CS323-Compiler-Phase3-Report

November 2022

XuJia Zhang SID: 12011122
Qi Zhang SID: 12010123
ChengHuan Huang SID: 11912908

1 Introduction

This is the third phase of CS323 project-design a compiler for the SPL language.

- In phase 1, we have completed the tasks of lexical and syntax analysis, which end up with a syntax tree representation for syntactically-valid SPL program.
- In phase 2, we determine whether the SPL program is legal or not through semantic analysis.
- In phase 3, we can ensure that all the program in this phase will not have any grammer mitake or they will not pass phase 1 and phase 2 analysis.It's time for us to transform the original program to IR, which is tree address code (TAC) in our design.

2 Requrired Rules

2.1 Assumptions

This project has the following assumptions

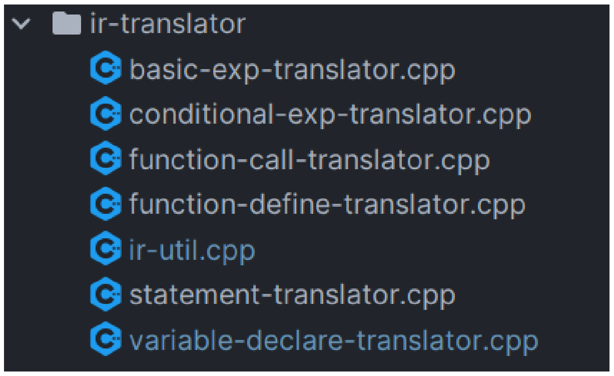
- Assumption 1: all tests are free of lexical/syntax/semantic errors (suppose there are also no logical errors)
- Assumption 2: there are only integer primitive type variables
- Assumption 3: there are no global variables, and all identifiers are unique
- Assumption 4: the only return data type for all functions is int
- Assumption 5: all functions are defined directly without declaration

- Assumption 6there are no structure variables or arrays
- Assumption 7function's parameters are never structures or arrays

3 Implementation

3.1 Detail

- 1) we use the top down approach to implement the phase 3.
- 2) we divide all the cases into 5 major cases and implement them respectively.



```
ir-translator
    basic-exp-translator.cpp
    conditional-exp-translator.cpp
    function-call-translator.cpp
    function-define-translator.cpp
    ir-util.cpp
    statement-translator.cpp
    variable-declare-translator.cpp
```

- 3) We use C++'s unordered_map library to maps a variable to its associated register and the symbolTable from phase 2 to get the function name when we parsing the function node.
- 3) Phase 3's project structure is more professional than phase 2.(Please check our project for more info)

3.2 Project Structure

We refactor our project structure to make it more intuitive.

- 1) we use include directory to manage the hpp files
- 2) we use -j8 optimization to make makefile's compile process faster
- 3) Phase 3's project structure is more professional than phase 2.

3.3 enum class: TAC$_TYPE$

```
enum class TAC_TYPE {
    LABEL,
    FUNCTION,
    ASSIGN,
    ADDITION,
    SUBTRACTION,
    MULTIPLICATION,
    DIVISION,
    GOTO,
    CONDITION_LT,
    CONDITION_LE,
    CONDITION_GT,
    CONDITION_GE,
    CONDITION_NE,
    CONDITION_EQ,
    RETURN,
    PARAM,
    ARG,
    CALL,
    READ,
    WRITE
};
```

3.4 Class: TAC

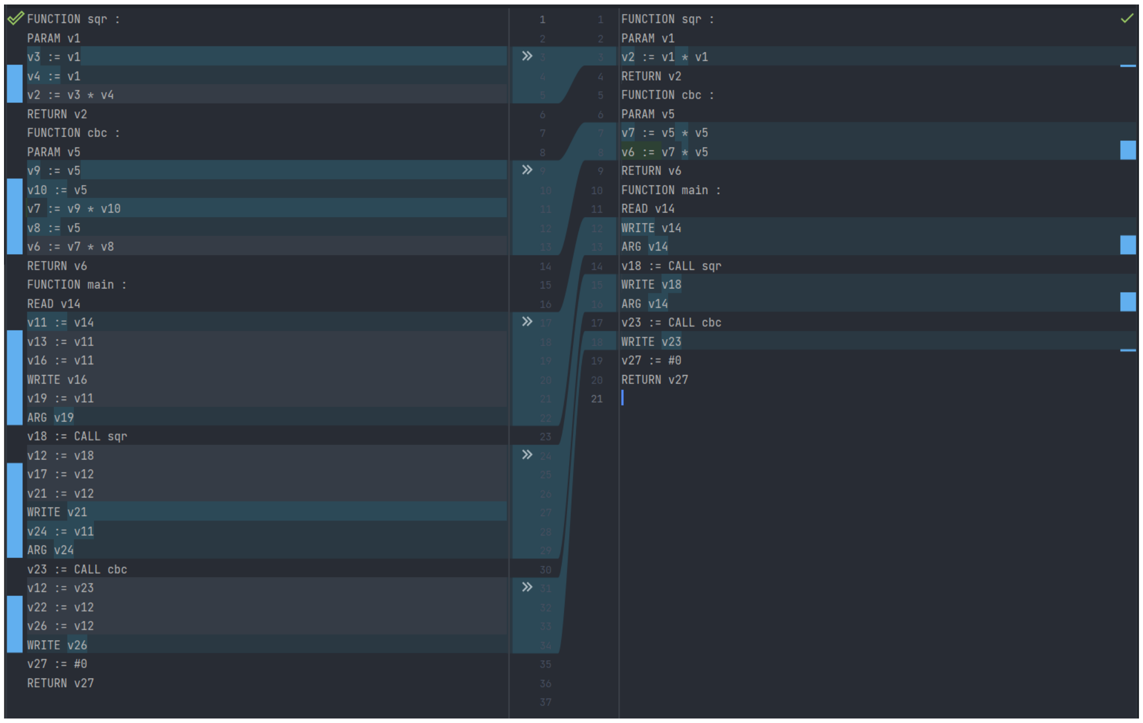
- TAC is the class we design to store the data of a single three address code. We found that all the three address code can be represent by 3 operator and a type.
- So we can use a vector container to store all the TAC we get in the analysis process.Then swap or delete some of them without changing the meaning of it to make it faster.

```
class TAC {
public:
    string X;
    string Y;
    string Z;
    enum TAC_TYPE type;
};
```

3.5 optimization

Because of COVID, we didn't do much in the optimization part and report.

- 1) register optimization
  By using the production and SDD provided by Prof.Liu, we notice that the result TACs have multiple duplicated assign operation or the reg is assigned by it self. We should delete these duplicated assign TACs and change the name of these register in the relevant TAC, such as plus,relop and so on.



```
FUNCTION sqr :
PARAM v1
v3 := v1
v4 := v1
v2 := v3 * v4
RETURN v2
FUNCTION cbc :
PARAM v5
v9 := v5
v10 := v5
v7 := v9 * v10
v8 := v5
v6 := v7 * v8
RETURN v6
FUNCTION main :
READ v14
v11 := v14
v13 := v11
v16 := v11
WRITE v16
v19 := v11
ARG v19
v18 := CALL sqr
v12 := v18
v17 := v12
v21 := v12
WRITE v21
v24 := v11
ARG v24
v23 := CALL cbc
v12 := v23
v22 := v12
v26 := v12
WRITE v26
v27 := #0
RETURN v27
```

```
FUNCTION sqr :
PARAM v1
v2 := v1 * v1
RETURN v2
FUNCTION cbc :
PARAM v5
v7 := v5 * v5
v6 := v7 * v5
RETURN v6
FUNCTION main :
READ v14
WRITE v14
ARG v14
v18 := CALL sqr
WRITE v18
ARG v14
v23 := CALL cbc
WRITE v23
v27 := #0
RETURN v27
```

Figure 1: Reg optimization example

- 2) label optimization
  By using the production and SDD provided by Prof.Liu, we notice that the result TACs have multiple consequent label TAC. Only the first label of these multiple consequent labels is necessary. We should delete the duplicated labels and change the name of these register in the relevant conditional TAC.

4 Environment

Tool	Version
OS	WSL2(Ubuntu 20.04)
IDE	VScode 1.72.2
Bison	3.5.1
Flex	2.6.4
Python	3.8.10
g++	9.4.0
makefile	4.2.1