香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Simplified C Compiler

Project IV

CSC4180: Compiler Construction

Name: *Derong Jin*

Student ID: *120090562*

Date: July 18, 2024

Contents

1 Overview

This project is about to design and implement a compiler for translating simplified C language to corresponding `MIPS` assembly. According to the context-free-grammar of the simplified C language (which can be found in Appendix. B), this project designs and implements a scanner and a parser in order to convert the input program into the abstract syntax tree (AST) representing the grammar included in the source program. This converting process consists of first tokenizing the program into a token stream by the scanner and then conducting a LR(1) parsing process on the token stream to generate the corresponding AST. After the AST is constructed, the code generation process generates target `MIPS` code recursively on the AST.

2 Design

This section contains the design ideas of the three parts in the compiler: the scanner, the parser and the code generator.

2.1 C Scanner

The scanner is the first step in the compiling process for C programs. It takes source code as input and generates a stream of tokens. Each token consists of a token type and a lexeme. The grammar defines the language's token types, and each token type is defined by a lexical structure (a regular expression in this project). The lexeme is the matched characters of the program by the token's lexical structure. During the scanning process, the scanner maintains an input program stream and generates a token that represents several characters at the head of the input stream each time. To avoid ambiguity, the grammar defines the identifying order of these tokens. The scanner's goal is to find the first identified longest pattern as lexeme and push it with its token type as a token to the output token stream.

In practice, regular expressions (RE) are a perfect tool to identify lexical patterns in the program. The tokens of the simplified C language are defined using regular expressions. Specifically, there are five types of tokens: `keyword`, `punctuation`, `operator`, `literal` and `identifier`. The first three classes are defined as a fixed pattern. For example, the `keyword WHILE` is defined as `while`, the `punctuation COMMA` is defined as `,`, and the `operator SHL_OP` is defined as `<<`. The other two classes are defined with regular expressions that match a variety of strings. An `INT_LITERAL` is defined as `[0-9]+` and an `ID` is defined as `[a-zA-Z][a-zA-Z0-9_]*`.

To support the matching of regular expressions, this project first construct a non-determined finite automaton (NFA) that expresses the same matching semantics as the original RE from the original RE. And then a determined finite automaton (DFA) equivalent to the NFA will be constructed to boost the matching process. After each RE is transformed all the way to its equivalent DFA, the scanner will use the DFAs to perform the matching task and generates corresponding tokens.

To enable regular expression matching, the project initially constructs a non-deterministic finite automaton (NFA) that expressses the same matching semantics as the original RE. Next, a deterministic finite automaton (DFA) equivalent to the NFA is constructed to enhance the matching process. Each regular expression is finally transformed to its equivalent DFA, which the scanner then utilizes to perform the matching and token generation tasks.

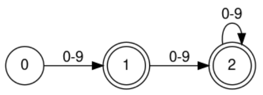


Figure 1: DFA for regex `[0-9]+`

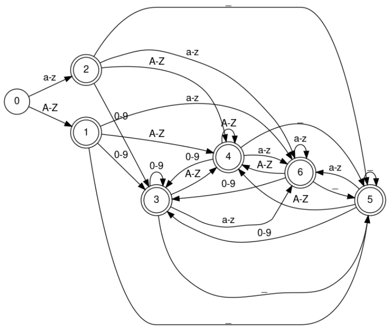


Figure 2: DFA for regex `[a-zA-Z][a-zA-Z0-9_]*`

2.2 C Parser

This subsection discusses the C parser design, which utilizes the canonical LR(1) parsing technique for constructing the LR parsing table from the grammar. In this technique, an LR(1) item is of the form $[A \to \alpha \cdot \beta, a]$, which has three components: a production rule from the grammar $A \to \alpha\beta$, a dot indicating the parsing process, and a lookahead terminal a. The initial item set is $[\texttt{goal} \to \cdot\texttt{program}, \$]$.

The state in LR parsing is the closure of a set of items. The initial state, I_0, is defined as the closure of the set containing $[\texttt{goal} \to \cdot\texttt{program}, \$]$. Additionally, the GOTO$(I, X)$ function calculates the next state of I after making an X move by collecting its possible successors. By conducting a BFS starting from I_0, we can construct the GOTO graph of the grammar.

Next, we construct the ACTION and GOTO parsing tables for the grammar from the GOTO graph. To do this, we enumerate every state and make their indices the row indices of the tables. For each state, say I_i, which is a set composed of LR(1) items, we check the following:

(a) If $[A \to \alpha \cdot a\beta, b]$ is in I_i, GOTO$(I_i, a) = I_j$, and a is a terminal, then we set ACTION$[i, a]$ to shift j".

(b) If $[A \to \alpha\cdot, a]$ is in I_i, and $A \neq \texttt{goal}$, then we set ACTION$[i, a]$ to reduce $A \to \alpha$."

(c) If $[\texttt{goal} \to \texttt{program}\cdot, \$]$ is in I_i, then we set ACTION$[i, \$]$ to "accept."

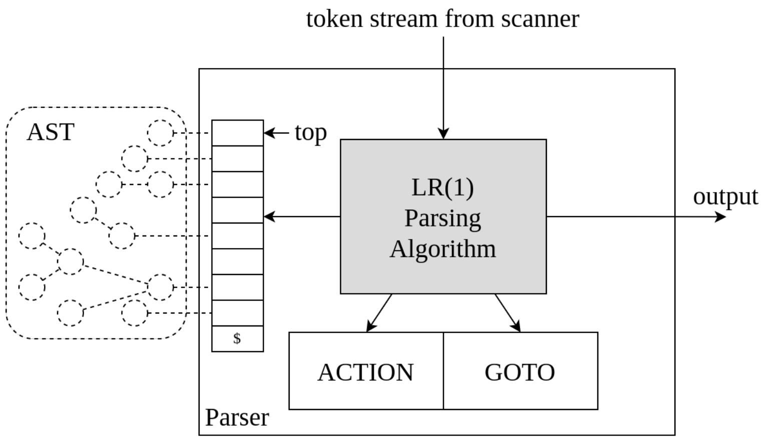


Figure 3: LR(1) parsing algorithm

We set GOTO $[i, A] = j$ if GOTO $(I_i, A) = I_j$, where A is a non-terminal. With these steps, we have constructed the canonical LR(1) parsing tables.

During parsing, two components need to be considered: the stack content and the remaining input. The former stores symbols that are not reduced completely into the AST, and the latter is a token stream from the scanner. Since we only have a single lookahead, the scanner takes one token at a time. The next move of the parser is determined by reading a, the current input terminal, and the state on the top of the stack. The parser then consults the entry ACTION $[s, a]$ in the parsing action table, and the state's movement follows the transition of the GOTO table.

The parser state transits through the following pattern until the accept state or one error state is reached (s is set to be the state on the top of the stack, and a is initially the first symbol from the scanner), the transition action only depends on the ACTION table:

1. ACTION $[s, a] =$ shift t: push t into the stack, update a to the next token.
2. ACTION $[s, a] =$ reduce $A \to \alpha$: pop size(α) symbols from the stack, push GOTO [stack.top, A] onto the stack, and output the production $A \to \alpha$

2.3 Code Generation

Once the C-compiler front-end generates the abstract syntax tree for the input program, the back-end analyzes it and generates corresponding machine code. Despite the absence of function calls in the language, the activation record (AR) structure is still used to store variables and temporary results. To generate the target MIPS assembly code in this project, a stack-machine approach is adopted, where the stack usage grows during computation and shrinks upon completion. To minimize changes to the stack pointer, each temporary's position is calculated at compile time. To achieve this, we define a code generation function, `cgen(e)`, that satisfies the following properties:

- generates target MIPS code for AST node representing production rule `e`
- preserves the stack structure before and after calling `cgen(e)`
- stores the desired result in register `$a0`

In the abstract syntax tree (AST), a production rule is always a node on the tree, the function `cgen` in this project is implemented recursively and by doing a traversal on the AST. The following shows some examples of the `cgen` implementation:

1. Fetching a variable (`x_off` is the variable with offset `off`, this information is stored in the sybol table):

```
cgen(x_off, NT):
        lw      $a0, off($fp)
```

2. Loading a constant:

```
cgen(c, NT):
        li      $a0, c
```

3. Binary operation (except those with short-circuit evaluation):

```
cgen(e1 op e2, NT):
        cgen(e1, NT)
        sw      $a0, -4 * NT ($fp)
        cgen(e2, NT + 1)
        lw      $t1, -4 * NT ($fp)
        cgen(op, NT)
```

4. Binary operator (always assume the left operand in `$t1`, and the right operand in `$a0`), take "+" for example, almost the same for the others:

```
cgen(+, NT):
        addu    $a0, $t1, $a0
```

5. Binary operation with short-circuit evaluation (`||` and `&&`), here take `&&` as example, the `cgen` for `||` is similar:

```
cgen(e1 && e2, NT):
        cgen(e1, NT)
        beqz    $a0, result_zero
        cgen(e2, NT)
        beqz    $a0, result_zero
        li      $a0, 1
        b       end_eval
    result_zero:
        move    $a0, $zero
    end_eval:
```

6. read/write statement: (these statements are similar, take `printf` as example)

```
cgen(printf \( exp \), NT):
        cgen(exp, NT)
        li      $v0, 1
        syscall
        li      $v0, 4
        la      $a0, break_line     # print new line '\n'
        syscall
```

7. Assign statement: (here taking array entry assignment as example):

```
cgen(x_off \[ exp_idx \] = exp, NT):
        cgen(exp_idx, NT)
        sll     $a0, $a0, 2
        addu    $a0, $fp, $a0
        sw      $a0, -4 * NT ($fp)      # save to temporary
        cgen(exp, NT + 1)
        lw      $t1, -4 * NT ($fp)
        sw      $a0, off ($t1)
```

8. do while statement:

```
cgen(do stmt while \( exp \), NT):
    do_while_begin:
        cgen(stmt, NT)
        cgen(exp, NT)
        bnez    $a0, do_while_begin
    do_while_end:
```

9. while loop statement:

```
cgen(while \( exp \) stmt, NT):
    while_begin:
        cgen(exp, NT)
        beqz    $a0, while_end
        cgen(stmt, NT)
        b       while_begin
    while_end:
```

10. open if statement:

```
cgen(if \(exp\) stmt, NT):
        cgen(exp, NT)
        beqz    $a0, end_if
        cgen(stmt, NT)
    end_if:
```

11. closed if statement:

```
cgen(if \(exp\) stmt1 else stmt2, NT):
        cgen(exp, NT)
        bnez    $a0, if_true
        cgen(stmt2, NT)
        b       end_if
    if_true:
        cgen(stmt1, NT)
    end_if:
```

With well-defined `cgen` function, the MIPS code is generate by calling `cgen(AST_root, 0)`.

3 Implementation

The C-compiler in this project is implemented in `C++` and consists of three main components: a scanner, a parser, and a code generator. The scanner tokenizes the input program, producing a stream of tokens that the parser uses to build an abstract syntax tree (AST). The code generator the traverses the AST, emitting `MIPS` assembly code.

3.1 Simple Regex Engine

The regular expression parsing system is a separate unit in the project and is utilized by the scanner to match the regular expression pattern with strings. This regex engine provides a series of APIs to construct REs, and match REs with input strings. To be specific, the engine denote a regular expression pattern as a class: `class Regex::RegexPatObj`, which provide 3 ways to declare a simple regular expression pattern and 3 ways to combine REs to generate more complex and powerful regular expression pattern.

- `RegexPatObj(const char * const pat);` generate a regex pattern that matches a fixed pattern `pat`

- `RegexPatObj(const std::set<char> & acc_char_set);` generate a regex pattern that accepts a set of single characters, e.g., `regex = '[abc]'`
- `RegexPatObj(char range_first, char range_last);` generate a regex pattern that accepts any character in a given range, e.g., `regex = '[a-z]'`
- `RegexPatObj operator+ (const RegexPatObj & rhs) const;` concatenate the two regexes
- `RegexPatObj operator| (const RegexPatObj & rhs) const;` regex alternation choosing from `rhs` and `*this`.
- `RegexPatObj RegexIter(const RegexPatObj & regex, int min_times = 0);` repetition, more than or equal to a given number of times.

After the regex expression pattern is construct, use the `max_matched_length` method of `RegexPatObj` to accquire the maximum accepted length, which is defined as:

`int Regex::RegexPatObj::max_matched_lenghth(const char *str, int length) const;`

For more details, please refer to `include/regex/regex.h`.

3.2 Scanner Class

The scanner uses regular expressions to match the input program against a set of defined tokens. This project implements a `Scanner`, which is initialized by a `std::string` representing the source code or by a `std::istream` containing the source code. After initialization, there are two methods to be called:

1. `Token next_token();` returns the next token from the source program. The returned token must have the longest matched lexeme at the start of remaining source code, if the lexeme can match multiple token type, then the method returns the first matched one. And returns `END` if thehed re is no more tokens.
2. `bool empty();` returns `true` if there is no more tokens.

For more details, please refer to `include/scanner.h`.

3.3 Parser Class

The parser constructs a canonical LR(1) parsing table and parse the program with LR(1) parsing algorithm. This project encapsulate the variabels and functions related to the parsing process into a class: `class Parser`

The parser class is initialized with a `unique_ptr` to a `Scanner` class. In the initialization process, the `Parser` load the grammar of the simplified C language and constructs LR(1) parsing table using the methods introduced in Design section. After the parser is initialized, calling method `std::shared_ptr<ASTNode>parse();` will perform LR(1) parsing on the program and returns the root node of the AST.

In the parsing process, two main helper classes are used:

1. `class Symbol` implements a unified class that can represent either a non-terminal or a terminal symbol and provide basic operations such as `operator==` to simplified the access to tokens and production rules in the parsing process.
2. `class ASTNode` implements a node on the AST, which has 4 attributes:
   - `Symbol symbol;` the terminal or non-terminal symbol represented by the node.
   - `int prod_idx;` the number of production rule that the node is reduced from.
   - `std::vector<std::shared_ptr<ASTNode>> children;` the children of the current node, typically the child nodes are the right-hand-side of the production rule.
   - `std::string lexem;` is the lexeme the current token is carrying.

   We will find `class ASTNode` provide good interface for the phase of code generation.

For more details, please refer to `include/parser.h`.

3.4 Code Generator Class

The code generator takes the AST generate by the parser as its input, and outputs corresponding `MIPS` code. This involves a `cgen` function working on each production rule (AST node) and generate the corresponding `MIPS` as expected. In this project, a class `class mipsCodeGen` is implemented to support the code generation process. To be specific, it consists two DFS passes on the AST. the first pass happens in the initialization part, in the first pass, the generator maintains the `symbol_table`, allocates stack memory for every variable, and records the variables' offsets in the activation record. In the second pass, it performs `cgen` function recursively and outputs the generated `MIPS` code.

The `cgen` function in Design section is implemented as a method of the `mipsCodeGen`:

```
void cgen_(std::shared_ptr<ASTNode> node_ptr, int nt, std::ostream &os);
```

the method recognize the production pattern of the current ASTNode and then generates the `MIPS` code using pre-defined template and completes the missing part recursively (as described in Design section).

Additionally, the symbol table is implemented with a `std::map<std::string, int>` that maps a lexeme to its offset in the activation record.

The operations above are encapsulated in the `class mipsCodeGen`, at construction time, it takes a `std::shared_ptr<ASTNode> node_ptr` to initialize, and the `MIPS` code can be generate to stream `os` by calling method `void mipsCodeGen::generate(std::ostream &os);`.

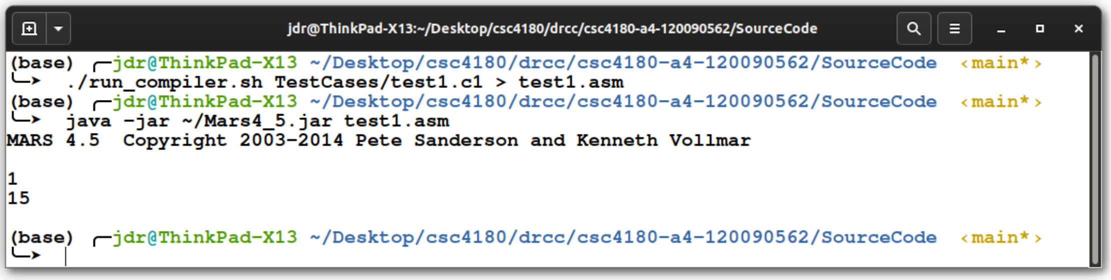
For more details, please refer to `include/code_gen.h`.

4 Tests & Evaluation

The compiler program is developed and tested under the following environment:

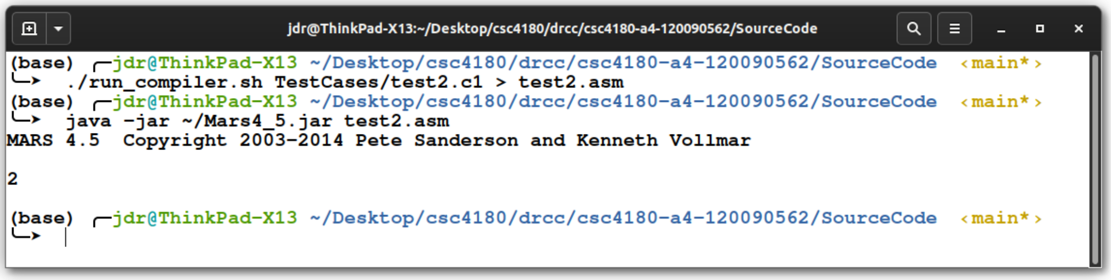
OS/Software	Version
OS	Ubuntu 20.04 focal
gcc	version 9.4.0 (Ubuntu 9.4.0-1ubuntu1 20.04.1)

The compiler passed all 5 provided test cases:



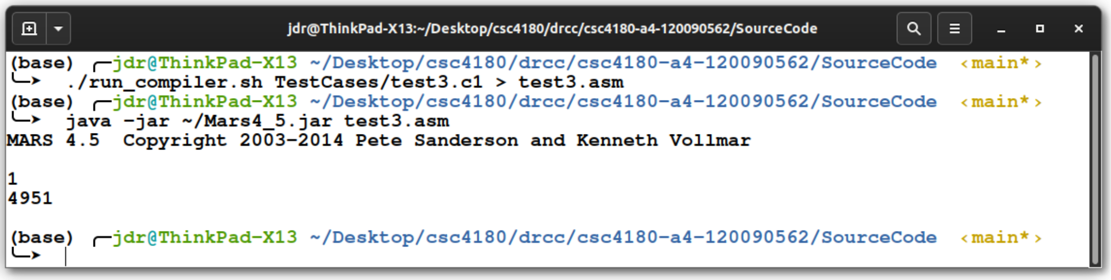
```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  ./run_compiler.sh TestCases/test1.c1 > test1.asm
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test1.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

1
15

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤
```

Figure 4: Testcase: `test1.c1`



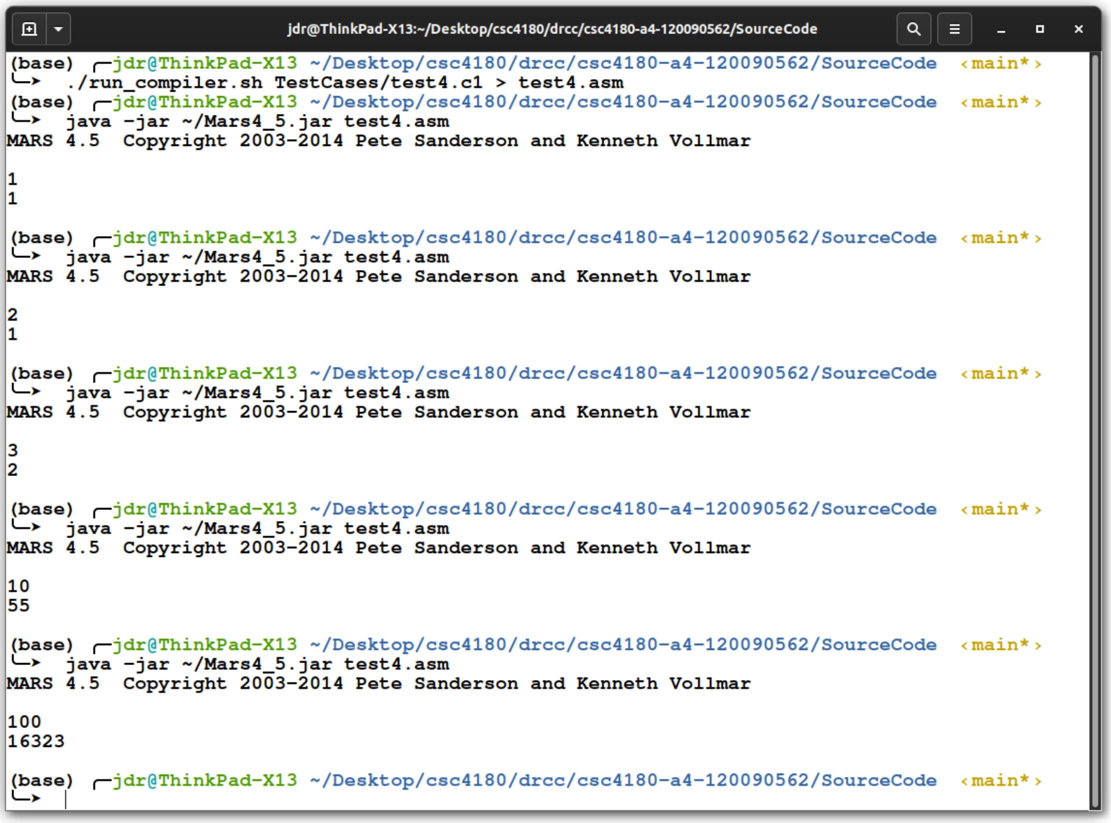
```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  ./run_compiler.sh TestCases/test2.c1 > test2.asm
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test2.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

2

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤
```

Figure 5: Testcase: `test2.c1`


jdr@ThinkPad-X13:~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode

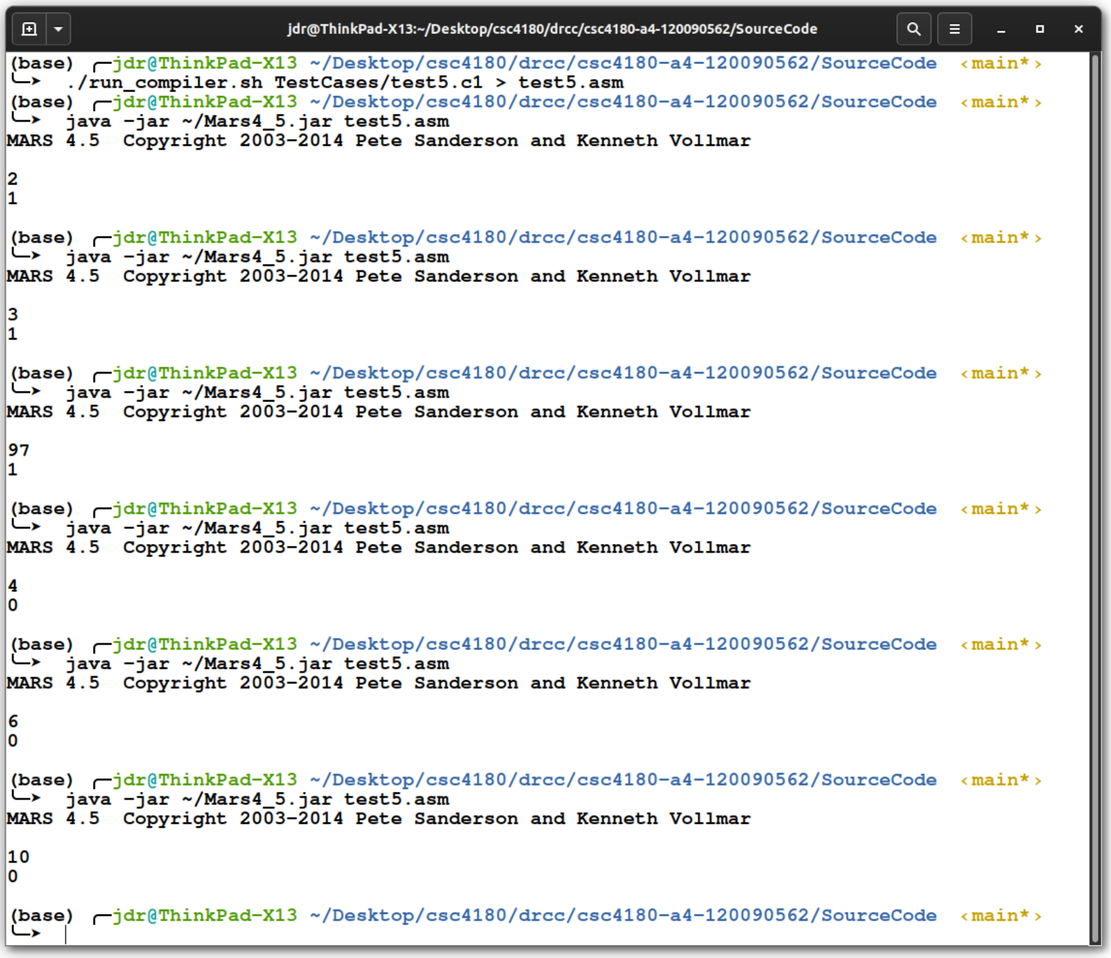
```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  ./run_compiler.sh TestCases/test3.c1 > test3.asm
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test3.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

1
4951

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤
```

Figure 6: Testcase: `test3.c1`


jdr@ThinkPad-X13:~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode

```
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  ./run_compiler.sh TestCases/test4.c1 > test4.asm
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test4.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

1
1

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test4.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

2
1

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test4.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

3
2

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test4.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

10
55

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test4.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

100
16323

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤
```

Figure 7: Testcase: `test4.c1`



```
jdr@ThinkPad-X13:~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  ./run_compiler.sh TestCases/test5.c1 > test5.asm
(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test5.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

2
1

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test5.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

3
1

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test5.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

97
1

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test5.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

4
0

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test5.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

6
0

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  java -jar ~/Mars4_5.jar test5.asm
MARS 4.5  Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

10
0

(base) ┌─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562/SourceCode  ‹main*›
└─➤  
```

Figure 8: Testcase: `test5.c1`

5 Conclusion

In conclusion, the development of the simplified C compiler written in C++ has been a challenging but rewarding experience. In this project, I have implemented a simplified compiler for the simplified C language, which consists of three main parts: the scanner, the parser, and the code generator. The scanner uses regular expressions (REs) to define and identify tokens and convert the input program string into a token stream. The parser utilizes the token stream and perform LR(1) parsing on it, and builds the abstract syntax tree (AST) of the original program. The code generator perform two times of depth first traversal on the computed AST and emits the target `MIPS` code. The compiler program successfully passed all the provided test cases. After I have done this project, I personally have gained a deep understanding of the complexities involved in implementing a compiler program. This project has been an excellent opportunity to apply the concepts and techniques learned in the lecture to a real-world problem.

Appendices

Appendix A Code Usage

The submission file `csc4180-a4-120090562.zip` has the following structure:

```
csc4180-a4-120090562.zip
|- csc4180-a4-120090562-report.pdf
|- csc4180-a4-120090562.Dockerfile
\-- SourceCode
    |- include
    |   |- code_gen.h
    |   |- miscs.h
    |   |- parser.h
    |   |- regex
    |   |   |- fsm.h
    |   |   \-- regex.h
    |   |- scanner.h
    |   |- symbols.h
    |   \-- tokenType.h
    |- Makefile
    |- run_compiler.sh
    \-- src
        |- c_grammar.cpp
        |- code_gen.cpp
        |- main.cpp
        |- miscs.cpp
        |- parser.cpp
        |- regex
        |   |- dfa.cpp
        |   |- nfa.cpp
        |   \-- regex.cpp
        |- scanner.cpp
        \-- symbols.cpp
```

A.1 How to compile & run the C compiler

This project provides a `Makefile` to automatise the compilation.

To generate the C compiler executable `drcc`:

```
$make all
```

To clean the workspace and delete **every** generated file (including `drcc` executable):

```
$make clean
```

After compilation, the following command is used to run the compiler:

```
$./drcc
```

The program read C source code from `stdin` and print the tokens to `stdout`. Hence if the source code is stored in a file, binding this file to `stdin` is all the thing needs to do:

```
$./drcc < <path-to-c-program>
```

More specifically, the executable `drcc` takes the C program code as input and reads it from `stdin`, and the compiled `MIPS` code to `stdout`, error messages are printed to `stderr`.

We also provide a bash file to run the compiler.

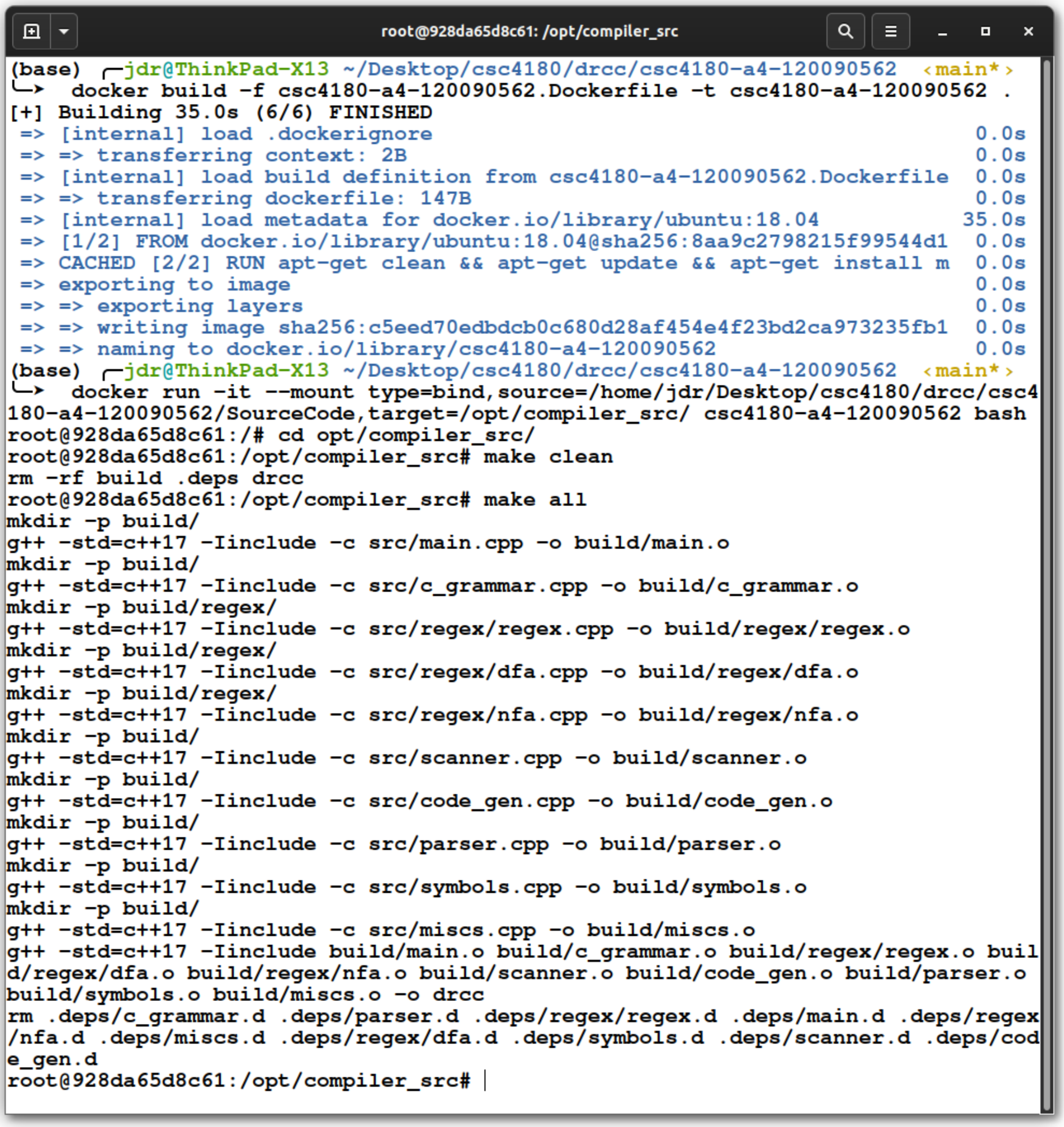
```
$bash run_compiler.sh <path-to-cfile>
```

Or

```
$./run_compiler.sh <path-to-cfile>
```

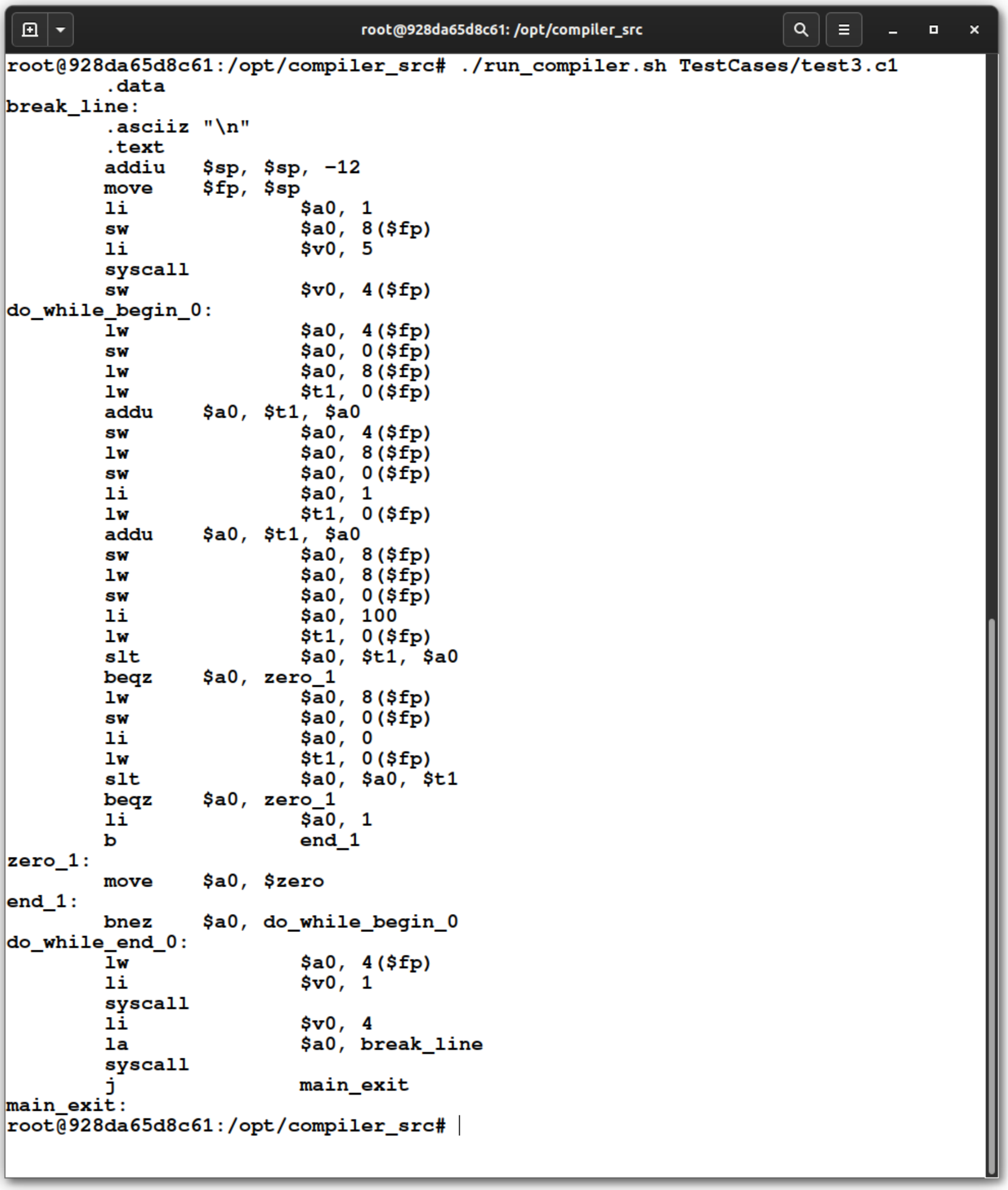
A.2 Demo

To make it portable to compile on other machines, a `Dockerfile` is provided in the submission `zip` file. A demo execution is as below:



```
(base) ╭─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562  ‹main*›
╰─➤  docker build -f csc4180-a4-120090562.Dockerfile -t csc4180-a4-120090562 .
[+] Building 35.0s (6/6) FINISHED
 => [internal] load .dockerignore                                          0.0s
 => => transferring context: 2B                                            0.0s
 => [internal] load build definition from csc4180-a4-120090562.Dockerfile  0.0s
 => => transferring dockerfile: 147B                                       0.0s
 => [internal] load metadata for docker.io/library/ubuntu:18.04           35.0s
 => [1/2] FROM docker.io/library/ubuntu:18.04@sha256:8aa9c2798215f99544d1  0.0s
 => CACHED [2/2] RUN apt-get clean && apt-get update && apt-get install m  0.0s
 => exporting to image                                                     0.0s
 => => exporting layers                                                    0.0s
 => => writing image sha256:c5eed70edbdcb0c680d28af454e4f23bd2ca973235fb1  0.0s
 => => naming to docker.io/library/csc4180-a4-120090562                    0.0s
(base) ╭─jdr@ThinkPad-X13 ~/Desktop/csc4180/drcc/csc4180-a4-120090562  ‹main*›
╰─➤  docker run -it --mount type=bind,source=/home/jdr/Desktop/csc4180/drcc/csc4
180-a4-120090562/SourceCode,target=/opt/compiler_src/ csc4180-a4-120090562 bash
root@928da65d8c61:/# cd opt/compiler_src/
root@928da65d8c61:/opt/compiler_src# make clean
rm -rf build .deps drcc
root@928da65d8c61:/opt/compiler_src# make all
mkdir -p build/
g++ -std=c++17 -Iinclude -c src/main.cpp -o build/main.o
mkdir -p build/
g++ -std=c++17 -Iinclude -c src/c_grammar.cpp -o build/c_grammar.o
mkdir -p build/regex/
g++ -std=c++17 -Iinclude -c src/regex/regex.cpp -o build/regex/regex.o
mkdir -p build/regex/
g++ -std=c++17 -Iinclude -c src/regex/dfa.cpp -o build/regex/dfa.o
mkdir -p build/regex/
g++ -std=c++17 -Iinclude -c src/regex/nfa.cpp -o build/regex/nfa.o
mkdir -p build/
g++ -std=c++17 -Iinclude -c src/scanner.cpp -o build/scanner.o
mkdir -p build/
g++ -std=c++17 -Iinclude -c src/code_gen.cpp -o build/code_gen.o
mkdir -p build/
g++ -std=c++17 -Iinclude -c src/parser.cpp -o build/parser.o
mkdir -p build/
g++ -std=c++17 -Iinclude -c src/symbols.cpp -o build/symbols.o
mkdir -p build/
g++ -std=c++17 -Iinclude -c src/miscs.cpp -o build/miscs.o
g++ -std=c++17 -Iinclude build/main.o build/c_grammar.o build/regex/regex.o buil
d/regex/dfa.o build/regex/nfa.o build/scanner.o build/code_gen.o build/parser.o
build/symbols.o build/miscs.o -o drcc
rm .deps/c_grammar.d .deps/parser.d .deps/regex/regex.d .deps/main.d .deps/regex
/nfa.d .deps/miscs.d .deps/regex/dfa.d .deps/symbols.d .deps/scanner.d .deps/cod
e_gen.d
root@928da65d8c61:/opt/compiler_src# |
```

Figure 9: A Demo Compilation



```
root@928da65d8c61:/opt/compiler_src# ./run_compiler.sh TestCases/test3.c1
        .data
break_line:
        .asciiz "\n"
        .text
        addiu   $sp, $sp, -12
        move    $fp, $sp
        li              $a0, 1
        sw              $a0, 8($fp)
        li              $v0, 5
        syscall
        sw              $v0, 4($fp)
do_while_begin_0:
        lw              $a0, 4($fp)
        sw              $a0, 0($fp)
        lw              $a0, 8($fp)
        lw              $t1, 0($fp)
        addu    $a0, $t1, $a0
        sw              $a0, 4($fp)
        lw              $a0, 8($fp)
        sw              $a0, 0($fp)
        li              $a0, 1
        lw              $t1, 0($fp)
        addu    $a0, $t1, $a0
        sw              $a0, 8($fp)
        lw              $a0, 8($fp)
        sw              $a0, 0($fp)
        li              $a0, 100
        lw              $t1, 0($fp)
        slt             $a0, $t1, $a0
        beqz    $a0, zero_1
        lw              $a0, 8($fp)
        sw              $a0, 0($fp)
        li              $a0, 0
        lw              $t1, 0($fp)
        slt             $a0, $a0, $t1
        beqz    $a0, zero_1
        li              $a0, 1
        b               end_1
zero_1:
        move    $a0, $zero
end_1:
        bnez    $a0, do_while_begin_0
do_while_end_0:
        lw              $a0, 4($fp)
        li              $v0, 1
        syscall
        li              $v0, 4
        la              $a0, break_line
        syscall
        j               main_exit
main_exit:
root@928da65d8c61:/opt/compiler_src# |
```

Figure 10: A Demo Execution

Appendix B The simplified C grammar

In the simplified C language, we are interested in the following C tokens (with their regular expressions as identifiers):

1. Keywords:

```
INT: int
MAIN: main
VOID: void
BREAK: break
DO: do
ELSE: else
IF: if
WHILE: while
RETURN: return
READ: scanf
WRITE: printf
```

2. punctuation & operators:

```
LBRACE: {
RBRACE: }
LSQUARE: [
RSQUARE: ]
LPAR: (
RPAR: )
SEMI: ;
PLUS: +
MINUS: -
MUL_OP: *
DIV_OP: /
MOD_OP: %
AND_OP: &
OR_OP: |
NOT_OP: !
ASSIGN: =
LT: <
GT: >
SHL_OP: <<
SHR_OP: >>
EQ: ==
NOTEQ: !=
LTEQ: <=
GTEQ: >=
ANDAND: &&
OROR: ||
COMMA: ,
```

3. Literals & variable identifiers:

```
INT_NUM: [0-9]+
ID: [a-zA-Z][a-zA-Z0-9_]*
```

The simplified C language has the following production rules:

```
1. goal -> program
2. program -> var_declarations statements
3. var_declarations -> var_declarations var_declaration
                     | /* empty */
4. var_declaration -> INT declaration_list SEMI
5. declaration_list -> declaration_list COMMA declaration
                     | declaration
6. declaration -> ID ASSIGN INT_NUM
                | ID LSQUARE INT_NUM RSQUARE
                | ID
7. code_block -> LBRACE statements RBRACE
8. statements -> statement
              | statements statement
9. statement -> open_stmt
             | closed_stmt
10. closed_stmt -> simple_stmt
                 | IF LPAR exp RPAR closed_stmt ELSE closed_stmt
                 | WHILE LPAR exp RPAR closed_stmt
11. open_stmt -> IF LPAR exp RPAR statement
              | IF LPAR exp RPAR closed_stmt ELSE open_stmt
              | WHILE LPAR exp RPAR open_stmt
12. simple_stmt -> assign_stmt SEMI
                 | ctrl_stmt
                 | io_stmt SEMI
                 | code_block
                 | exp SEMI
                 | SEMI
13. ctrl_stmt -> do_while_stmt SEMI
              | return_stmt SEMI
14. io_stmt -> read_stmt
            | write_stmt
```

```
15. assign_stmt -> ID LSQUARE exp RSQUARE ASSIGN exp
                 | ID ASSIGN exp
16. do_while_stmt -> DO statement WHILE LPAR exp RPAR
17. return_stmt -> RETURN
18. read_stmt -> READ LPAR ID RPAR
19. write_stmt -> WRITE LPAR exp RPAR
20. exp -> exp12
21. exp12 -> exp12 op12 exp11 | exp11
22. op12 -> OROR
23. exp11 -> exp11 op11 exp10 | exp10
24. op11 -> ANDAND
25. exp10 -> exp10 op10 exp8 | exp8
26. op10 -> OR_OP
27. exp8 -> exp8 op8 exp7 | exp7
28. op8 -> AND_OP
29. exp7 -> exp7 op7 exp6 | exp6
30. op7 -> EQ | NOTEQ
31. exp6 -> exp6 op6 exp5 | exp5
32. op6 -> GT | LT
33. op6 -> GTEQ | LTEQ
34. exp5 -> exp5 op5 exp4 | exp4
35. op5 -> SHL_OP | SHR_OP
36. exp4 -> exp4 op4 exp3 | exp3
37. op4 -> PLUS | MINUS
38. exp3 -> exp3 op3 exp2 | exp2
39. op3 -> MUL_OP | DIV_OP
40. exp2 -> op2 exp2 | exp1
41. op2 -> PLUS | MINUS | NOT_OP
43. exp1 -> INT_NUM | ID | ID LSQUARE exp RSQUARE | LPAR exp RPAR
```