CSC4180 Assignment 1: Design a compiler for Micro

Release Date: February 8, 2023
Due Date: February 28, 2023

1 Introduction

In this assignment, you are required to **design a compiler for Micro Language** which transforms the **Micro code** into corresponding **MIPS assembly code**.

You are strongly recommended to use **Lex** and **Yacc** taught in the tutorial to design the compiler. However, it is not forcible. You still can choose **Any Programming Language** you like to finish the task. Some languages also provide tools like Lex and Yacc, and you are free to use them. If you want to design your own scanner and parser, it is welcome.

Since it is a senior elective course, we don't want to set any limitation for you. However, **you still should ensure that we can run and grade your program conveniently**. Therefore, we require you to submit your code with a **DockerFile** to provide the environment. **Any plagiarism will not be tolerated!**

2 Micro Language

Before design your compiler, it is necessary to learn **Micro Language**. Actually, it is a very simple language:

- Only integers(i32); No float numbers
- No Declarations
- Variable consists of A..Z, 0..9, at most 32 characters long, 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

Micro Language has 14 Tokens. The right side represents the matched string.

BEGIN: ‘‘begin”,	END: ‘‘end”,	READ: ‘‘read”,	WRITE: ‘‘read”,
LPAREN: ‘‘(”,	RPAREN: ‘‘)”,	SEMICOLON: ‘‘;”,	COMMA: ‘‘,”,
ASSIGNOP: ‘‘:=”,	PLUOP: ‘‘+”,	MINUSOP: ‘‘–”,	ID: variable ,
INTLITERAL: integer(i32)		SCANEOF: end of stream	

2.2 Context Free Grammar

Here is the extended CFG defining Micro Language:

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

Contents inside {} means that it can appear 0, 1 or multiple times.

2.3 Example

Here is a very simple Micro program: begin A := BB − 314 + A; end SCANEOF
In this program, BB and A's value are both initialized as 0. Therefore, A = -314 at end.

3 MIPS

We think all of you have learnt MIPS in CSC3050. If you haven't learnt or just forget, you should learn by yourself. Here is a simple reference. Don't be afraid, as the assignment just needs very simple MIPS knowledge and doesn't need you to be a MIPS expert.

3.1 How to run MIPS assembly code

For the convenience and consistence, we decide to use Mars as the MIPS simulator both for your debug and our evaluation. We don't require your generated MIPS assembly code is very professional(e.g. assembly code generated by mips-gcc). As long as your generated code can be executed in Mars and behave correctly, it's good.

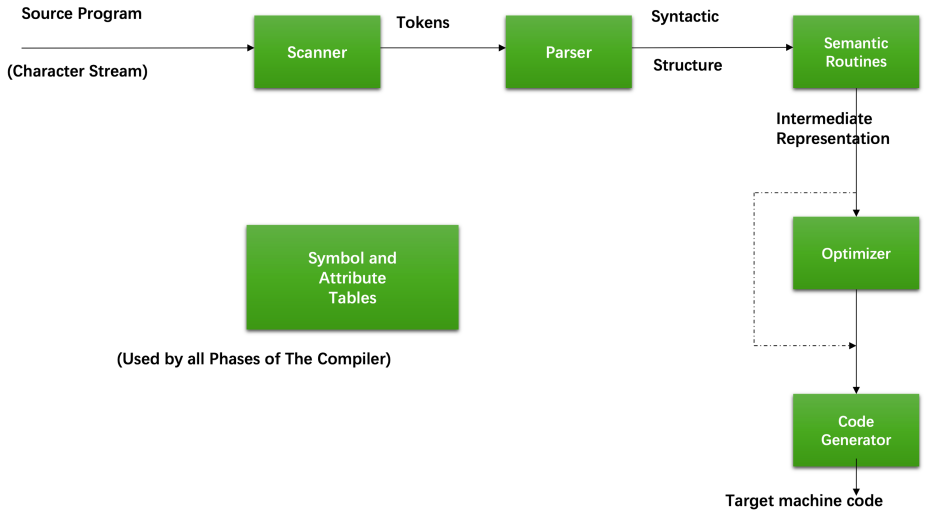

Figure 1: Compiler Structure

4 Program Design

Figure 1 shows the structure of a compiler. In this assignment, your task is to write a program which contains Scanner, Parser and Code Generator to form a Micro → MIPS compiler. You are free to use any tools like Lex and Yacc for help.

Your program may take a Micro Code File as the input , and finally print the generated MIPS assembly code to the std output. For example:

```
(base) -> Assignment 1 ./compiler ./TestCases/test0.m
addi $t0, $s0, -314
add $t0, $t0, $s1
add $s1, $t0, $zero
```

4.1 Scanner

Scanner receives input Character Stream and translates it into a series of Tokens. Figure 2 shows an example.

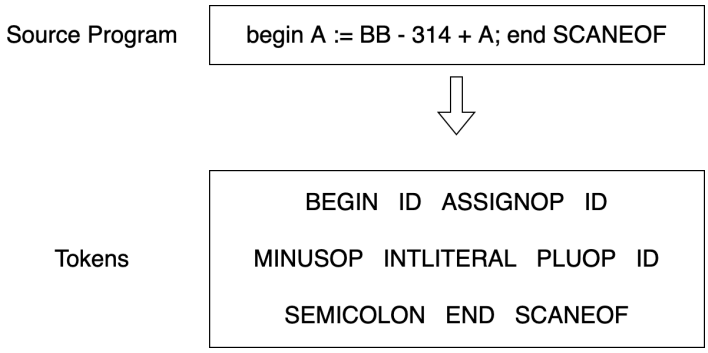

Figure 2: Scanner Example

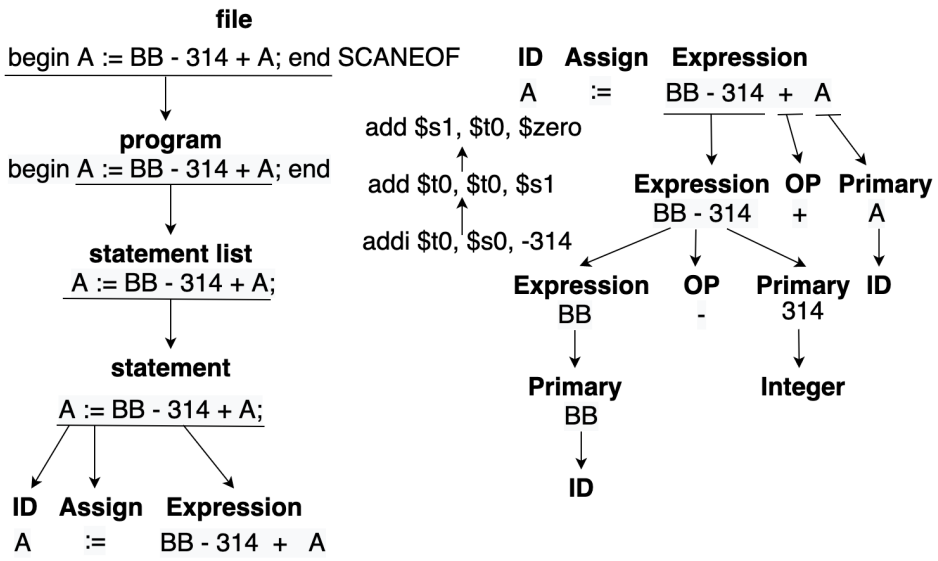

Figure 3: Syntax Tree Example

4.2 Parser and Code Generator

Parser receives the Tokens from Scanner, and generates a Syntax Tree based on the given CFG. At the nodes of the Syntax Tree, it will call functions provided by the Code Generator to generate appropriate code. Figure 3 shows an example.

In this course, Optimizer is not required, but we encourage you to generate more efficient code.

5 Submission and Evaluation

5.1 Submission with DockerFile

Since we don't set any limitations on the programming languages and tools, you should submit your code with a DockerFile as the execution environment for our grading.

You should package your DockerFile, technical report(pdf) and SourceCode Directory in a zip file for the submission as following:

```
csc4180-a1-117010349.zip
|-
|--- csc4180-a1-117010349.Dockerfile
|-
|--- csc4180-a1-117010349-report.pdf
|-
|--- SourceCode
        |-
        |---Makefile
        |-
        |---run_compiler.sh
        |-
        |---Your Code Files
```

Please follow the above file naming rules: DockerFile, zip, and pdf should be named to **csc4180-a1-YourID**.

SourceCode is the directory to place your source code. In addition to your source code files, it should also contains a Makefile and a shell script named run_compiler.sh.

Makefile is used to generate the executable program and is optional. If you choose to use a script language like Python, it is not required. Otherwise, your Makefile should support make all and make clean command. We will directly use the two commands to test your program.

run_compiler.sh is used to run your executable program. It should take the path to the Micro Code File as the first argument, and print the generated MIPS assembly code to the std output.

We will strictly follow the commands below to test your program(please replace 117010349 with your ID).

```
(base) -> docker build -f csc4180-a1-117010349.Dockerfile -t csc4180-a1-117010349 .
(base) -> docker run -it --mount type=bind,source=SourceCode,target=/opt/compiler_src/
          csc4180-a1-117010349 bash
-> cd /opt/compiler_src
-> make clean
-> make all
-> ./run_compiler.sh test0.m
addi $t0, $s0, -314
add $t0, $t0, $s1
add $s1, $t0, $zero
```

- Firstly, we will build with your DockerFile to get the execution environment.
- Then we will run the image with your source code mount to the container.
- Finally, we will enter the container to test your program.

Therefore, you should check by yourself carefully to ensure that your program and DockerFile can be compiled and executed correctly before the submission.

At last, we will use Mars to check the correctness of the generated MIPS assembly code.

5.2 Technical Report

We only care about three questions from your report:

- How do you design the Scanner?
- How do you design the Parser?
- How the code is generated?

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.

5.3 Grading Criterion

- Program Correctness: 80%. We have prepared 10 test cases, and 5 of them will be provided to you at the very beginning. If your program can pass the 5 given cases, you will get 60 points. If your program can pass rest of the cases, you will get the remaining 20 points.
- Code Style and Comment: 10%. If your code is clean and has necessary comments, you will get 10 points.
- Technical Report: 10%. If your report clearly answers the three questions and the format is clean, you will get 10 points.