Simple Small C Compiler

Name: Wenhao Zhu

Student ID: 5130309717

Introduction

A simplified compiler frontend, including a **lexical analyzer** and a **syntax analyzer**, for `Small-C`, which is a C-like language containing a subset of the C programming language. Besides, it also implements a **code generator** to translate the intermediate representation, which is produced by syntax analyzer, into `LLVM` instructions.

Using **Flex**, **Bison** and **LLVM**

Structure

- **lex.l** Lexical Analyzer
- **parser.y** Syntax Analyzer
- **Node.h** Node declaration
- **Node.cpp** Node implement
- **syntax_tree.h** Tree structure and IR gen declaration
- **syntax_tree.cpp** Tree structure and IR gen implement
- **Makefile** `make` to easy compile the compiler, and `make clean` to delete useless files
- **quick-test.sh** shell script for easy test, test input is based on 7 given testcases, output is llvm IR code in `testcase-output-IR`

Usage

1.Quick test with testcase using scripts

```
$ chmod +x quick-test.sh
$ ./quick-test.sh
```

Detail of the shell script is:

```
$ make;
$ ./scc testcase-input/arth/arth.sc                testcase-output-
IR/arth.ll
$ ./scc testcase-input/fib/fib.sc                  testcase-output-
IR/fib.ll
$ ./scc testcase-input/gcd/gcd.sc                 testcase-output-
IR/gcd.ll
$ ./scc testcase-input/io/io.sc                    testcase-output-
IR/io.ll
$ ./scc testcase-input/if/if.sc                   testcase-output-
IR/if.ll
$ ./scc testcase-input/queen/queen.sc             testcase-output-
IR/queen.ll
$ ./scc testcase-input/struct/struct.sc           testcase-output-
IR/struct.ll
$ make clean;
```

All the testcase output will be saved at `testcase-output-IR` .
Then you're free to use llvm runtime to excute IR codes. For example:

```
$ lli arth.ll
```

ATTENSION!! As llvm IR syntax rules changes with version. My code generation is based on **llvm 3.5!!**. I've test that **llvm-3.7** or higher does not support the current IR code. Thus, you may need to run:

```
$ lli-3.5 arth.ll
```

2.Run test manually

```
$ make
```

1. Input from command line and output on command line

```
$ ./scc
```

> You are required to input your code. And the output of IR code will be will be print in command line and saved in file `NVM_RC_VERSION=`

2. Input from file and output on command line

```
$ ./scc   inputPath
```

> Output of IR code will be print in command line and saved in file `NVM_RC_VERSION=`

3. Input from file and output to file

```
$ ./scc   inputPath   outputPath
```

Highlights

1. Syntax Error Handling

If there is a syntax error, the compiler will return which line and what input causes the error.
For example:

```
Input:
int mian(
}

Output
Error: syntax error at line 2
Parser does not expect }
```

2. Semantic Error Handling

I have done some of the semantic error detection.

Operand type checking

As operation like `dot` or `[]` , we will first checking whether the operation is valid for the object. For example,:

```
Input:

int main()
{
    int x;
    x.get = 5;
    return 0;
}

---
Output:
....
Parsing Complete!
Error: Semantic error at line 4
Expected rules: EXPS: ID DOT ID
only struct can  be used as first parameter of Dot
Exit
```

Declaration checking

It maintain a symbol table the know whether the symbol has been declared. For example:

```
Input:

int main()
{
    return p;
}

---
Output:
....
Parsing Complete!
Error: Semantic error at line 3
Expected rules: EXPS: ID ARRS
not found symbol:p
Exit
```

Resevered word handling

This is implemented by the flex and yacc, as there are reserved words ,It will generate error during the building of the tree.

Break and continue checking

Break and continue can only be used in a for-loop. A stack is used to maintain if program is inside a for loop, and each time there meets the token will first check the stack. For example:

```
Input:

int main()
{
    break;
    return 0;
}

---
Output:
....
Parsing Complete!
Error: Semantic error at line 3
Expected rules: STMT: BREAK ;
break must be in for statement
Exit
```

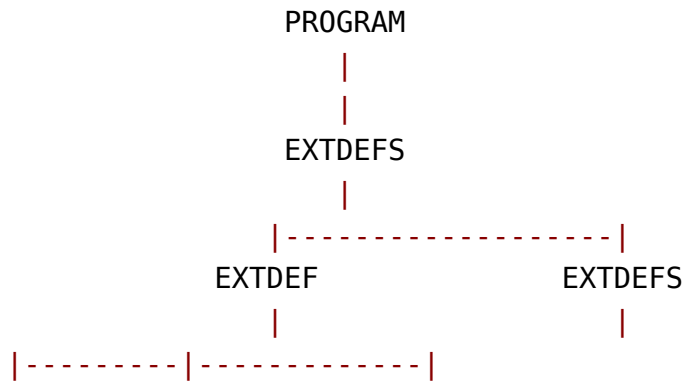
3. Code Optimization

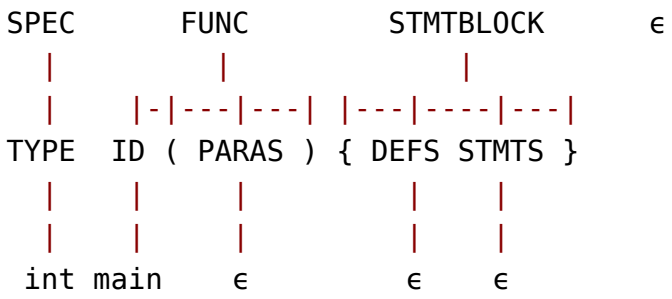
As there maintain a symbol table, for **unused function declaration**, the internal Tree Node will be directly removed for dead code elimination. As the same, **unused struct declaration** are also eliminated by removing the Node.

4. Tree Structure Printing

As mentioned in project 1, I don't use pre-order to print the parse tree. Instead, I use complicated functions to print a more distinct and clear tree structures for sake of beauty and more intuitive sense of the parse tree.

ATTENSION!! As I print the parse tree in lines for beauty, like this:





Thus, the output parse is **very wide** when the input code is complex.

You may need `MonoDevelop` under Ubuntu or some other text editors to open the output, otherwise some text editors like `sublime text3` will automatically create a new line for wide output which may affect the beauty. Or
you can just input fewer codes to see the output.

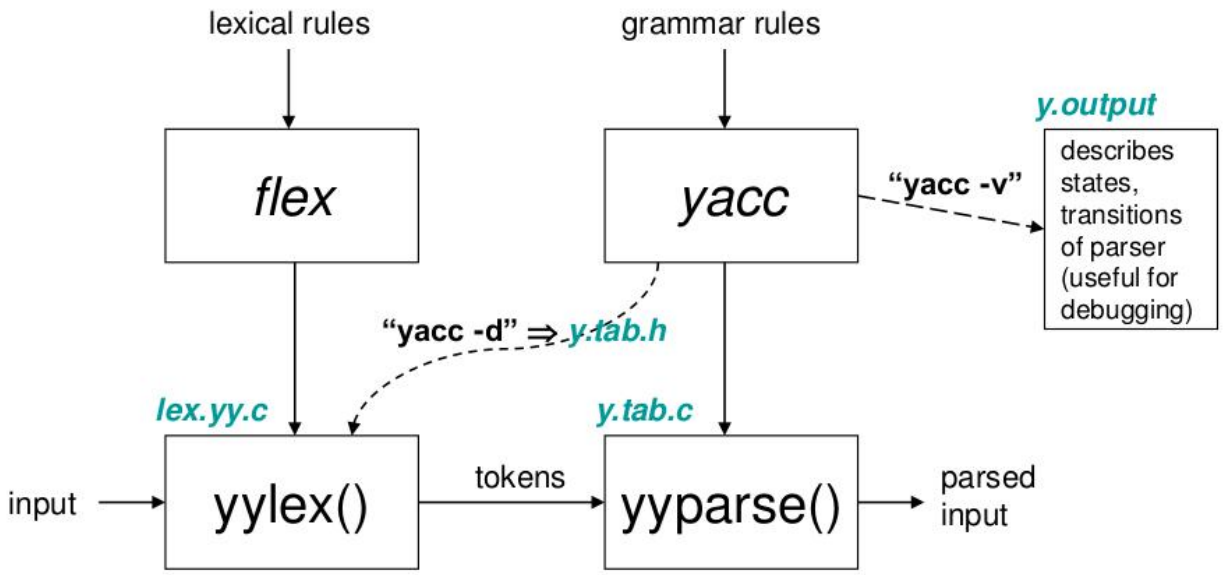
5. Timer

The compiler will get the total time for the procedure. You can analysis the efficiency with the clock time.

```
...
--------------------
InputFile -> testcase-input/queen/queen.sc
Parsing Complete!
Translation Complete!
Total time spent: 2.74ms

--------------------
InputFile -> testcase-input/struct/struct.sc
Parsing Complete!
Translation Complete!
Total time spent: 1.11ms
...
```

A Little More Details



Lexical Analyzer

A lexical analyser has been implemented in this part. It reads the source codes of **SMALLC** and separates them into tokens. The work is done using *FLEX* and the related file is `"lex.l"`

Read and Write

Since most of the details of **TOKENS** and **Operators** are given to us in the project requirement, I will not talk about them in the report. Instead, I will show you how to deal with **read** and **write** tokens. They are also very simple:

```
read            { yylval.string = strdup(yytext); return (READ);}
write           { yylval.string = strdup(yytext); return (WRITE);}
```

Syntax Analyzer

In this step, I performed the syntax analysis using *YACC* and the file name is `"parser.y"` .

Precedence of *IF* and *IF ELSE* Statement

There exists a conflict in the implementation of "*IF LP EXP RP STMT*" and "*IF LP EXP RP STMT ELSE STMT*", and the former one should have a lower precedence than the latter one. Here is the implementation:

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
%%
STMT:
 | IF LP EXP RP STMT %prec LOWER_THAN_ELSE
 | IF LP EXP RP STMT ELSE STMT
 ;
```

Error Message

The error message is done together with generating the parse tree. Once an error occurs during the procedure, the parse process will shutdown and report the just-found mistake:

```
int yyerror(const char *msg)
{
    fflush(stdout);
    fprintf(stderr, "Error: %s at line %d\n", msg,yylineno);
    fprintf(stderr, "Parser does not expect '%s\n'",yytext);
}
```

It will show you the line number of the error and its error text.

Semantic Analyzer & IR Generation

In this section, I have implemented the Semantic Analyzer & IR Generation together with the formation of the parse tree.

Tree Generation

The Parse Tree Generation is based on the construction of different kinds of Node and implements its `codegen` function. For example, some of the different variables of *Node* and there usage are shown in the table below:

Contained By	Variable Name	Usage
All	int mi_LineNum	Line Number, used by error message
	NodeType	Type of the node, e.g. Expression, Statement, Declaration

	Name	Different uses in different variables, e.g. function name
Expression	ExpressionType	The type of expression, e.g. binary_operator
	bool ValueExpression	Whether it is an L-Value Expression
…	…	…

Register Allocation

According some reference book, the algorithm is shown below, a little bit like **FIFO**:
1. Maintain a set `freeRgSet` for free registers .
2. whenever `regFree` is called, then the specific register will be add to `freeRgSet`
3. whenever `regAllocate` is called, then return the first element of `freeRgSet` . If there are no valid reg then return a newly created name.

Intermediate Representation

These are all related to llvm documents. And implements through code-gen functions in all inherits of Node class.

Small C	**LLVM IR**
read	%s = call i32 (i8*, …)* @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32* %s)
write	%s = call i32 (i8*, …)* @__isoc99_scanf(i8* getelementptr inbounds ([3 x i8]* @.str, i32 0, i32 0), i32* %s)
continue, break	br label %%%s
return	ret i32 %s
….	….

Other

All related information can be also found on my github.

Any problem happens to my code(can't run.. etc), plz contact me at weehowe.z@gmail.com

请用 **lli-3.5** 执行IR代码, 有问题的话, 可以联系我远程演示= =