Syntax Directed Translation

Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser. In other words, the parsing process and parse trees are used to direct semantic analysis and the translation of the source program to translate and evaluate .

Basically we try to integrate the generation of parse tree and the evaluation by traversing the parse tree .

Production : A -> XY

Semantic Rule : A.a = f(X.b,Y.c)

where a is attribute associated to A ,b is attribute associated with X and c is attribute associated with Y .

SDT : XY { Action or code program fragment }

In above SDT rule the position of action varies as per the type of SDD ( L-attributed or S-attributed ) and the type of attribute ( synthesized or inherited ) to be evaluated .

SDT for infix to postfix conversion of expression for given grammar :

Grammar :

E -> E + T { print(‘+’) }

E -> E – T { print(‘-‘) }

E -> T { }

T -> id { print(‘id’) }

SDT for evaluation of expression for given grammar :

Grammar :

L -> E { print(E.val) }

E -> E + T { El.val = Er.val + T.val }

E -> T { E.val = T.val }

T -> T \* F { Tl.val = Tr.val \* F.val }

T -> F { T.val = F.val }

F -> id { F.val = id.lexval }

Get Equivalent SDT from :

1. S – Attributed SDD :

For all semantic rules , generate action to compute synthesized attribute of Non-Terminal in the head of production from the synthesized attributes of Non – Terminals in the body of production .

Rules :

1. Translate those semantic rules to equivalent code fragments or Action .
2. Place those actions at the end of the production.

Above SDT is an example of conversion of S-attributed SDD to equivalent SDT .We always append the action to the end of the production and this type of SDT is called **Postfix SDT** .

To evaluate the expressions we use value stack for evaluation along with symbol stack which is used for parsing . Using above SDT we translate the i/p expression : 3\*4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Symbol Stack | Value Stack | Input String | Syntax Action | Semantic Action |
| $ | $ | 3\*4 | Shift |  |
| $id | $3 | \*4 | Reduce  F -> id | F.val = id.val |
| $F | $3 | \*4 | Reduce  T -> F | T.val = F.val |
| $T | $3 | \*4 | Shift |  |
| $T\* | $3\* | 4 | Shift |  |
| $T\*id | $3\*4 | $ | Reduce  F -> id | F.val = id.val |
| $T\*F | $3\*4 | $ | Reduce  T -> T\*F | T.val = T.val \*F.val |
| $T | $12 | $ | Reduce  E -> T | E.val = T.val |
| $E | $12 | $ | Reduce  L -> E | Print(E.val) |

1. L-attributed SDD :

Let product stated below is a rule of LL(1) grammar

Rule : A -> X1X2X3

Semantic Rule : X2.b\_inh = F(X1.a\_sym,A.x\_inh)

A.x\_sym = F(X1.a\_sym,X2.b\_sym,X3.c\_sym)

Where all of the Non-terminals may have synthesized and inherited attributes .

SDT :

A -> X1 { X2.b\_inh = f(X1.a\_sym,A.x\_inh) } X2X3

{ A.x\_sym = F(X1.a\_sym,X2.b\_sym,X3.c\_sym) }

To evaluate the synthesized attribute we append action at the end of the production and to evaluate the inherited attribute we append the action before the non-terminal symbol whose attribute has to be calculated .