Report for CS323-Project2

Jin Zhaoxuan
11911413

Xu Tianqi
11912921

Zhao Yunlong
11911309

I. Overview

Our program has used *L-attributed syntax-directed definition* (SDD) to implement a semantic analysis, which is based on two self-written data structure models, *Type* and *Symbol table*.

We have invested a lot of effort to make our compiler easier to use and it has the following main features:

- Multiple scope design. Variables in different scopes can share the same identifier, and variables defined in the outer scope will be shadowed by the inner variables with the same identifier.
- Structure equivalence. While determining the type equivalence of two *struct* variables, the compiler will use structure equivalence rather than name equivalence, so that even two types are different, their variables can be assigned with each other as long as they have the same field structure.
- Extra error types. Besides all the 15 error types, it can detect other 5 error types to cover almost all the assumptions in the document and some errors not mentioned.
- User-friendly error reporting. The compiler will suppress the subsequent errors cascading from same problem, which makes the error messages briefer and more understandable. Besides, our compiler will provide a detailed information of error, such as the type information in mismatch type errors and the variable names in undefined variable errors.

In this report, we will start from introducing some basic data structures we used in this project to support the analysis. Then we will talk about the process of the semantic analysis and introduce the implementation of the features above in detail. At last, we will point out the differences between the example result and our compiler's result, then explain the reasons.

II. Data Structures

A. Type

We defined a `Type` structure with three fields, `category`, `name` and `data`, to represent all the types. The field `category` is an enumeration indicating the category of the type, the field `name` indicates the name of this type (only for array and structure types), and `data` is an union storing each type's information, which we will explain later.

```
struct Type{
    enum{
        PRIMITIVE,
        ARRAY,
        STRUCTURE,
        FUNCTION
    } category;
    const char *name;
    union{
        PrimitiveType primitive_type;
        Function *function;
        ArrayInfo *array_info;
        FieldNode *field_list;
    };
};
```

`PrimitiveType` is an enumeration to represent which primitive type the type exactly is. `P_INT`, `P_FLOAT`, `P_CHAR` stand for int, float, char separately.

```
enum PrimitiveType
{
    P_INT,
    P_FLOAT,
    P_CHAR
};
```

The struct `Function` is a composition of return type (a field of `Type` type) and arguments (stored as a linked list of type `ArgNode`).

```
struct ArgNode
{
    Type *type;
    ArgNode *next;
};

struct Function
{
    Type *return_type;
    ArgNode *arg_list;
};
```

Struct `ArrayInfo` stores the information of array containing the size and the type of the array's elements.

```
struct ArrayInfo
{
    Type *base;
    int size;
};
```

Struct `FieldNode` stores the name and type of current field, and we use a linked list of `FieldNode` to represent the structure of a *struct*.

```
struct FieldNode
{
    const char *name;
    Type *type;
    FieldNode *next;
};
```

B. Type Comparison

We offered a function `compare_type()` to the analyzer to help it identify if two type is equal. All of the four types can be compared by this function, and we will compare different types using different strategies.

- Two primitive types are equivalent if their primitive type is the same.
- Two array types are equivalent if the size and the base type is the same.
- Two structure types are equivalent if each of their field's types are equivalent (in the order of field list).
- Two function types are equivalent if the return type is the same and each of their argument's types are equivalent.

C. Symbol Table

We chose hash map as the data structure of the the symbol table to obtain a higher efficiency of operating the symbol table. We used *PJW hash function* [1] as our hash algorithm and separate chaining is applied to avoid hash conflict. If a conflict occurs, the inserted node will be appended to the end of the linked list. When we look up the element, we will first calculate the hash of key to find the linked list, and then compare each key in the linked list with the given key by string comparison to find the exact element.

To let the analyzer operate the symbol table, we offered some functions including `insert_symbol()` and `find_symbol()`, etc. In these functions we will operate the hash maps for the analyzer, like inserting pairs into the hash map corresponding to current scope's symbol table, searching the innermost variable of the given key in the hash maps, etc. After that, the result will be returned to the analyzer.

D. Scope

Scope is a section of program text enclosed by `{` and `}` (in most cases), and the struct representing a scope is shown below.

```
struct Scope
{
    int scope_level;
    HashMap symbol_table;
    HashMap structure_prototype;
    Scope *last_scope;
};
```

Each scope has two hash maps:

- `symbol_table` stores variables and their names declared in the scope. Since we regard functions as a kind of variable, they will be stored in symbol table as well.
- `struct_prototype` stores the structure types defined in this scope and their names.

When the analyzer wants to find a variable, the compiler will look up the variable from the symbol table of the current scope to the symbol table of the outermost scope and return the first variable it found. we can see that each scope will have a pointer to the previous scope, which makes traversing from inner to outer scope easy.

We offered two functions, `enter_scope()` and `exit_scope()` to the analyzer. When the the analyzer enters a scope, it will call `enter_scope()` and a new scope will be generated. When `exit_scope()` is called, the innermost scope will be destroyed.

III. Semantic Analysis

In semantic analysis program, we use bottom-top way to traverse the parsing tree we generated in previous project and use L-attributed SDD to resolve dependence. We create at least one functions for every production in the CFG. To manage some productions that will appear in different structure, we create multiple functions to analysis it, for example, the production $VarDec \rightarrow ID$ will perform in different ways as structure declaration, as function declaration, or as simple statement.

A. Variable declaration

Every Nonterminal $Specifier$ has a synthesized attributes $Specifier.type$, which have structure `Type` we have introduced above. Then we will use it in production $ExtDef \rightarrow Specifier\ ExtDecList\ SEMI$, and *ExtDecList* will have a inherited attributes $ExtDecList.type$ from $Specifier$.

B. Multiple scope

There are three different places we should enter a new scope.

- Enter at the beginning of program and leave at end of program.
- Enter before *VarList* in production $FunDec \rightarrow ID\ LP\ VarList\ RP$ and leave at end of the function body.
- Enter at beginning of every *CompSt* and leave at end of one.

A example is given in `ex-test1.spl` to help you understand our strategy of multiple scope.

C. Structure equivalence

Due to the function `compare_type` of structure `Type`, we can easily implement structure equivalence in semantic analysis. We provided a detailed testcase `ex-test2.spl` to show our structure equivalence feature.

D. Extra grammar rules

To handle more complex situation, we have changed some rules in our SPL:

- == and != can be used with any type including structure and array as long as LHS and RHS are same type. The whole expression will be regarded as int type.
- <, >, <=, and >= can be used with int, float, and char. The whole expression will be regarded as int type.
- Assign operation = can be used with any type including structure and array as long as LHS and RHS are same type. The whole expression will be regarded as type of operator.

E. Extra error types

We design 4 extra error types to handle our new rules and some assumption in requirement:

- Error type 16: other types rather than int type occurs in boolean expression, such as IF expression and WHILE expression.
- Error type 17: size comparison of non-primitive types.
- Error type 18: other types rather than int and float variables do arithmetic operations.
- Error type 19: define a variable with undefined structure type name.

We provided a detailed testcase `ex-test3.spl` to explain extra rules and extra error types.

F. Error reporting

The compiler will suppress the subsequent errors cascading from same problem. More specifically, if a expression is invalid, its attribute `Exp.type` will become undefined, and any error caused by an undefined type will be ignored. This method guarantees that the reported errors are exactly at the most bottom place of the parsing tree where problem occurs, and ignore any other errors caused by this problem. We provided a detailed testcase `ex-test4.spl` to show our error suppression.

Another feature we implement in error reporting is detailed information of error. As you can see, our output of any testcase contain extra information of the specific error. For example:

- Name of undefined and redefined variables and functions.
- Type names of unmatching types appear at both sides of the operates.
- Reason why function's arguments mismatch the declared parameters, types or numbers.

IV. Differences to Example Results

As we declared above, our compiler will suppress the subsequent errors cascading from same problem, which makes it performed a little bit different from the example output. The different testcases are `test_2_r07`, `test_2_r12`, and `test_2_r14`.

References

[1] Wikipedia contributors, "Pjw hash function — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=PJW_hash_function&oldid=997863283, 2021, [Online; accessed 19-November-2021].