CSC4180 Assignment 1 Report

Name: Guangxin Zhao

Student ID: 120090244

Date: Feb 22nd, 2024

Code Structure

```
project
|-
|--- README.md
|-
|--- testcases
|-
|--- src
        |-
        |--- ir_generator.cpp
        |--- ir_generator.hpp
        |--- main.cpp
        |--- Makefile
        |--- node.cpp
        |--- node.hpp
        |--- parser.y
        |--- scanner.l
```

How to execute the compiler?

Use commands below to run the compiler:

```
cd /path/to/project
make all

120090244@c2d52c9b1339:~/CSC4180-Compiler/Assignment1/src$ ./compiler --help
CUHK-SZ CSC4180 Assignment-1: Micro Language Compiler Frontend
Usage: Usage: compiler [options] source-program.m
Allowed options:
  -h [ --help ]                    Usage: compiler [options]
                                   source-program.m
  -s [ --scan-only ]               [Default: false] print out token class and
                                   lexeme pairs for each token, no parsing
                                   operations onwards
  -c [ --cst-only ]                [Default: false] generate concrete syntax
                                   tree only, do not generate AST and LLVM IR
```

```
  -d [ --dot ] arg (=ast.dot)        [Default: ast.dot] the .dot filename where
                                     compiler will output the tree
  -o [ --output ] arg (=program.ll)  [Default: program.ll] LLVM IR file compiled
                                     from source code
  --source-program arg               source Micro program to compile
```

Sample: `test0.m`

```
120090244@c2d52c9b1339:~/CSC4180-Compiler/Assignment1/src$ ./compiler ../testcases/test0.m
120090244@c2d52c9b1339:~/CSC4180-Compiler/Assignment1/src$ dot -Tpng ./ast.dot -o ./ast.png
120090244@c2d52c9b1339:~/CSC4180-Compiler/Assignment1/src$ opt ./program.ll -S --O3 -o ./program_optimized.ll
120090244@c2d52c9b1339:~/CSC4180-Compiler/Assignment1/src$ llc -march=riscv64 ./program_optimized.ll -o ./program.s
120090244@c2d52c9b1339:~/CSC4180-Compiler/Assignment1/src$ riscv64-unknown-linux-gnu-gcc ./program.s -o ./program
120090244@c2d52c9b1339:~/CSC4180-Compiler/Assignment1/src$ qemu-riscv64 -L /opt/riscv/sysroot ./program
30
```

How do I design the Scanner?

The scanner is designed to extract tokens of Micro language according to regular expressions. There are totally 14 tokens in Micro and the Regular Expression rules I designed are addressed below:

```
EOLN         "\n"
TAB          "\t"
COMMENT      --.*\n
BEGIN_       "begin"
END          "end"
READ         "read"
WRITE        "write"
LPAREN       "("
RPAREN       ")"
SEMICOLON    ";"
COMMA        ","
ASSIGNOP     ":="
PLUSOP       "+"
MINUSOP      "-"
ID           [a-zA-Z][a-zA-Z0-9_]{0,31}
INTLITERAL   -?[0-9]+
```

Extracting the tokens using Regular Expressions, we can get tokens stored with type `yylval.strval` or `yylval.intval`, and pass to the parser as `T_TOKENNAME`.

The scanner is also able to print both token class and lexeme (i.e. `<token-class, lexeme>`) for each token with `--scan-only` option. Here is a sample output for the print function on `test0.m`:

```
<BEGIN_, begin>
<ID, A>
<ASSIGNOP, :=>
<INTLITERAL, 10>
<SEMICOLON, ;>
<ID, B>
<ASSIGNOP, :=>
<ID, A>
<PLUSOP, +>
<INTLITERAL, 20>
<SEMICOLON, ;>
<WRITE, write>
<LPAREN, (>
<ID, B>
<RPAREN, )>
<SEMICOLON, ;>
<END, end>
<SCANEOF>
```

How do I design the Parser?

The Parser is designed to receive the tokens extracted from scanner. It also generates a parse tree (or concrete syntax tree), and futhermore, the abstract syntax tree (AST) based on the context-free grammar (CFG).

Here is the CFG of Micro language:

```
<start> → <program> SCANEOF
<program> → BEGIN <statement list> END
<statement list> → <statement> { <statement> }
<statement> → ID ASSIGNOP <expression>;
<statement> → READ LPAREN <id list> RPAREN;
<statement> → WRITE LPAREN<expr list> RPAREN;
<id list > → ID { COMMA ID }
<expr list > → <expression> { COMMA <expression> }
<expression> → <primary> { <add op> <primary> }
<primary> → LPAREN <expression> RPAREN
<primary> → ID
<primary> → INTLITERAL
<add op> → PLUSOP
<add op> → MINUSOP
```

In `parser.y`, I designed three `yylval` data types:

```
%union {
    /* Data */
    int intval;
    const char* strval;
    struct Node* nodeval;
};
```

Then, assign these data types to terminal symbols and non-terminal symbols.

Terminal Symbols:

```
%token <intval> T_INTLITERAL
%token <strval> T_BEGIN_ T_END T_READ T_WRITE T_LPAREN T_RPAREN T_SEMICOLON
%token <strval> T_COMMA T_ASSIGNOP T_PLUSOP T_MINUSOP T_SCANEOF T_ID
```

Non-Terminal Symbols:

```
%type <nodeval> program statement_list statement id_list expr_list expression primary
```

How do I design the Intermediate Code Generator?

The intermediate representation (IR) of the compiler is the LLVM IR, and the IR generator converts the AST given from the parser to LLVM IR (`.ll`) file. The `ir_generator.cpp` file has the following structure:

Export AST to LLVM IR

The `export_ast_to_llvm_ir()` function generates the LLVM IR header and main function structure. And it calls the `gen_llvm_ir()` function.

Generate LLVM IR

The generate LLVM IR sequence recursively calls the main routine or the sub-routines to generate the full LLVM IR.

- Function `gen_llvm_ir()`

This is the main routine in the generation part, it calls sub-routines such as `gen_assignop_llvm_ir()`, `gen_read_llvm_ir()`, `gen_write_llvm_ir()`, and also, the main routine `gen_llvm_ir()` to traverse the whole AST tree.

- Function `gen_assignop_llvm_ir()`

This sub-routine handles the following LLVM IR instructions:

```
%<variable> = alloca i32
store i32 <rvalue>, i32* %<variable>
```

- Function `gen_read_llvm_ir()`

This sub-routine handles the following LLVM IR instructions:

```
%<variable> = alloca i32
%_scanf_format_1 = alloca [# x i8]
store [# x i8] c"%d ... %d\00", [# x i8]* %_scanf_format_1
%_scanf_str_1 = getelementptr [# x i8], [# x i8]* %_scanf_format_1, i32 0, i32 0
call i32 (i8*, ...) @scanf(i8* %_sacnf_str_1, i32* %<variable>)
```

- Function `gen_write_llvm_ir()`

This sub-routine handles the following LLVM IR instructions:

```
%_printf_format_1 = alloca [# x i8]
store [# x i8] c"%d ... %d\0A\00", [# x i8]* %_printf_format_1
%_printf_str_1 = getelementptr [# x i8], [# x i8]* %_printf_format_1, i32 0, i32 0
call i32 (i8*, ...) @printf(i8* %_printf_str_1, i32 <rvalue>)
```

- Function `gen_operation_llvm_ir()`

This sub-routine handles the following LLVM IR instructions:

```
%_tmp_1 = load i32, i32* %<variable>
%_tmp_2 = sub i32 <expression>, %_tmp_1
%_tmp_3 = add i32 %_tmp_1, %_tmp_2
```

- Function `gen_expression_llvm_ir()`

This sub-routine handles the following LLVM IR instruction:

```
%_tmp_1 = load i32, i32* %<variable>
```