Syntax Tree Implementation

There are some considerations in regards of the implementation of the abstract syntax tree. They are the following:

Consideration 1

The elimination of left recursion in the modified BNF grammar led to the use of inherited attributes, and therefore to the construction of an attribute stack to build the AST. However, the AST was implemented using only synthetized attributes. For instance, the original semantic actions for the construct *term* was:

Note: *.i* and *.s* denote inherited and synthetized values respectively *U* denotes union. Numbers 1 and 2 are only to denote different instances of the same construct.

term --> factor { term'.i = factor.s } term' { term.s = term'.s }
term1' --> multop factor { term2'.i = term1'.s U factor.s U multop.s} term2' { term1'.s = term2'.s }
| *epsilon* { expression'.s = *epsilon* }

According to this attributes, the node construction is performed on term1' subroutine, and then passed back to the original construct *term*.

Nonetheless, it is constructed within the sobroutine *term* after having called both *factor* and *term'*. To do this, *term'* subtree is stored in a temporary node, then the relevant information is retrieved: operation type (multop.name) and the subtrees derivated in *term'* (*term'*.leftchild and *term'*. centerchild). In this manner:

term --> factor term'	<u>case term' did not derive epsilon:</u>
	{ term.s = maketree(multop.name+"main", factor.s, term'.leftchild, term'.centerchild) }
	<u>case term' derived epsilon:</u>
	{ term.s = factor.s }
term' --> multop factor term'	{ term'.s = maketree(multop.op, factor, term') }
| *epsilon*	{ term'.s = *epsilon* }

So, only synthetized values are used. This is also the case for *add-exp* and *expression* constructs.

Consideration 2. Nodes are implemented this way:

NODE
|------ child1 field ---> subtree attached

|------ child2 field ---> subtree attached
|------ child3 field ---> subtree attached
|------ leaf field ---> EMPTY!

Consideration 3. Leaves are implemented like this:

NODE
|------ child1 field ---> empty
|------ child2 field ---> empty
|------ child3 field ---> empty
|------ leaf field ---> LEAF
|---- token_name field ---> token_name (ID or BLIT or NUM)
|---- table_entry field -----> TABLE_ENTRY
|-------- lexeme ---->
|-------- index ----->
|-------- lineno ---->

The reason for this, is that I couldn't find a way to set a leaf data structure directly to a node's child fields, because childs are supposed to be node data structures, not leaves.

So, it was added a leaf field to the original node data structure, allowing me to insert one leaf per node. Nevertheless, after inserting the leaf, it would contain no information about the position of the leaf in the node father, that is, whether the leaf is attached as child1, child2 or child3. Also what if it is needed more leaves per node? (although it doesn't happen with c11 grammar).

Thus, the leaf is inserted in the leaf field of an empty node. Then the empty node is attached to the desired positon -child1, child2 or child3- in the father node.

For example, consider the following construct and its semantic action:

nid-factor --> num { nid-factor.s = makeleaf("num", num.table_entry) }

The code for this is:

```
//matches NUM, and gets a temporary token of the matched NUM... this is so, because
//match(token) function changes the lookahead to the next one in the stream of tokens.
auxtkn = match( NUM );

// a leaf is created with the corresponding table entry information, also it's type is
// setted to int
pleaf = makeleaf( LBL_NUM, auxtkn.table_entry );
settype_leaf( pleaf, TYPE_INT );

//finally, the leaf is attached to an empty node and returned, so it can be attached to any
//position of the node father
return maketree( LBL_NUM, NULL, NULL, NULL, pleaf );
```

So, instead of returning a leaf data structure:

NUM (leaf)
|--------> token_name

|--------> table_entry

It actually returns this:

NUM (node)
|------> empty
|------> empty
|------> empty
|------> NUM (leaf)
|------ token_name
|------ table_entry

Consideration 4

Although in the grammar many constructs are stablished as leaves (REF, VOID, EPSILON, etc.), only those who have a table entry are implemented as such (as explained in consideration 3) . They are: ID, NUM, BLIT.

The rest are implemented as empty nodes.

Therefore, leaves implemented as empty nodes will look like this:

NODE
|------> empty
|------> empty
|------> empty

For instance:

REF
|------> empty
|------> empty
|------> empty

EPSILON
|------> empty
|------> empty
|------> empty