CS323 Project 3 Report

11812804 董正

11813225 王宇辰

1 Introduction

1.1 Project Requirements

In project 3, our compiler will generate a particular intermediate representation (IR) for a given source program. The IR can be further optimized for better runtime performance.

- Support `read` and `write` function.
- Print out IR for given SPL programs.
- IR can execute correctly in `irsim`.
- Optimize IR to minimize number of executed instructions.

We did not implement bonus part, i.e. there are no arrays or structures.

1.2 Development and Test Environment

- Linux
  - `Ubuntu 18.04.6 LTS x86_64` with `Linux 5.4.0-87-generic`
  - `gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0`
  - `g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0`
  - `flex 2.6.4`
  - `bison (GNU Bison) 3.0.4`
  - `GNU Make 4.1`
- macOS (Arm)
  - `macOS 12.0.1 21A559 arm64` with `Darwin Kernel Version 21.1.0`
  - `Apple clang version 13.0.0 (clang-1300.0.29.3)`
  - `flex 2.6.4`
  - `bison (GNU Bison) 3.8.2`
  - `GNU Make 3.81`

2 Design and Implementation

2.1 Translation

For IR generation, we designed a class `Generator` to traverse the parse tree after semantic check.

On top levels, it will do nothing. Most codes are in tree nodes related to functions, arguments, statements, and expressions. For these translations, we refered to pseudocode in table 1-4 in project document.

For example:

```
// IF LP Exp RP Stmt ELSE Stmt
string lb1 = createLabel();
string lb2 = createLabel();
string lb3 = createLabel();
string code1 = translateCondExp(node->child[2], lb1, lb2) + "LABEL " + lb1 + " :\n";
string code2 = translateStmt(node->child[4]) + "GOTO " + lb3 + "\nLABEL " + lb2 + "
:\n";
string code3 = translateStmt(node->child[6]) + "LABEL " + lb3 + " :\n";
return code1 + code2 + code3;
```

In addition, we created a new map `vmap` to store variable names such as `v0, v1`. They are created and added to `vmap` in `translateVarDec` function.

Speaking to functions, we need to predefine `read` and `write` functions and add them to symbol table. This is done in `function_init()`. For other functions, print `FUNCTION f :` in `translateFunDec` and then analyze and print its parameters in `translateParamDec`.

`Exp` is the most complex part in the whole program since it has many branches. Luckily we have reference in project doc. Therefore, it is not hard to implement, but requires carefulness.

2.2 Optimization

In this project, we optimized redundant codes.

For example, in the 5th test,

```
a = 1;
b = 2;
c = 3;
```

And they are never used. Therefore, they should not be translated to IR.

To achieve this, we designed a **dependency graph**.

First we created a new node class for the graph. It contains node name and its adjacent list.

```
class DNode {
public:
    string name;
    deque<DNode *> adj;

    bool visited;

    DNode(string name) {
        this->name = name;
        this->visited = false;
    }
};
```

Then we generate the graph when translating corresponding productions. It is a directed graph.

For example:

```
v1 = v2 + v3;
if (v4 == v5 + v6) {
  t1 = 0;
  v7 = t1;
  write(v7);
}
```

Then the dependency graph is:

```
v1 -> v2; v1 -> v3;
root_write -> v7 -> t1
root_cond -> v4; v4 -> v5; v4 -> v6;
```

Then we start BFS at all roots, then `v1, v2, v3` are not visited, which means they are useless. And in fact they are really useless because they are not printed or used in other operations. `v1 = v2 + v3;` is a redundant line. Therefore, for these unvisited variables, we will not generate IR for them.

Therefore, in the 5^{th} test, there is no IR generated for `a, b, c`.

What's more, we also optimized temporary variables for `IF` and `RETURN` when there are immediate number or single variable.

For example:

```
if (v1 == 0) {
  return v1;
}
```

Before optimization:

```
t1 := #0
IF v1 == t1 ...
...
t2 := v1
RETURN t2
```

After optimization:

```
IF v1 == #0 ...
...
RETURN v1
```

By these optimations steps, we have decreased the number of instructions substantially.

3 Conclusion

In this project, we implemented an generator of parse tree to generate IR codes. The most difficulity we met is how to return and print these codes when traversing the tree. What's more, the dependecy graph is really a challenge for us since it changed many logic in the whole program. And there is an interesting thing that our code runs correctly on our m1 Macs, but failed in Ubuntu. Finally we found out that this problem is about different compilers.