CS323-Compiler-Phase2-Report

November 2022

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

1 Introduction

This is the second phase of CS323 project. In first phase, we have completed the tasks of lexical and syntax analysis, which end up with a syntax tree representation for syntactically-valid SPL program.
In this phase, we should determine whether the SPL program is legal or not.

2 Requrired Rules

2.1 Assumptions

This project has the following assumprions

- Assumption 1 char variables only occur in assignment operations or function parameters/arguments
- Assumption 2 only int variables can do boolean operations
- Assumption 3 only int and float variables can do arithmetic operations Assumption 4 no nested function definitions
- Assumption 5 field names in struct definitions are unique (in any scope), i.e., the names of struct fields, and variables never overlap
- Assumption 6 there is only the global scope, i.e., all variable names are unique
- Assumption 7 using named equivalence to determine whether two struct types are equivalent

2.2 semantic errors

Semantic analyzer should be able to detect the following semantic errors

- Type 1 a variable is used without a definition
- Type 2 a function is invoked without a definition
- Type 3 a variable is redefined in the same scope
- Type 4 a function is redefined (in the global scope, since we don't have nested functions)
- Type 5 unmatching types appear at both sides of the assignment operator (=)
- Type 6 rvalue appears on the left-hand side of the assignment operator
- Type 7 unmatching operands, such as adding an integer to a structure variable
- Type 8 a function's return value type mismatches the declared type
- Type 9 a function's arguments mismatch the declared parameters (either types or numbers, or both)
- Type 10 applying indexing operator ([...]) on non-array type variables
- Type 11 applying function invocation operator (foo(...)) on non-function names
- Type 12 array indexing with a non-integer type expression
- Type 13 accessing members of a non-structure variable (i.e., misuse the dot operator)
- Type 14 accessing an undefined structure member
- Type 15 redefine the same structure type

3 Implementation

3.1 SymbolTable

We use C++'s unordered_map library to maps a name to its associated information. The notion "name" is the identifier which includes (but is not limited to): variable names, function names, user-defined type names, labels; "information" has a broader meaning, such as the type of identifier, the data type, array's dimension, numerical values, etc.

```
#include <unordered_map>
using std::unordered_map;
unordered_map<string,Type*> symbolTable;
```

3.2 SAUtill

getReturnExpFromCompSt()	return Expression contains "RETURN"
countParamNum()	return number of parameters
vectorTofieldlist()	convert vector into list
typeCheckingMethods	check the input type

3.3 type.hpp

To indicate the information of the name, we use type.hpp to show the type of the name.

- PRIM represents Primitive Types. Primitive Types are the base types of the language. These types are provided directly by the underlying hardware. They are int, char and float
- Array class represents Array Types. Array Type contains size and base, size is the amount of data array can hold and the base is the type of array.
- FieldList param represents parameter in the function definition
- FieldList structure represents variables defined in structure.

```
enum class PRIM
{
    INT,
    FLOAT,
    CHAR
};
enum class CATEGORY
{
    PRIMITIVE,
    ARRAY,
    STRUCTURE,
    FUNCTION,
};

/// @brief
class Type
{
public:
    string name;
    enum CATEGORY category;
    union
    {
        enum PRIM primitive;
        Array *array;
        FieldList *structure;
        FieldList *param;
    } foo;
    Type *returnType = nullptr;
    ...
};
class Array
{
public:
    int size;
    Type *base = nullptr;
    ...
};
class FieldList
{
public:
    string name;
    Type *type = nullptr;
    FieldList *next = nullptr;
    ...
};
```

3.4 semantic errors

3.4.1 define and redeine error

Error Type 1,2,3,4,15 are all defined and undefined issues, and these errors can be easily identified as long as we register them in the symbol table.

- For error type 1,3 In synatex.y, When reducing to def, there is a Specifier in the reduction to indicate that the variable is defined at this time, we can identify the variable name here and add it to the symbol table. The next time it reappears, the error is detected.

```
Def:Specifier DecList SEMI
```

- For error 2,4, we define the function in Fundec

```
FunDec:ID LP VarList RP|ID LP RP
```

- For error 15, we define the structure in ExtDef

```
ExtDef: Specifier ExtDecList SEMI | Specifier SEMI
```

3.4.2 unmatching operands

Error Type 5,6,7 are all unmatching operands issues, s, and these errors can be easily identified as long as check type both sides when the operands appear in exp reduction.

```
Exp:  Exp  ASSIGN  Exp|Exp  LT  Exp|Exp  PLUS  Exp
```

3.4.3 dot misuse

Error Type 13,14 are all dot misuse, and these errors can be easily identified as long as check type both sides when the operands appear in exp reduction.

```
Exp:Exp  DOT  ID
```

3.4.4 returnType error

Error Type 8 is returnType error, we get returnType in exp redeuction and check return type in ExtDef reduction.

```
ExtDef:FunDec  CompSt
...
Exp:ID  LP  RP|ID  LP  Args  RP
```

3.4.5 param error

Error Type 9 is param error, we check param type in exp reduction.

```
Exp:ID  LP  RP|ID  LP  Args  RP
```

3.4.6 array misuse error

Error Type 10,12 are array misuse,we check type's category and array base type in exp reduction.

```
Exp:LB  Exp  RB
```

3.4.7 function misuse

Error Type 11 is the function misuse, we check the variable type in exp reduction.

```
Exp:ID  LP  RP|ID  LP  Args  RP
```

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