CSCI 4160 Project3

Due: see class calendar

Goal:

This assignment serves several purposes:

- to be familiar with Bison,
- to create the context free grammar for COOL language.

Description:

This is a project that builds on the previous project. In this project you are required to use Bison to build a parser for the COOL language. More specifically, write the context free grammar for the Tiger language. Please read COOL language manual carefully before you start. The manual can be found in the D2L Content | Module 6: class repository.

What to do in this project?

Come up with context free grammar for COOL language and use Bison to generate the parser for it. Your grammar should have as few shift-reduce conflicts as possible, and no reduce-reduce conflicts. When you compile your parser file, Bison will output information about the number of shift-reduce conflicts as well as the number of reduce-reduce conflicts. If your grammar has a reduce-reduce conflict, please talk to me to see how to get rid of it.

The precedence level and association of operators in COOL can be found on page 16 of the manual.

Extra rules should be provided to recover from errors. The grammar you provided should generate the same output on the sample file cool1.cl as the one provided by the instructor. **No actions needed for rules in this assignment.**

Most COOL CFG can be found in page 17 of the manual. However, some changes are required due to the inconsistencies between the CFG given in the manual and the scanner used in the project. Please take the following actions when working on the project.

1. Different token names are used, for example, TYPE and ID used in the manual, but TYPEID and OBJECTID used in the scanner.
   replace TYPE with TYPEID
   replace ID with OBJECTID
   replace *integer* with INT_CONST
   replace *string* with STR_CONST
   replace *true/false* with BOOL_CONST
2. For tokens with two or more characters, use token names instead of characters. For example, the rule *ID <- expr* should be written as *OBJECTID ASSIGN expr* in your project. Similarly, use DARROW for => and LE for <=.
3. For tokens with a single character like +, please use single quote to enclose it since tokens are supposed to be integers (including char). So the following rule *expr + expr* should be written as *expr '+' expr*. Should do similar things for all token of single characters like: + / - * = < . ~ , ; : ( ) @ } { .
4. The manual uses the following CFG extensions.
   - [[…]]* represents repetition of zero or more times. It typically represents a list of things although the list maybe empty. For example, [[feature;]]* represents a list (maybe empty) of features and each feature is terminated by a semicolon.
   - [[…]]$^{+}$ represents repetition of one or more times. It typically represents a list of at least one thing. For example, [[expr;]]$^{+}$ represents a list of at least one expressions and each expression is terminated by a semicolon.
   - [..] means optional. It appears zero or one time.

The following picture is used to demonstrate how to convert CFG extensions to regular CFG productions in your project.

class ::= class TYPE [inherits TYPE] { [[*feature*;]]* }
feature ::= ID([*formal* [[, *formal*]]*]) : TYPE { *expr* }
| ID : TYPE [<- *expr*]

- [[feature;]]* : a list of features and each feature is terminated by a semicolon. Since the list may be empty, we introduce two nonterminals: optional_feature_list and feature_list. Feature_list represents a non-empty list of features.

```
optional_feature_list    :    //empty body for this rule
                     | feature_list
                     ;
feature_list         : feature ';'
                     | feature ';' feature_list
                     ;
```

- [formal [[, formal]]*]: the whole thing is optional, and , formal should be repeated 0 or more times. The whole thing represents a list (maybe empty) of formal separated by comma. What's the difference between the previous one and this one? The previous feature list requires each element to be terminated by semicolon (i.e. the last element should be followed by a semicolon), while the list of formals requires element to be separated by comma (i.e. the last element should have no comma).

```
optional_formal_list : /* empty body */
                     | formal_list
                     ;
formal_list          : formal
                     | formal ',' formal_list
                     ;
```

- In the project, you will see two kinds of lists of expressions like the following:
  1,2,3 (This list is used to pass arguments to method calls)
  1;2;3; (This list is used to specify a sequence of expressions enclosed by curly braces.
  You should use different non-terminals to represent the above two kinds of list.)

- You should have some error recovery rules to make sure your output on COOL1.cl is the same as mine. You need to understand what the errors are in COOL1.cl and then decide where to put the error recovery rules. For example, if there is a syntax error in argument list of a method call, you want to have an error recovery rule for the method definition, by using ***error*** token to replace the argument list.

Set up environment:
I strongly suggest you use the sample Visual Studio solution provided by me.

1. There is no more cool.ll file since the instructor provides lex.yy.cc file.
2. Put your grammars in cool.yy file, which is the only file you need to work on.
3. To compile your project,
   a. Compile cool.yy.
   b. Build the MainDriver project.
4. There are two ways to debug your grammar,
   a. In the sample solution provided by the instructor, there is a file "cool.output" under "FlexBison Tools"\Resource Files within Visual Studio environment. This file is

generated every time cool.yy is parsed. It contains all state information of your grammars.

b. Reset yydebug to 1 at line 29 at main.cpp file. This will generate debug information on terminal when compiling a tiger file. However, the debug information is probably hard to understand.

The project also provides for Linux platform and the following shows the instructions to compile the project in Linux platform.

--work on COOL.yy file
make --compile your project using make command
./main cool0.cl --run your program against cool0.cl, which can be replaced by other COOL files

You can also use Linux version to work on MacOS with the following changes:

- make sure latest version of Flex and Bison are installed on your machine.
- change "g++" at line 9 of **makefile** to the compiler you want to use.

The compilation and execution should be similar to the Linux version.

If you want to know more about make and makefile, please check this page.

Instructor provided files in the class repository

The following files are provided by the instructor:

- Skeleton source files provided in the sample project are listed below:
  - lex.yy.cc: provided by instructor
  - cool.yy: a skeleton file for tiger CFG.
  - cool.tab.hh & cool.tab.cc: generated by Bison when compiling cool.yy.
  - cool.output: debug file for CFG
  - ErrorMsg.h: contains the definition of error handler
  - main.cpp: the driver
  - cool0.tig and cool1.tig: test cases of COOL language
- Description3.pdf: this file
- Rubric3.doc: the rubric used to grade this assignment.
- cool0.txt and cool1.txt: expected output for cool0.cl and cool1.cl, respectively.

How to submit

Please submit your solution to D2L dropbox.