Project 2: Implementing a Code Generator

November 4, 2015

1 Introduction

In this project, you are required to implement a code generator to translate the intermediate representation, which is produced by your syntax analyzer implemented in project 1, into **LLVM** instructions. Your code generator should return a **LLVM** assembly program, which can be run on **LLVM** (http://llvm.org/). After finishing this project, you will get a compiler, which can translate Small-C source programs to **LLVM** assembly programs.

2 Step 1: Semantic Analysis (Optional)

Since not every program that matches the grammar is a semantically correct program, after generating a parse tree, we need to do semantic analysis and syntactic checking to examine whether there is any semantic error. For example, if a is an integer, and there is a statement "$a.b = 1$" in the program, your compiler need to report an error, because a is not an object.

In contrast to project 1, we do not have an off-the-shelf tool to rely on. Instead, all the codes should be written by yourselves. There are a number of things we can do as follows.

- Variables and functions should be declared before usage.
- Variables and functions should not be re-declared.
- Reserved words cannot be used as identifiers.
- Program must contain a function *int main*() to be the entrance.
- The number and type of variables passed should match the definition of the function.

- Using operator[] to a non-array variable is not allowed.
- Operator. can only be used to an object of a struct.
- *break* and *continue* can only be used in a for-loop.
- Right-value can not be assigned by any value or expression.
- The condition of *if* statement should be an expression with *int* type.
- The condition of *for* should be an expression with *int* type or ϵ.
- Only expression with type *int* can be involved in arithmetic.
- etc.

You can implement as many of them as possible, and describe your implementations in the project report.

3 Step 2: Optimization (Optional)

Please refer to our textbook for different ways of optimization. For example, we can do:

- Common subexpression elimination;
- Dead code elimination;
- Power reduction;
- etc.

4 Step 3: Machine-Code Generation

The last thing we need to do is to translate the intermediate representation into target codes. In our project, we choose the **LLVM** assembly codes to be the target language.

Here, we split code generation into two parts. The first part is instruction selection, and the second part is register allocation.

4.1 Instruction Selection

In this part, we assume that there are infinite registers. In other words, we do not need to consider which variables should be stored in register and which variables should be stored in memory. Therefore, the only thing we need to do is to choose suitable instructions and translate the intermediate representation into **LLVM** assembly codes. Note that there may be more than one ways to do the translation.

4.2 Register Allocation

Please implement a register allocation algorithm for your code generator. It is not required to apply the algorithm in the textbook. You can also implement your own algorithm.

4.3 Input and Output

To test your compiler, we need to add two special functions to Small-C . They are:

- read(int x); input an integer to x,
- write(int x); output the value of integer x in a line.

We need to first revise our token list and grammar to support these two functions, and then refer to the **LLVM** documentation to find out how **LLVM** hands I/O.

Requirements

1. Pack the source files into a file named StudentID.rar. Use meaningful names for the files, so that the contents of the file are obvious. A single makefile that makes the executables out of all the source codes should be provided in the submission. Enclose a README file that lists the files you have submitted along with a one sentence explanation.
2. The name of your executable file of compiler MUST be **scc**.
3. Your analyzer will be tested by the following command:
   ./scc “Source file name” “Output file name”.
   Your analyzer needs to read source code from the source file, and outputs the compiled **LLVM** assembly code to the output file. If there

is any error, output “Error.” to the output file. Please output other information to *stderr*.

4. Please state clearly the purpose of each program at the start of the source program, and clearly comment your programs.

5. Submit a well-organized report in **PDF** format. Since this project is really large, you need to state clearly how you design your compiler in your report.

6. Make sure your program can pass the pretest, which can be a toy Small-C created by yourself. If you do not handle I/O, or there is any other reason causing that your program cannot pass the pretest, please write down the possible reasons in your report. Then your program will be tested manually.

7. Send your StudentID.rar file to sjtucs215@163.com.

8. Due date: Jan. 16, 2016, midnight.