CSCI 4160 Project4

Due: see class calendar

Goal:

This assignment is used to create the Abstract Syntax Tree for COOL language.

Description:

This is an individual project that build on the previous project. What you need to do in this assignment is to define actions for each production of COOL language defined in COOL.yy. These actions will create the abstract syntax tree (AST). Before defining production actions, make sure you understand the following classes, which define type of nodes in the AST. You can find all class definitions in Absyn.h. Most likely, you will use constructors of these classes in the project.

class TreeNode: This abstract base class is the ancestor of all nodes in AST. It contains two member data: lineno and colon, which represents the location information of the construct stored in the node.

class List: Defines a list class. It will be used to define a list of expressions/formal parameters/methods/attributes.

class Program_class: Represents a COOL program as a list of classes
class Class_class: Represents a COOL class
class Feature_class: Represents a feature (i.e. method or attribute) of a COOL class

- class Method: Represents a method of a COOL class
- class Attr: Represents an attribute of a COOL class

class Formal_class: Represents a formal parameter in a method
class Branch_class: Represents a branch of a case expression

class Expression_class: Represents a COOL expression

- class AssignExp; represents assignment expression like: x <- 5 + 9;
- class CallExp; represents a method call like: an_object.gcd(x, 20);
- class StaticCallExp; represents a static method call: an_object@parent_type.gcd(x, 20);
- class IfExp; represents an if-else expression
- class WhileExp; represents a while expression.
- class CaseExp; represents a case expression
- class BlockExp; represents a block expression
- class LetExp; represents a let expression
- class OpExp; represents the expressions involving a binary operator, for example: x < y
- class NotExp; represents the negative of a Boolean expression
- class IntExp; represents an integer literal like: 10
- class StringExp; represents a string literal like “This is a string literal”.
- class BoolExp; represents a boolean literal like true or false
- class NewExp; represents a COOL expression to create a new object like: new a_className.
- class IsvoidExp; represents the expression to check if the value of an expression is void or not, like: isvoid(a_expression)
- class ObjectExp: represents an object
- class NoExp: represents a NIL expression.

Set up environment:

I strongly suggest you use the sample Visual Studio solution provided by me.

1. Use lex.yy.cc provided by the instructor.

2. Correct errors in COOL.yy from previous assignment.
3. Add actions to each grammar rule in COOL.yy file, which is the only file you need to work on. Please ignore ALL PRODUCTIONS THAT PERFORM ERROR RECOVERY.
4. To compile your project,
    a. Compile cool.yy.
    b. Build the MainDriver project.

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 example.cl   --run your program against example.cl
```

If you want to use MacOS for the project, you can download the Linux version. Please make sure latest version of Flex and Bison are installed on your machine. You may need to 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.

Tips for the project:

- Due to the definition of class LetExp in Absyn.h file, it is better to define the let expression as the following:

```
optional_initialization :       /* Empty */
                        |       ASSIGN expr
                        ;

let_list:       OBJECTID ':' TYPEID optional_initialization IN expr %prec LET_STMT
        |       OBJECTID ':' TYPEID optional_initialization ',' let_list
        ;
expr    :       LET let_list
```

- The following productions can be used to specify case expressions:

```
expr :          CASE expr OF case_list ESAC
case_list:      simple_case    /* One branch */
        |       simple_case case_list
        ;

simple_case : OBJECTID ':' TYPEID  DARROW expr ';'
            ;
```

- Due to the definition of List class in Absyn.h, it is better to use right recursion in the CFG. So if you have production with left recursion like the following (i.e. production head appears as the first symbol in the production body of a rule):

```
expr_list_comma     : expr
                    | expr_list_comma ',' expr
```

You need to rewrite it as right recursion like the following:

```
expr_list_comma     : expr
                    | expr ','  expr_list_comma
```

- In this project, the action for almost all rules is to construct the parse trees represented by the production head. More specifically, the action should look like: $$ = …., i.e. trying to assign a

value to $$, the value associated to the production head. Before implementing the action for a rule, please make sure you understand the type of value associated to the production head. Most actions are just one statement.

- The production body for certain rules is empty, such as
  Optional_expr_list: / empty body */ { $$ = nullptr; }*
  | expr ';' expr_list
  ;
  For such rules, the action should be: $$ = nullptr;

- For production **expr : ~ expr** where ~ represents complement of expr. You should interpret it as **0 – expr**. In the action, you should use: *new IntExp(@1.first_line, @1.first_column, inttable.add_int(0))* to represent 0.

- The production: **expr -> OBJECTID '('optional_expr_list_with_comma ')'** represents a method call. But in the constructor of CallExp, an object is required. Semantically, it should be the object through which the method is invoked. This object is represented by the key word 'self'. So in the action of this rule, you need to construct a self_obj first as shown below:

  Expression self_obj = new absyn::ObjectExp(@1.first_line, @1.first_column, idtable.add_string("self"));

- Please use provided single_list and pair_list functions to construct list.

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
  - COOL.yy: a skeleton file for COOL CFG. Please follow the instructions to modify it.
  - 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
  - Dump.cpp: used to print the abstract syntax tree.
  - Absyn.h: contains class definition for all AST node types.
  - AbsynExtension.h: contains extensions to nodes defined in AST
  - StringTab.h and StringTab.cpp: contains definition of string table
  - main.cpp: the driver
  - example.cl: test program of COOL language
- Description4.pdf: this file
- Rubric4.doc: the rubric used to grade this assignment.
- example.txt. sample output when parsing example.cl file

How to submit

Please submit COOL.yy only to D2L dropbox.