1. Description

Bingbing Rao Robin.Rao@knights.ucf.edu

This is an implementation of a Compiler for PL/0 program language.

Directories and files:

```
|-root
    |---bin : This directory stores the executing files generated by Gcc compiler.
    |---input :This directory contains the test files as the input of c-unit.
    |---obj :This directory contains objective files generated by gcc compiler.
    |---ouput :This directory contains the output of PL/0 Compiler.
    |    |---ast.txt    : the information about abstract syntax tree of input source program.
    |    |---machinecode.txt    :the final machine code of input source program.
    |    |---symbol_table.txt    :the symbol table of input source program.
    |    |---tokenfile.txt    :the token file of the input program.
    |    |---unparsemachinecode.txt    : the assemble language of the input source program.
    |    |---unparsesrc.txt    : unparse the source code.
    |    |---virtualmachine.txt : the stack information about virtual machine.
    |
    |---src :
    |    |---dfa    :the source code for Deterministic finite automaton(DFA) model
    |    |---driver    :the source code for compiler driver
    |    |---gencode    :the source code for code generator model
    |    |---include    :the head files
    |    |---parse    :the source code for parser model
    |    |---scanner    :the source code for scanner model
    |    |---symboltable    :the source code for symbol table model
    |    |---test    :the source code for test programs
    |    |---virtualmachine    :the source code for vitrual machine model
    |    |---Makefile    :the second level makefile to build all models.
    |
    |---Makefile    :The root level makefile to build whole project.
    |---README.md    :Documents about this project.
```

To run the examples, the followed tools should be installed in your environment:

- **Unix/Linux compile environment**
- **gcc** (>4.8.0)
- **make**
- **Markdown Reviewer**

2. Finished Work

In this project, I have implemented the following components of a compiler for PL/0 program language.

- **Virtual Machine**
    - Using 16 common registers to facilitate the process of virtual machine;
    - handle the function or method invoke;
    - output to a specific file or system stdout;
    - When we print the stack info, using "|" symbol to separate each AR;
- **Symbol Table**
    - use hashtable to store symbol entry.

- look up a symbol with a specific key.
- for each key in a symbol, there is a struct named value to store its attributes.
- Using a key of symbol to get its attribute.
- get the hash value for each symbol.
- judge a symobl exist via the key.
- update the attribute value of a symbol in the table.

- **Scanner**

  - DFA machine to recognize a string to generate a lexeme.
  - Using a list to store the lexemes;
  - Print the lexemes with source code;
  - Parsing a symbol from the list of lexemes and then using a hashtable to store them.

- **Parser**

  - Build an abstract syntax tree for input source program.
  - output the AST information of the input source program.
  - tranverse the AST.
  - unparse the input source code and output to console or a file.
  - Check Syntax error and provide debug infor to user to locate error.

- **CodeGenerator**

  - Generate machine code for each procedure.
  - Combine machine code of each procedure into final machine code.
  - API to generate machine code for each instruction.
  - unparse machine code to assemble code.
  - output the assemble code to console or a file.

- **Compiler Driver**

  - Check the input parameters and output the status of each models.
  - According to the user's directives to ouput the information to console

- **Test Cases**

  - Unit test for scanner
  - Unit test for parser.
  - Unit test for virtual machine.

3. User Mannual

In this section, we will describe how to build whole project and run PL/0 compiler to compile your source program.

The root Makefile supports the following targets:

- **make all** build all targets (build,compile,test)
- **make build:** Enter the subdirectory recursively and compile, then the objetive file will ouput to the "obj" directory;
- **make test:** Run all test cases;
- **make compile** Build whole project and generate the PL/0 compile executing file.

3.1 How to build whole project

Using the following command to build whole project.

1. Enter the root directory of project
2. Run the following commands to build whole project
   "make"|"make all"|"make build"

3.2 How to build a model

1. Enter the root directory of project
2. Using the following command to build a model(DFA)
   make -c src/dfa

3.3 How to generate PL/0 compiler execute file

1. Enter the root directory of project
2. Using the following command to generate executer file
   make compile

3.4 How to build and execute test unite

1. Enter the root directory of project
2. Using the following command to build and execute test unite
   make test
   make test_scanner
   make test_parser
   make test_vm

3.5 Directive of PL/0 compiler

1. Using command methionded in section 3.3 to build a compiler execute file.

2. According to the following instruction to run compiler.
   Usage: compile [OPTION] ... [FILE] ...
   Option:

   - -h :print out the help info
   - -i inFile: inFile\tspecific the input files that compiler will execute
   - -l : print the list of lexemes/tokens (scanner output) to the screen
   - -a : print the generated assembly code (parser/codegen output) to the screen
   - -v : print virtual machine execution trace (virtual machine output) to the screen
   - -s : print symbol table to the screen
   - -t : print abstract syntax tree to the screen
   - -u : print the result of unparse for source program to the screen

   Example

   - compile -h
   - compile -i input/dfa.txt [-l|-a|-v|-s|-t|-u]

3.6 EBNF of tiny PL/0

```
program ::= block "." .
block ::= const-declaration  var-declaration  statement.
constdeclaration ::= [ “const” ident "=" number {"," ident "=" number} “;"].
var-declaration  ::= [ "var" ident {"," ident} “;"].
statement   ::= [ ident ":=" expression
            | "begin" statement { ";" statement } "end"
            | "if" condition "then" statement
            | "while" condition "do" statement
            | "read" ident
            | "write"  ident
            | e ] .
condition ::= "odd" expression
           | expression  rel-op  expression.
rel-op ::= "="|“<>"|"<"|"<="|">"|">=“.
expression ::= [ "+"|"-"] term { ("+"|"-") term}.
term ::= factor {("*"|"/") factor}.
factor ::= ident | number | "(" expression ")“.
number ::= digit {digit}.
ident ::= letter {letter | digit}.
digit ;;= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9“.
letter ::= "a" | "b" | … | "y" | "z" | "A" | "B" | ... |"Y" | "Z".
```