REPORT ON TUTORIAL (30th October 2013)

Implementation of L – Attributed SDD :

Methods to do translation by traversing a parse tree:

1. Build the parse tree and annotate.
2. Build the parse tree , add actions , and execute the actions in preorder. This works for L-attributed definition.

Methods for translation for translation during parsing:

1. Use a recursive – descent parser with one function for each nonterminal. The function for nonterminal A receives inherited attributed of A as arguments and returns the synthesized attributed of A.
2. Generate code using a recursive – descent parser.
3. Implement a SDT in conjunction with an LL-parser.

Translation during Recursive – Descent Parsing:

A recursive – descent parser has a function A for each nonterminal A.

a) The argument of function A are the inherited attributed of non-terminal A.
b) The return value of function A is the collection of synthesize attributes of non-terminal A.
c) Preserve, in local variable, the values of all attributes needed to compute inherited attribute for non-terminals in the body or synthesize attribute for the head non terminal.
d) Call functions corresponding to non-terminals in the body of the selected production, providing them with the proper arguments.

A -> X_1 X_2 ---------- (1)

Attributes for A: a.syn , a.inh
Attributes for **X_1:** x1.syn Attributes of **X_2:** x2.inh

We know , inherited attributes for non-terminal in the body of production can be the function of inherited of parent and attributes of non-terminal left to it.

x2.inh = f(x1.syn, a.inh)

and, synthesize attributes of the head of the production is the function of synthesized attributes of the children.

a.syn = fun(x1.syn, x2.syn)

Equivalent SDT, for production (1):
A -> X_1 { x2 = f(x.syn, a.inh) } X_2 { a-syn = f(x1.syn, x2.syn) }

1. Inherited attributed rule of the non-terminal of the body of production will come just before that NT in the that production.

2. If A (head of the production) has any synthesized attributed rules, then that will come after all non-terminal of the body of production as the fragmented code.

For example:

T -> F T' { T'.inh = F.val / T.val = T'.syn }
T' -> * F T_1' { T_1'.inh = T'.inh * F.val / T'.syn = T_1'.syn }
T' -> epsilon { T'.syn = T'.inh }
F -> id { F.val = id.val }

Equivalent SDT:

T -> F { T'.inh = F.val } T' { T.val = T'.syn }
T' -> * F { T_1'.inh = T'.inh * F.val } T_1' { T'.syn = T'.inh }
T' -> epsilon { T'.syn = T'.inh }
F -> id { F.val = id.val }

Writing function for a non terminal

```
A( )
{
        Declare : a-syn, x1-syn, x2.inh,  x2-syn;
        If(current symbol (a) == Terminal X1)
                        Move i/p pointer to next symbol of the input string
        Else if ( X2 is NT)
        x1-syn = X1( );
        x2.inh = f ( x1-syn );
        x2-syn =  X2( x2.inh );
        a.syn = f ( x1-syn,x2-syn );
        return a-syn;
}
```

Now, functions for the non-terminals introduced in the above example:

```
F( )
{
    Declaration: F-val;
    If(id == T and Id matched with the input symbol)
            Move input  pointer to the next symbol.
    F.val = id.val ( get from the lexical analyser )
    Return F.val;
}

T( )
{
    If( F matched with the input symbol & F == T)
            Move input  pointer to the next symbol.
    If ( F == NT)
    {
            F-val = F( );
            T'-inh = F-val;
```

```
    }
    If(  T' == a && T' is T )
            Move ptr. To next;
    If( T' == NT )
    {
            T'-syn = T'( T'-inh );
            T-val = T'-syn;
            Return T-val;
}
```