Compiler Principles and Design

Final Project Report

Student Id	Name
1820176963	Arrival Dwi Sentosa

BIT Mini C Compiler

北 京 理 工 大 学

Beijing Institute of Technology

Introduction

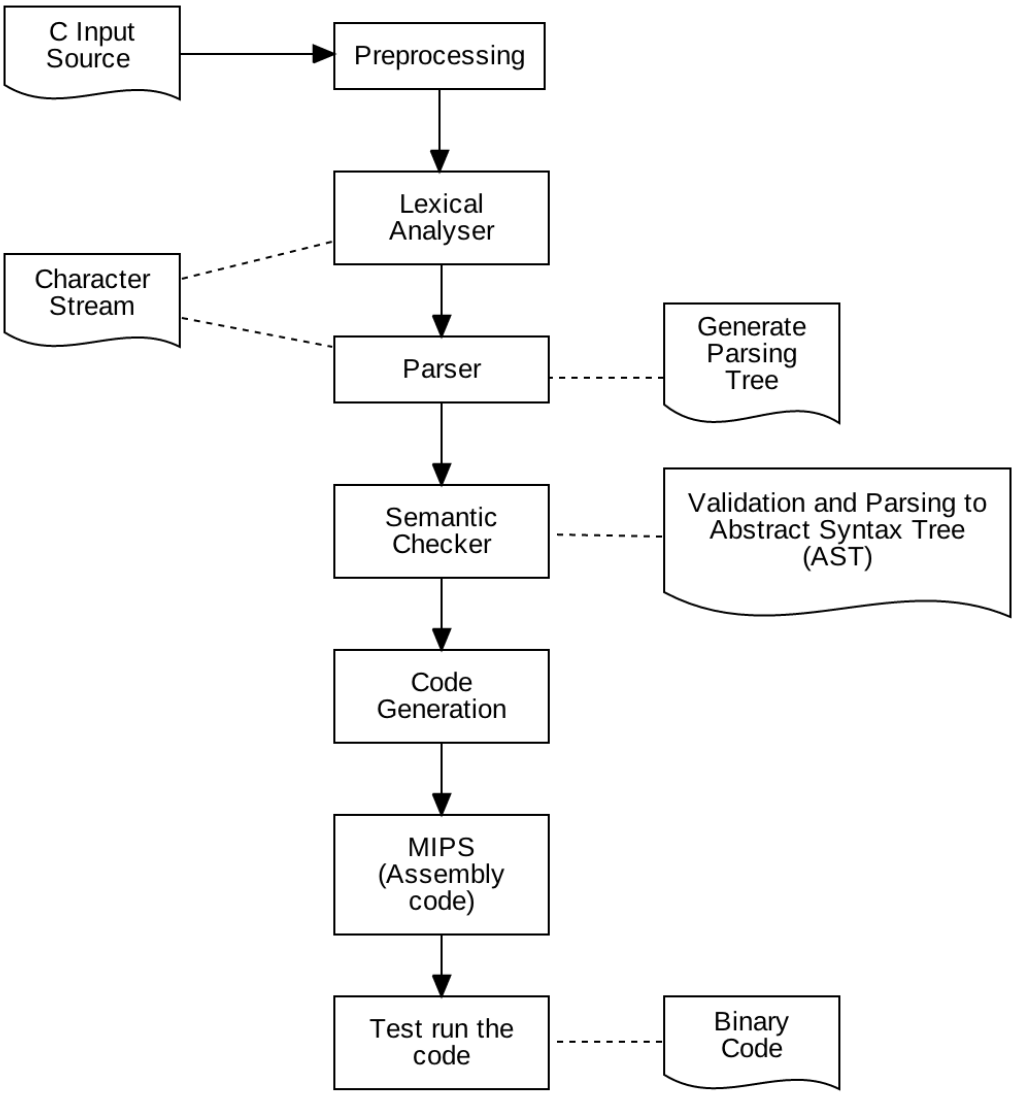
In this course we learn about the fundamentals of Compiler, how we plan, design and implementing how Compiler works and how to make it. And for the project we're should to do the Mini C Compiler using Java Programming Language to compile C Source Program as the input and doing the all process then will be generated the MIPS (Assembly Code) as the output for simulation that compiler is success and running correctly. Because sometimes we understand the theory, but we don't have an idea how to make the real compiler and how it exactly works and it's happen to me, So in this report we will explain the step how to make the Mini C Compiler.

Requirements

To do this project there's some of preparation you need:

1. Java JDK & JRE >= 7
   http://www.oracle.com/technetwork/java/javase/downloads/
2. Java IDE (Eclipse or Netbeans)
   http://netbeans.org/
3. Text Editor (Additional)
   https://www.sublimetext.com/
4. MARS (MIPS IDE)
   http://courses.missouristate.edu/KenVollmar/MARS/
5. Patience, Hard work and Keep learning!

How it works



Arrival Dwi Sentosa
2018

1. Preprocessing

Preprocessing state, in this state the compiler will get the input based on C Source program and should to process some of functions that is:

- Remove all the comments inside the code
- Remove unwanted white space

Example:

Before Preprocessing:

```
line 1:
line 2: /* this is our test file*/
line 3:
line 4: int main(){
line 5:    int i;
line 6:
line 7:    return d+4+2+1;
line 8: }
line 9:
line 10: int foo(){
line 11:         foo1();
line 12:         return 0;
line 13: }
line 14:
line 15: int foo1(){
line 16:         return 1;
line 17: }
```

After Preprocessing:

```
int main(){ int i; return d+4+2+1;} int foo() { foo1(); return 0; } int
    foo1() { return 1; }
```

2. Lexical Analyser

Lexical Analyser state, in this state compiler will doing attribute character stream to recognize the character based on lexical rules (set unique token for each character that meet the lexical rules).

Example:
Before

```
int main(){
        int i;

        return d+4+2+1;
}
```

After

```
TNT_CMPL_UNIT:int:0
    TNT_FUNC_LIST:int:0
        TNT_FUNC_DEF:int:0
            TNT_TYPE_SPEC:int:0
            TNT_ID:main:0
            TNT_SP_KB_OPENING:int:0
            TNT_ARG_LIST:int:0
            TNT_SP_KB_CLOSING:int:0
            TNT_CODE_BLOCK:int:0
                TNT_STMT:int:0
                    TNT_STMT_DECL:int:0
                        TNT_TYPE_SPEC:int:0
                        TNT_ID:i:0
                    TNT_STMT:int:0
                        TNT_STMT_RTN:int:0
                            TNT_EXPR:int:0
                                TNT_FACTOR:int:0
                                    TNT_ID:d:0
                                TNT_EXPR2:int:0
                                    TNT_OP_PLUS:int:0
                                    TNT_FACTOR:int:0
                                        TNT_CNST_INT:4:0
                                    TNT_EXPR2:int:0
                                        TNT_OP_PLUS:int:0
                                        TNT_FACTOR:int:0
                                            TNT_CNST_INT:2:0
                                        TNT_EXPR2:int:0
                                            TNT_OP_PLUS:int:0
                                            TNT_FACTOR:int:0
                                                TNT_CNST_INT:1:0
```

Token set based on lexical rules at picture above is represent by (TNT).

3. Parser

Parser state, in this state compiler will get input of attribute character stream (token) from lexical analysis state and generate an output as parsing tree.

Result:

```
TNT_CMPL_UNIT:int:0
    TNT_FUNC_LIST:int:0
        TNT_FUNC_DEF:int:0
            TNT_TYPE_SPEC:int:0
            TNT_ID:main:0
            TNT_SP_KB_OPENING:int:0
            TNT_ARG_LIST:int:0
            TNT_SP_KB_CLOSING:int:0
            TNT_CODE_BLOCK:int:0
                TNT_STMT:int:0
                    TNT_STMT_DECL:int:0
                        TNT_TYPE_SPEC:int:0
                        TNT_ID:i:0
                    TNT_STMT:int:0
                        TNT_STMT_RTN:int:0
                            TNT_EXPR:int:0
                                TNT_FACTOR:int:0
                                    TNT_ID:d:0
                                TNT_EXPR2:int:0
                                    TNT_OP_PLUS:int:0
                                    TNT_FACTOR:int:0
                                        TNT_CNST_INT:4:0
                                    TNT_EXPR2:int:0
                                        TNT_OP_PLUS:int:0
                                        TNT_FACTOR:int:0
                                            TNT_CNST_INT:2:0
                                        TNT_EXPR2:int:0
                                            TNT_OP_PLUS:int:0
                                            TNT_FACTOR:int:0
                                                TNT_CNST_INT:1:0
```

4. Semantic Checker

Semantic Checker state, in this state will get input parsing tree from parser and convert it to Abstract Syntax Tree (AST):

Result:

```
semantic checking...
-------------parsing to ast-----------------------
TNT_CMPL_UNIT:int:0
    TNT_FUNC_DEF:int:0
        TNT_TYPE_SPEC:int:0
        TNT_ID:main:0
        TNT_SP_KB_OPENING:int:0
        TNT_ARG_LIST:int:0
        TNT_SP_KB_CLOSING:int:0
        TNT_CODE_BLOCK:int:0
            TNT_STMT_DECL:int:0
                TNT_TYPE_SPEC:int:0
                TNT_ID:i:0
            TNT_STMT_RTN:int:0
                TNT_EXPR:int:0
                    TNT_OP_PLUS:int:0
                        TNT_OP_PLUS:int:0
                            TNT_OP_PLUS:int:0
                                TNT_FACTOR:int:0
                                    TNT_ID:d:0
                                TNT_FACTOR:int:0
                                    TNT_CNST_INT:4:0
                            TNT_FACTOR:int:0
                                TNT_CNST_INT:2:0
                        TNT_FACTOR:int:0
                            TNT_CNST_INT:1:0
```

5. Code Generation

Code Generation state, in this state will generate the assembly code based on output of all of process step above. (Result at [6]).

6. MIPS (Assembly Code)

Result of Code generation:

```
Code generating...
# This file is generated by BIT-MiniCC
#
#
        .data


        .text
#code for initialization
init:
        addiu $fp, $zero, 1024
        addiu $sp, $zero, 1024
        addiu $sp, $sp, -64
        jal main

        li $v0, 10
        syscall
#code for function main
main:
        addiu $sp, $sp, -64
        addiu $fp, $fp, -64

        addiu $sp, $sp, 64
        addiu $fp, $fp, 64

#code for function foo
oo:
       addiu $sp, $sp, -64
       addiu $fp, $fp, -64

       sw $ra, 0($fp)
       jal foo1
       lw $ra, 0($fp)


       addiu $sp, $sp, 64
       addiu $fp, $fp, 64
       jr $ra



#code for function foo1
oo1:
       addiu $sp, $sp, -64
       addiu $fp, $fp, -64

       addiu $sp, $sp, 64
       addiu $fp, $fp, 64
       jr $ra
```

7. Test run the code

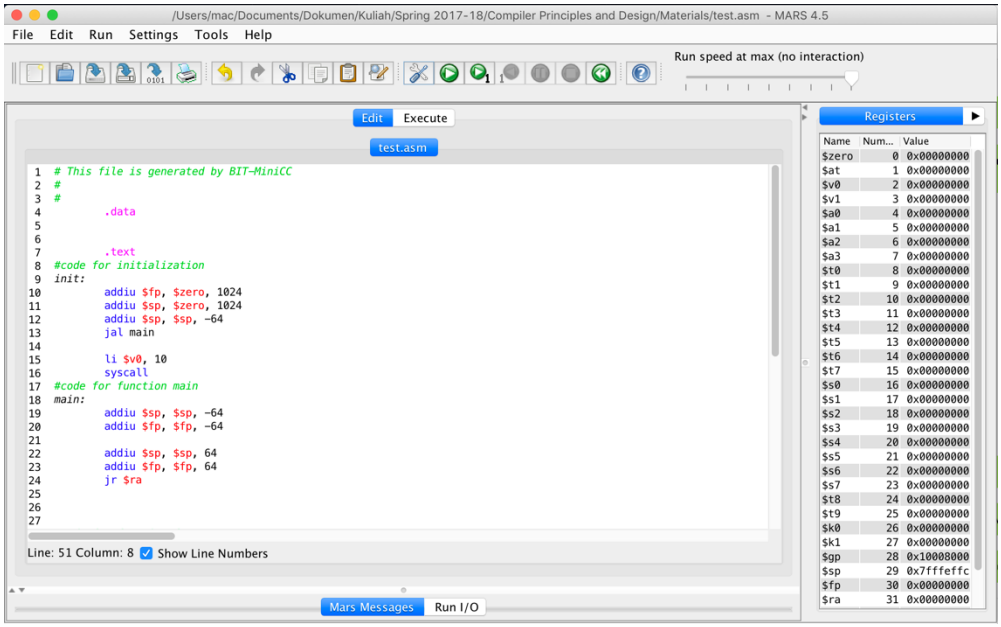


Figure 1. Copy Generated code to MARS

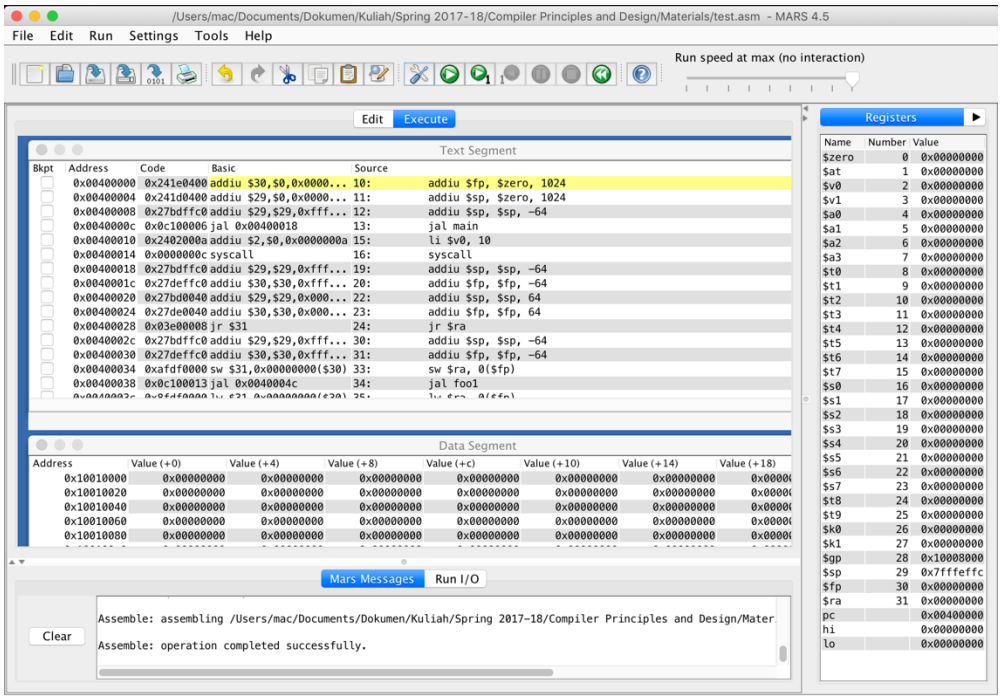


Figure 2. Assemble and Run the code

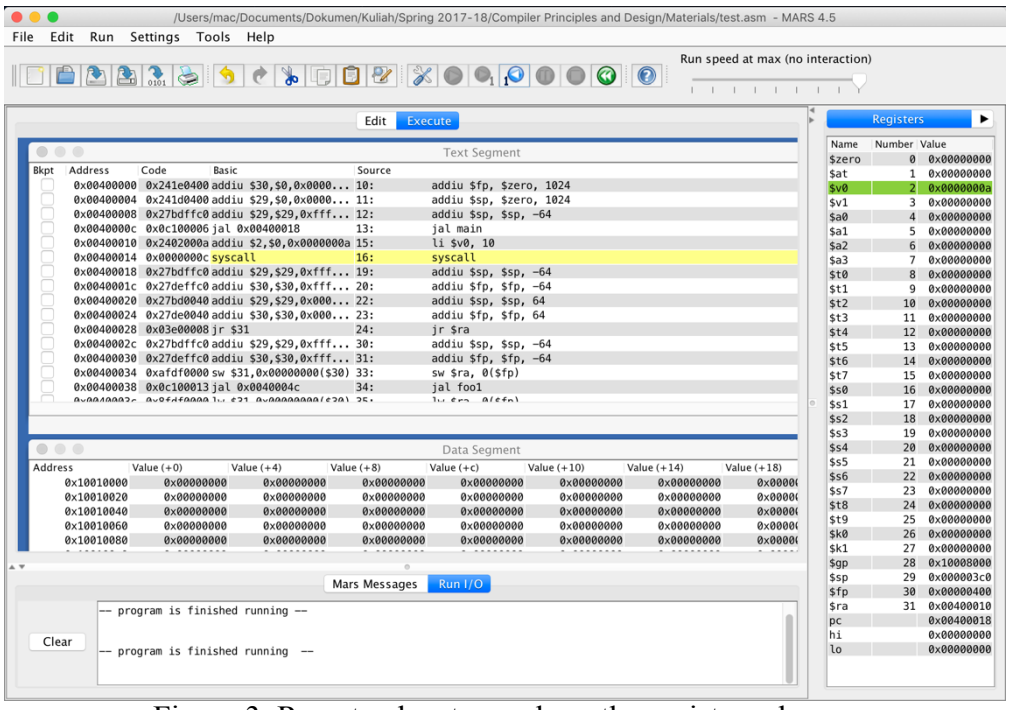


Figure 3. Run step by step and see the registers changes

Conclusion

This Mini C Compiler is not fully completed but the basic of understanding to build an simple C Compiler is done, from here there's important step to do there're: Preprocessing -> Lexical Analyser -> Parser (Generate Parsing Tree) -> Semantic Checker (Generate AST) -> Code Generation to Assembly -> Run testing

Project Source code is available in my github, please check it out:
https://github.com/arrivaldwis/BIT-MiniCC

Thanks

1. Ji Weixing 计卫星 as my Compiler Principles and Design Teacher that explain very calm and clearly that help me a lot to understanding the course and project.
2. Also all of my classmates