Project 4: Code Generation

Introduction to Compilers

TA: Donguk Ju (donguk.red@snu.ac.kr)

Released: November 24, 2025
Due: December 19, 2025

Contents

1 Introduction

1.1 Code generation

After analyzing the source code, the compiler emits either intermediate code or machine code. The intermediate code is a form of intermediate representation (IR) designed for analysis of computer programs. The compiler optimizes the intermediate code by applying some transformations, and then translates it into machine code for execution. Machine code is the lowest-level program that can be directly executed on the real machine. Some compilers generate machine code without explicitly generating intermediate code. However, most modern optimizing compilers use intermediate code as their IR for multiple purposes.

1.2 Implementing a code generator

In this project, we will implement an intermediate code generator using GNU Bison as a continuation of the previous project. The intermediate code is similar to the Java bytecode. We will execute generated codes on the stack machine simulator and observe its behavior.

2 Environment Setup

2.1 Pull the repository

Attach to the container and pull the repository to set up the environment for this project.

```
docker exec -it intcmp_2025 bash
cd intcmp-2025
git pull
```

3 The Stack Machine Simulator

The code we will generate is based on our custom ISA. It was designed to be executed on the stack machine simulator developed for this class. It is a kind of virtual machine, and can be used to simulate the run-time behavior of the generated code. Similar to the Java Virtual Machine (JVM), the simulator operates on the operands after popping them from the stack, and pushes the result back to the stack.

You may inspect the function `simulate_stack_machine()` located around line 357 of the `./simulator/gram.y` file to examine the exact behaviors of each instruction.

3.1 Usage

Go to the `./simulator` directory, and build the simulator.

```
cd ./simulator
make all
```

Execute an assembly code using the following command. The simulator will output the result of the program execution. If `[input_file]` is left blank, `stdin` will be used as the input stream.

```
./sim [input_file]
```

4 Instruction Set Architecture

This section describes the instruction set architecture (ISA) of the stack machine simulator. In this architecture, int, char, and pointer type objects have the size of a word, which is equal to 4-bytes.

4.1 Registers

There are only three registers in the stack machine simulator: the stack pointer (SP), the frame pointer (FP), and the program counter (PC).

Stack Pointer (`sp`)

- Points to the top of the stack.
- Mainly used for storing local variables.
- The stack grows upward, and each element of the stack contains a word.

Frame Pointer (`fp`)

- Points to the base of the current stack frame.
- Used for maintaining the frames for function calls and returns.

Program Counter (`pc`)

- Points to the address of the currently executing instruction.
- It is modified when handling the branch instructions.

4.2 Instructions

This section describes the available instructions in the stack machine simualtor. Please utilize some of these instructions to implement the code generator.

4.2.1 Global data area management

A global data area is allocated separately from the stack, and can only be accessed with a global data area pointer.

`<GP>` is the name of a global data area pointer.
`<SIZE>` is the size of a data area in words.
`<STRING_CONST>` is a string constant to be stored in the area.

Instruction	Meaning
`<GP>. data <SIZE>`	Specify the name and size of a global data area.
`<GP>. string <STRING_CONST>`	Specify the name and a stored string value of a global data area.

Examples

- `Lglob. data 3` : Allocate 12-bytes of uninitialized global data area, pointed by 'Lglob'.
- `Str0. string hello_world\n` : Store '`hello_world\n`' in the location 'Str0'.

4.2.2 Unary arithmetic and logic instructions

The unary arithmetic/logic instructions pop the top element of the stack, apply an operation to its value, and push the result back onto the stack.

Instruction	Meaning
`negate`	`-op`
`not`	`!op`
`abs`	`|op|`

4.2.3 Binary arithmetic and logic instructions

The binary arithmetic/logic instructions pop two elements from the stack, apply an operation with the top element as the right operand (op1) and the second element as the left operand (op2), and then push the result back onto the stack.

Instruction	Meaning
`add`	`op2 + op1`
`sub`	`op2 - op1`
`mul`	`op2 * op1`
`div`	`op2 / op1`
`mod`	`op2 % op1`
`and`	`op2 && op1`
`or`	`op2 || op1`
`equal`	`op2 == op1`
`not_equal`	`op2 != op1`
`geater`	`op2 > op1`
`greater_equal`	`op2 >= op1`
`less`	`op2 < op1`
`less_equal`	`op2 <= op1`

4.2.4 Branch and control instructions

The branch and control instructions are used to manage the control flow of programs.
Conditional branch instructions pop the top element of the stack and use its value as the branch condition.

Instruction	Meaning
`jump <label>`	Direct jump to the specified label.
`jump <+/-offset>`	Relative jump by the offset from the current instruction.
`jump <label><+/-offset>`	Relative jump by the offset from the label.
`branch_true <label>`	Jump to the specified label if the condition is true.
`branch_true <+/-offset>`	Jump by the offset from the current instruction if the condition is true.
`branch_true <label><+/-offset>`	Jump by the offset from the label if the condition is true.
`branch_false <label>`	Jump to the specified label if the condition is false.
`branch_false <+/-offset>`	Jump by the offset from the current instruction if the condition is false.
`branch_false <label><+/-offset>`	Jump by the offset from the label if the condition is false.
`exit`	Terminate the program.

4.2.5 Stack manipulation instructions

Stack manipulation instructions consist of push, pop and shift operations for manipulating the stack.

Instruction	Meaning
`push_const <int_const>`	Push an integer constant onto the stack.
`push_reg <register>`	Push the value from the register onto the stack.
`pop_reg <register>`	Pop the top element of the stack and store its value to the register.
`shift_sp <int_const>`	Shift the stack pointer by the speficied amount.

4.2.6 Memory access instructions

Assign instruction pops two elements from the stack and stores the value of the top element (op1) in the address specified by the second element (op2).

Fetch instruction pops the top element of the stack to get a memory address, and pushes the value stored at that address onto the stack.

Instruction	Meaning
`assign`	`*op2 = op1`
`fetch`	Push `*op1` onto the stack.

4.2.7 I/O instructions

I/O instructions are used to read a value from the standard input, or write a value to the standard output.

Instruction	Meaning
`read_int`	Read an integer from the `stdin` and push it onto the stack.
`read_char`	Read a chracter from the `stdin` and push it onto the stack.
`write_int`	Pop the top element of the stack, and write it to the `stdout` as an integer.
`write_char`	Pop the top element of the stack, and write it to the `stdout` as a character.
`write_string`	Pop the top element of the stack, and write it to the `stdout` as a string.

4.2.8 Example: Startup code

The following is an example of a startup code. It might be changed depending on the calling convention. It is assumed that the `main` function does not have parameters.

```
  shift_sp 1
  push_const EXIT
  push_reg fp
  push_reg sp
  pop_reg fp
  jump main
EXIT:
  exit
main:
```

5 Goal of the Project

You have to implement a code generator which can translate subC source code into the Stack Machine Simulator intermediate code, building on the previous project. The project score will be determined by the correctness of the output when the generated code is executed on the Stack Machine Simulator. Therefore, minor variations in the generated code are acceptable as long as the output of the code remains consistent.

You can assume that the input source code is syntactically and semantically correct, so error checking is not necessary for this project. Thus, even if you have not completed project #3, you can still work on this project. However, it would be challenging if you have not finished implementing the fundamental features of project #3, such as a scoped symbol table, or handling variable, struct, and function declarations. Please make sure that these features are fully implemented before starting.

Complete the code in `subc.l`, `subc.y`, and `hash.c`. You may add or modify any files in the `src` directory as needed, but please make sure that the project can be built using `make all` command.

Also, please submit a report of 4 pages or less containing: (1) an overview of the design of your code generator, (2) a list of features that were implemented and those that were not, and (3) some details of any advanced features that were implemented.

5.1 Assumptions

The following assumptions about the input code, which were established in the previous project except for two new conditions, apply equally to this project. They are provided here for your convenience:

- Source codes containing `NULL` will not be given as input.
- There are no variable definitions in the `compound_stmt` which is not a function body.
  e.g. `while(1) { int a; }`
- There are no implicitly declared functions, recursive functions or structs which contain themselves as a member.
- Functions cannot be overloaded.
- There are no assignments of a string constant to a pointer type.
  e.g. `char *s = "Hello World";`
- There are no operations between the pointer type and array type, or between the array type and array type. For example:

```
int *a;
int b[9];
a = b;
a == b;
```

- There are no functions without a return statement, except for a few open test examples where the return is omitted for readability.

5.2 Built-in functions

The subC code generator must provide three built-in functions:

- `write_int(int n)` : print an integer to `stdout`.
- `write_char(char c)` : print a character to `stdout`.
- `wirte_string(const char *s)` : print a string constant to `stdout`.
  - Note that there are no pointers that points to a string constant in subC.
  - That is, the only usage of this function is to print string literals,
    e.g. `write_string("Hello World!\n");`

When the code generator encounters calls to these built-in functions, it should generate appropriate intermediate codes to perform the specified actions.

5.3 Advanced Features

The following features are required to receive full credit for the project. If you have completed implementing each feature, please provide a description of your implementation in the report.

5.3.1 Loops

- Non-nested `while` loops
- Non-nested `for` loops
- Nested `while` loops
- `break`, `continue`

5.3.2 Struct operations

- Assignment operation between `struct` types
- Functions with `struct` return types
- Functions accepting `struct` as parameters

5.4 Output format

Do **NOT** emit the intermediate code to `stdout` or `stderr`. Please emit it to the specified file in the command line argument. The usage of subC code generator should be implemented as follows:

```
./subc [input.c] [output.s]
```

A demonstrative example of an input-output pair is given below.

I/O Example

Input code: `./open_test/basic/input.c`

```
int main() {
  write_string("hello␣world\n");
  return 0;
}
```

Generate an intermediate code with the input:

```
./src/subc ./open_test/basic/input.c ./open_test/basic/output.s
```

Generated intermediate code: `./open_test/basic/output.s`

```
  shift_sp 1
  push_const EXIT
  push_reg fp
  push_reg sp
  pop_reg fp
  jump main
EXIT:
  exit
main:
main_start:
str_0. string "hello world\n"
  push_const str_0
  write_string
  push_reg fp
  push_const -1
  add
  push_const -1
  add
  push_const 0
  assign
  jump main_final
```

```
main_final:
  push_reg fp
  pop_reg sp
  pop_reg fp
  pop_reg pc
main_end:
Lglob.  data 0
```

Execution result of `output.s` on the Stack Machine Simulator:

```
$ ./simulator/sim ./open_test/basic/output.s
hello world
program exits
```

6 Submission

- Place your report somewhere in the project directory. The file name must be `report.pdf`.
- Run `./submit.sh <student_number>` in the container. For example, `./submit.sh 1234-56789`. It will compress `src` directory and `report.pdf`.
- Submit the compressed archive on eTL.
- Please make sure that the name of the archive is `project4_<student_number>.zip` and your student number is correct. **Incorrect student number may result in a penalty to your project score.**
- For delayed submissions, there will be a 20% deduction in scores per day.

7 Tips

- Define your own calling convention to handle function calls.
- The following is a generally recommended order for proceeding with the project:
  1. Thoroughly review the lecture slides, as there are useful examples for code generation.
  2. Understand the exact behavior of the Stack Machine Simulator by examining `simualte_stack_machine()` function located around the line 357 in `./simulator/gram.y`.
  3. Add necessary fields to the symbol table designed in project #3, and ensure that the data structures are properly linked with each non-terminal.
  4. Modify `subc.y` to generate the code at the right position.
  5. Start by generating relatively simple code, and extend your code generator step by step.

Appendix

GNU Bison Manual