REPORT ON ASSIGNMENT 3 COMP 30330 Nicolas Smith 11463982

Part 1: Modify the grammar

Given:

S ::= begin Decls Stmts end
Decls ::= Decls Decl | Decl
Decl ::= int Vars ; | bool Vars ;
Vars ::= id | Vars , id
Stmts ::= Stmt | Stmts ; Stmt
Stmt ::= Assn | Cond
Assn ::= id := Arith | id := Logical
Arith ::= Term | Arith + Term
Term ::= Factor | Term * Factor
Factor ::= id | num | (Arith) | - Factor
Cond ::= if Logical then S
Logical ::= [Comp]
Comp ::= Arith = Arith | Arith > Arith | Arith < Arith

Altered:

S ::=	begin Decls Stmts end
Decls ::=	Decl D'
D' ::=	Decl D' | EPSILON
Decl ::=	int Vars ; | bool Vars ;
Vars ::=	id V'
V' ::=	, id V' | EPSILON
Stmts ::=	Stmt ST | Stmts ; Stmt
ST::= ;	Stmt ST | EPSILON
Stmt ::=	Assn | Cond
Assn ::=	id := Arith | id := Logical
Arith ::=	Term AR
AR::=	+Term AR | EPSILON
Term ::=	Factor TR
TR::=	*FactorTR | EPSILON
Factor ::=	F | - F
F::=	id | num | (Arith)
Cond ::=	if Logical then S
Logical ::=	[Comp]
Comp ::=	Arith = Arith | Arith > Arith | Arith < Arith

Part 2: JavaCC & JJTree

In Eclipse one is given the option between a JavaCC file and a JJTree file template to use, the second one should of course be selected. These files generate code for a Lexer, Parser and Parse tree given some Grammar rules. The first step was to define all of our Terminals using regular expressions, for example:

```
< ID : (< LETTER >)+>
|      <#LETTER : ["a"-"z" , "A"-"Z"] >
```

Defines an Identifier in our Grammar.

The grammar rules for the parser then have to be established, at this stage we only want that it should recognise correct input correctly, we don't want to assign values to anything yet. Here is a simple example of how productions are dealt with in JavaCC:

```
void v() :
{}
{
        ( < COMMA > <ID > v()
                )*
}
```
This corresponds to the production v::= , id v'

Testing with our sample leads to the following result:
```
Reading from standard input...
Enter an expression like:begin
 int tom, dick; bool harry; tom := 55;
 dick:=tom*123+-42; harry:=[(tom+7)>222];
 if [tom>3] then begin int myconst; dick:=dick+tom*2+1 end
 end
Start
 decls
  decl
   vars
    v
     v
  d
   decls
    decl
     vars
      v
    d
```

… this goes on for quite a while so I shall not subject you to another full page of a printed parse tree. The important thing is that it works.

Part 3: Visitor

In theory this seems rather easy, simply write a Visitor that will traverse the tree and create instructions in assembly language based on the values it finds. But theory alas does not always correlate to reality.
The complex structure of JJTree projects makes performing this should-be simple task somewhat difficult.
Here is an example of how our Visitor is meant to act at, let's say, and assignment node:
Take the id being assigned and find it's location (destination), find value being loaded to it or its location(source). Create a vector of strings, the first element will be the string “load”, the second a string representing the destination register and the last either the name of the identifier aor the location it will take it's value from.

At the moment my Visitor will get the first three instructions correctly before crashing:

```
 if [tom>3] then begin int myconst; dick:=di
 end
[load, t1, dick, ] added to instructions
[load, t0, tom, ] added to instructions
[load, t2, harry, ] added to instructions
Oops.
```

After a bit more debugging it finally got down to the ar() node.

```
visited: assn
[loadliteral, t3, 55, ] added to instructions
ar
visited: arith
Oops.
```

Unfortunately due to time constraints and other assignments I was unable to continue debugging as the submission date was nigh.