temp() = t_2

gen (E.place = E_1.place + T.place)

⇒ $t_2 = a + t_2$

•9 gen (id.name = E.place)

⇒ $x = t_2$

Output :

$t_1 = b * c$

$t_2 = a + t_2$

$x = t_2$

Types of SDTs

- **Types of Attributes :**

- If a node's attribute is taken from its children, it is called as a synthesized attribute. Eg.

  $A \rightarrow BCD$ and

  $A.val = f(B.v_1, C.v_3, D.v_3)$ where f is some function and v_1, v_2 and v_3 are some attributes

  $\Rightarrow$ A is a synthesized attribute

- If a node's attribute is taken from its parent OR its siblings, it is called as an inherited attribute. Eg.

  $A \rightarrow BCD$ and

  $C.val = A.v_1$ OR

  $C.val = B.v_2$ OR

  $C.val = D.v_3$

  v_1, v_2 & v_3 are some attributes

  $\Rightarrow$ C is an inherited attribute

- **Types of SDTs :**

S-attributed SDT

•1 Use only synthesized attributes

•2 Semantic actions are placed at the RIGHT end of the RHS of the prod. rule. Eg.

$A \rightarrow BCD \quad \{ \ldots \}$

•3 Attributes are evaluated during bottom-up parsing

Explanation :

L-attributed SDT

•1 Use both synthesized and inherited attributes. Each inherited attribute is restricted to inherit either from parent or LEFT sibling only. Eg.

$A \rightarrow XY2$; Semantic rules :

$Y.val = A.v_1$ ✓

$Y.val = X.v_2$ ✓

$Y.val = Z.v_3$ ✗

Since Z is a right sibling of Y

(v_1, v_2 and v_3 are some attributes)

•2 Semantic rules are placed anywhere on the RHS of the prod. rule. Eg.

$A \rightarrow \{ \ldots \} BCD$

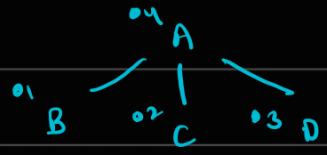
$A \rightarrow B \{ \ldots \} CD$

$A \rightarrow BC \{ \ldots \} D$

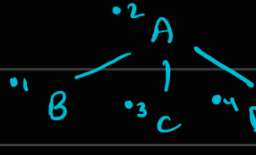
$A \rightarrow BCD \{ \ldots \}$

•3 Attributes are evaluated by traversing the parse tree depth-first, left to right

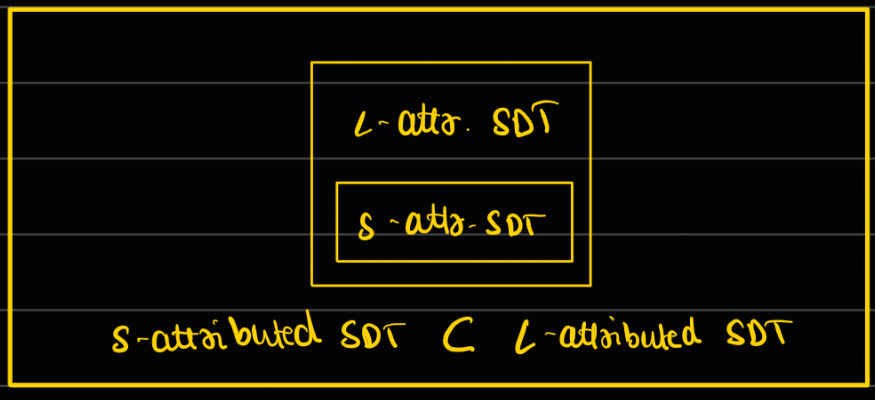
Explanation :



A (parent) requires the attributes of its children (B, C and D)



because C should be able to access the attributes of A (parent) and B (left sibling)



Eg. ① $A \rightarrow LM$ { $L.val = f(A.val)$; $M.val = f(A.val)$; $A.val = f(M.val)$ } (Syn. / Inh. / → Inh.)

⇒ Not S-attr. because we have Inh. attr.

⇒ Is L-attr. because we have both Syn. and Inh. attr.

② $A \rightarrow QR$ { $R.val = f(A.val)$; $Q.val = f(R.val)$; $A.val = f(Q.val)$ } (Inh / → Inh)

⇒ Not S-attr. because we have Inh. attr.

⇒ Not L-attr. because of rule $Q.val = f(R.val)$ since R is a RIGHT sibling of Q.

Symbol Table

- Again, we can use SDTs to store various info. to the symbol table. One example to store variable and its data type:

$D \rightarrow D_1$, id	|	{ addtype(id, D_1.type); D.type = D_1.type }
T, id		{ addtype(id, T.type); D.type = T.type }
$T \rightarrow$ int	|	{ T.type = int }
char	|	{ T.type = char }
float		{ T.type = float }

Input: int x, y, z

Parse Tree:

Symbol Table:

Variable	Data Type
x	int
y	int
z	int

Reduction at steps: •1, •4, •6 and •9