CSC4180 Assignment 1: Micro Language Compiler

Release Date: January 25, 2024
Due Date: February 28, 2024

1 Introduction

In this assignment, you are required to **design and implement a compiler frontend for Micro language** which transforms the **Micro Program** into corresponding **LLVM Intermediate Representation (IR)** and finally translated to RISC-V assembly code and executable with the help of LLVM optimizer and its RISC-V backend. After that, we can execute the compiled program on our RISC-V docker container to verify the correctness of your compiler.
Since it is a senior major elective course, we don't want to set any limitation for you. You are strongly recommended to use **Lex/Flex** and **Yacc/Bison** taught in tutorial 3 to design your compiler frontend, but it is not forcible. You can choose **Any Programming Language** you like for this assignment, but the RISC-V container we use only has C/C++ toolchain installed and you need to provide me a **Dockerfile** to run your compiler and execute the RISC-V program, which may need some extra effort. Some languages also provide tools like Lex and Yacc, and you are free to use them. It is also OK if you want to design the scanner and parser by hand instead of using tools.

2 Micro Language

Before we move on to the compiler design, it is necessary to have an introduction to **Micro Language**, that serves as the input of our compiler. Actually, it is a very simple language, with limited number of tokens and production rules of context-free grammar (CFG):

- Only integers(i32); No float numbers
- No Declarations
- Variable consists of a-z, A-Z, 0-9, at most 32 characters long, must start with character and are initialized as 0
- Comments begin with "––" and end with end-of-line(EOL)
- Three kind of statements:
  - assignments, e.g. `a:=b+c`
  - read(list of IDs), e.g. `read(a, b)`
  - write(list of Expressions), e.g. `write(a, b, a+b)`
- `BEGIN`, `END`, `READ`, `WRITE` are reserved words
- Tokens may not extend to the following line

2.1 Tokens & Regular Expression

Micro Language has 14 Tokens, and the regular expression for each token is listed below. Since BEGIN is a reserved word in C/C++, we need to use the alternative BEGIN_ as the token class.

```
1. BEGIN_: begin
2. END: end
3. READ: read
4. WRITE: write
5. LPAREN: (
6. RPAREN: )
7. SEMICOLON: ;
8. COMMA: ,
9. ASSIGNOP: :=
10. PLUSOP: +
11. MINUSOP: −
12. ID: [a−zA−Z][a−zA−Z0−9_]{0,31}
13. INTLITERAL: −?[0−9]+
14. SCANEOF: <<EOF>>
```

2.2 Context Free Grammar

Here is the extended context-free grammar (CFG) of Micro Language:

```
1. <start> → <program> SCANEOF
2. <program> → BEGIN <statement list> END
3. <statement list> → <statement> {<statement>}
4. <statement> → ID ASSIGNOP <expression>;
5. <statement> → READ LPAREN <id list> RPAREN;
6. <statement> → WRITE LPAREN<expr list> RPAREN;
7. <id list> → ID {COMMA ID}
8. <expr list> → <expression> {COMMA <expression>}
9. <expression> → <primary> {<add op> <primary>}
10. <primary> → LPAREN <expression> RPAREN
11. <primary> → ID
12. <primary> → INTLITERAL
13. <add op> → PLUSOP
14. <add op> → MINUSOP
```

Note: {} means the content inside can appear 0, 1 or multiple times.

2.3 How to Run Micro Compiler

Here is a very simple Micro program that we are going to use as the sample program throughout this instruction. SCANEOF is the end of line and we do not need to explicitly write it in the program.

```
−− Expected Output: 30
begin
        A := 10;
        B := A + 20;
        write(B);
end
```

We can use our compiler to compile, optimize and execute this program to get expected output 30. **Note:** The exact command to run your compiler is up to you. Just specify out in your report how to compile and execute your compiler to get the expected output.

```
118010200@c2d52c9b1339:~/A1$ ./compiler ./testcases/test0.m
118010200@c2d52c9b1339:~/A1$ llc −march=riscv64 ./program.ll −o ./program.s
118010200@c2d52c9b1339:~/A1$ riscv64−unknown−linux−gnu−gcc ./program.s −o ./program
118010200@c2d52c9b1339:~/A1$ qemu−riscv64 −L /opt/riscv/sysroot ./program
30
```

3 Compiler Design

Figure 1 shows the overall structure of a compiler. In this assignment, your task is to implement the frontend only, which contains scanner, parser and intermediate code generator and form a whole → compiler with LLVM for Micro language.

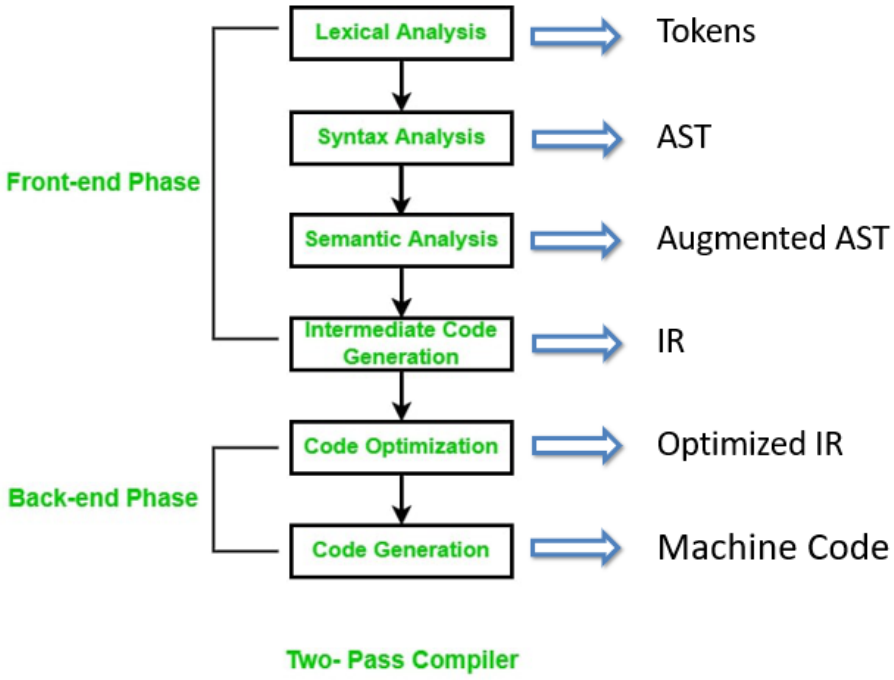

Figure 1: Compiler Structure

3.1 Scanner

Scanner takes input character stream and extracts out a series of tokens, and your scanner should be able to print out both token class and lexeme for each token. Figure **??** shows an example of the sample Micro program.

```
118010200@c2d52c9b1339:~/A1$ ./compiler ./testcases/test0.m ––scan–only
BEGIN_ begin
ID A
ASSIGNOP :=
INTLITERAL 10
SEMICOLON ;
ID B
ASSIGNOP :=
ID A
PLUOP +
INTLITERAL 20
SEMICOLON ;
WRITE write
LPAREN (
ID B
RPAREN )
SEMICOLON ;
END end
SCANEOF
```

3.2 Parser

Parser receives the tokens extracted from scanner, and generates a parse tree (or concrete syntax tree), and futhermore, the abstract syntax tree (AST) based on the CFG. Your compiler should be able to print out both the parse tree and the abstract syntax tree, and visualize them with graphviz.

3.2.1 Parse Tree (Concrete Syntax Tree)

Figure 2 shows an example of the concrete syntax tree generated from the sample program:

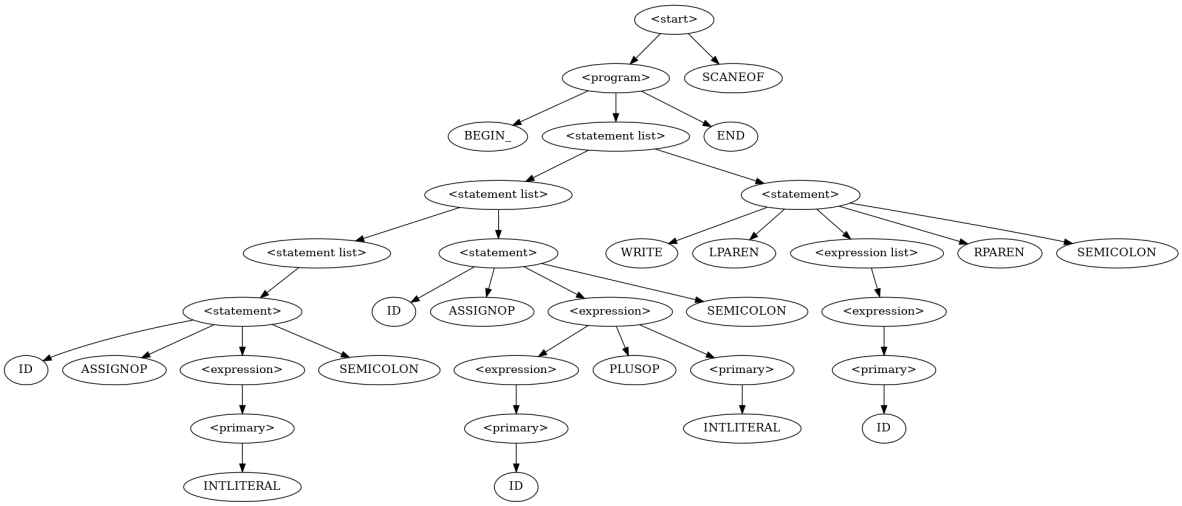

Figure 2: Concrete Syntax Tree of Sample Program

3.2.2 Abstract Syntax Tree

Figure 3 shows an example of the abstract syntax tree generated from the sample program:

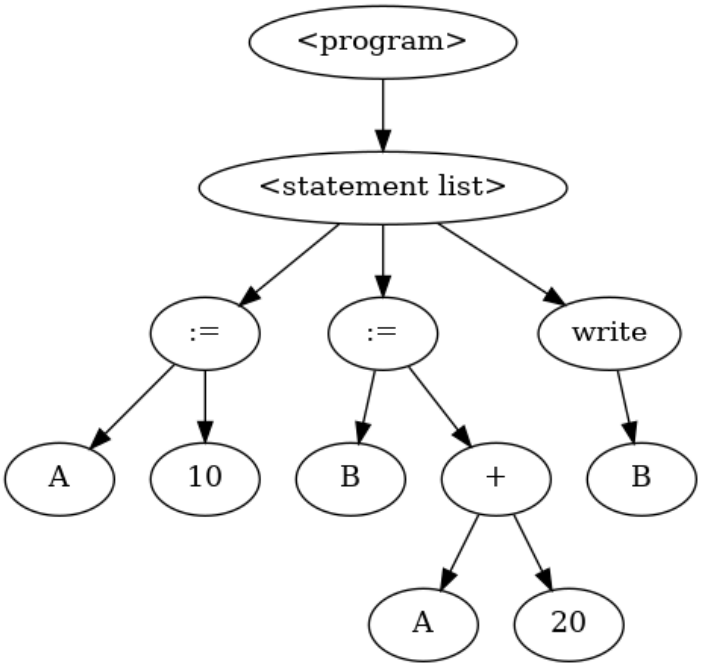

Figure 3: Abstract Syntax Tree of Sample Program

3.3 Intermediate Code Generator

For all the assignments in this course, the intermediate representation (IR) of the compiler should be the LLVM IR. There are mainly two reasons why LLVM IR is chosen. One is that LLVM IR can take advantage of the powerful LLVM optimizer and make up for the missing backend part of compiler in this course. The other is that LLVM IR can be easier translated into assembly code for any target machine, no matter MIPS, x86_64, ARM, or RISC-V. This functionality can make our compiler more compatible to machines with different architecture. The following shows the LLVM IR generated for the sample Micro program:

```
; Declare printf
declare i32 @printf(i8*, ...)

; Declare scanf
declare i32 @scanf(i8*, ...)

define i32 @main() {
        %_ptr0 = alloca i32
        store i32 10, i32* %_ptr0
        %A = load i32, i32* %_ptr0
        %_1 = add i32 %A, 20
        store i32 %_1, i32* %_ptr0
        %B = load i32, i32* %_ptr0
        %_scanf_format0 = alloca [4 x i8]
        store [4 x i8] c"%d\0A\00", [4 x i8]* %_scanf_format0
        %_scanf_str0 = getelementptr [4 x i8], [4 x i8]* %_scanf_format0, i32 0, i32 0
        call i32 (i8*, ...) @printf(i8* %_scanf_str0, i32 %B)
        ret i32 0
}
```

3.4 Bonus (Extra Credits 10%)

If you are interested and want to make your compiler better, you may try the following options:

- Add robust syntax error report
- Add a symbol table to make your compiler more complete
- Generate LLVM IR with LLVM C/C++ API instead of simply generating the string
- Optimize the LLVM IR generation plan for more efficient IR
- Any other you can think about...

4 Submission and Grading

4.1 Grading Scheme

- Scanner: 20%
- Parser: 40% (20% for parse tree generator and 20% for AST generation)
- Intermediate Code Generator: 30%

  We have prepared 10 test cases, and the points for each section will be graded according to the number of testcases you passed.
- Technical Report: 10%

  If your report properly covers the three required aspects and the format is clean, you will get 10 points.

- Bonus: 10%

  Refer to section 3.4 for more details. The grading of this part will be very flexible and highly depend on the TA's own judgement. Please specify clearly what you have done for the bonus part so that he do not miss anything.

4.2 Submission with Source Code

If you want to submit source C/C++ program that is executable in our RISC-V docker container, your submission should look like:

```
csc4180−a1−118010200.zip
|−
|−−− csc4180−a1−118010200−report.pdf
|−
|−−− testcases
|−
|−−− src
                |−
                |−−−Makefile
                |−−−ir_generator.cpp
                |−−−ir_generator.hpp
                |−−−node.cpp
                |−−−node.hpp
                |−−−scanner.l
                |−−−parser.y
                |−−−Other possible files
```

4.3 Submission with Docker

If you want to submit your program in a docker, your submission should look like:

```
csc4180−a1−118010200.zip
|−
|−−− csc4180−a1−118010200.Dockerfile
|−
|−−− csc4180−a1−118010200−report.pdf
|−
|−−− src
      |−
      |−−−Makefile
      |−
      |−−−run_compiler.sh
      |−
      |−−−Your Code Files
```

4.4 Technical Report

Please answer the following questions in your report:

- How to execute your compiler to get expected output?
- How do you design the Scanner?
- How do you design the Parser?
- How do you design the Intermediate Code Generator?
- Some other things you have done in this assignment?

The report doesn't need to be very long, but the format should be clean. As long as the three questions are clearly answered and the format is OK, your report will get full mark.