CS 441 – Fall 2018

Project 3: The Constrainer

Due: Wed Dec 12, midnight

30 points

Learning Objective:

- Constraining a parsed program by:
  - Checking for undefined variables
  - Type checking on assignment statements

Given Code:

proj3.zip can be found on the course website. You may also find all files on multilab (not zipped), along with my executable (comp), at ~kwjoiner/441/proj3/.

These contain all given code needed to start the project. All changes will be made in *constrainer.cpp*, though you may make other changes if you like.

If you want to use the code you've done this semester, you'll need to at least use the given *grammar.h* and *token.h/.cpp* as the grammar has been changed. You will also need to add the two invocations of the *constrain()* method to your parser (see the given parser.cpp). A new error message has also been added to the parser.

Overall Design:

When needed, new "constraint" non-terminals have been added to the grammar in some of the original productions. In the grammar, all of these are in the form $N \rightarrow \in$ so these do not change the phrase-structure of the grammar (note some other new productions, dealing with *writeln* have been added, but these will not be used in this project).

The purpose of these new non-terminals (the constants all begin with NTC_) is to trigger some action in the constrainer.

In the parser, tokens popped off the stack or tokens from the input file are constrained when they are consumed; that is, they are passed to the constrainer for further checking. Not all non-terminals or terminals will trigger action by the constrainer.

The overall design of the constrainer is then a *switch()* statement to decide what action needs to be taken depending on the token ID of the token just "consumed."

Some actions are simply to "remember" some information from the current token that will be needed when constraining future tokens. The primary method of "remembering" is to insert identifiers into a symbol table (provided/coded). Some information about an identifier will be parsed in "future" tokens, so information on identifers is built as it arrives in the constrainer.

Example:

```
VAR I,J,K : INTEGER;
```

The names of the variables arrive before the data type. So they will be inserted into the symbol table when they are parsed with an "unknown" data type. Later when the INTEGER arrives, all symbols with unknown data types will be changed to integer.

There are two other data members used to "remember":

- bool insertIdents;
  The idea is that parts of a Pascal program are *declarative* of identifiers (variable declarations, procedure names found in procedure headers, the program name, etc.) while others are *executable.* When parsing declarative sections, identifiers encountered are inserted into the symbol table. Redeclaration errors are detected during this phase. When parsing executable sections, checks are made to ensure identifiers are declared and that data types are used correctly.
  This data member maintains the current "phase", and the constraining of certain non-terminals triggers the turning on/off of this flag. The value of the flag then dictates the constraining done on identifiers and literals when they arrive in the constrainer.

- int constType
  This member is used to do type checking for assignment statements. When the constraining of an assignment statement begins, the data type of the LHS (variable on the Left Hand Side of the :=) is "remembered" by this data member. As literals/variables in the RHS are parsed, they are checked to be identical to that of the LHS. In our Pascal, strict type checking is enforced. (so not even float x = 0; is allowed…it must be float x = 0.0;).
  This data member is set to 0 when any non-assignment statements are being parsed.

Specifications:
Again, all changes (additions) to the code are in *constrainer.cpp*. Some parts of the switch() have been coded. Your job is to finish the code according to the comments given. Look for all **// TO DO:** comments and follow the instructions there.

Submission:
The same instructions as used for all projects this semester: all source files and a makefile, or compile with **g++ -o compile.out *.cpp**