CS323 SPL Parser - Phase 2

陈康睿(A)	樊斯特(A)	肖佳辰(A)
12110524	12111624	12112012

1 Quick Start

1.1 Project Structure

```
12110524-12111624-12112012-phase2
│  Makefile
│  test_ex.sh
│  test_others.sh
│  test_self.sh
│  test_std.sh
│  // test scripts
│
├─bin
│      splc
│
├─report
│      12110524-12111624-12112012-phase2.pdf
│
├─src
│      lex.l
│      ortho.c
│      ortho.h
│      syntax.y
│      treeNode.c
│      treeNode.h
│      type.c
│      type.h
│      type_op.c
│      type_op.h
│      uthash.h
│      utstack.h
├─test
│ // self-written testcases for basic features
│
├─test-ex
│ // self-written testcases for bonus features
│
└─test-std
  // official testcases
```

1.2 Environment

Tool	Version
C	C11
Bison	3.0.4+
Flex	2.6.4

1.3 How to Run

Under the root folder of this project, execute the following commands to generate the parser executable `splc` under `./bin` folder:

- `make splc`: Generate the parser, and clean the intermediate files.(For Bison 3.0.4)
- `make splcb`: Generate the parser, and clean the intermediate files.(For Bison 3.8.2)
- `make splcc`: Generate the parser, reserve the intermediate files.(For Bison 3.8.2)
- `make splcd`: Generate the parser, print examples of shift-reduce conflicts.(For Bison 3.8.2)

Execute `make clean` to clean the `./src` and `./bin` folder.

For an SPL source file `filename.spl`, execute `./bin/splc filename.spl` to write the output into `filename.out`; or execute `./bin/splc filename.spl result.out` to write the output into the given file `result.out`.

2 Basic Features

In this phase, we implement a semantic analyzer under the assumptions according the the project instruction. The incremental contents are mainly three sections: `type`, `ortho`, and `type_op`.

`type` defines the structure Type maintaining type information including `typesize`, `category`, and extended pointer for special types like Array, Structure, and Function. It also contains basic operations between Types.

`ortho` implements an orthogonal list to support scope operations. The details will be elaborated in the following Bonus Features part.

`type_op` links all other parts in the project. The functions inside realize semantic actions and are invoked in `syntax.y`, performing type info maintenance for tree nodes and scope stack control.

3 Bouns Features

3.1 `break` and `continue`

Our compiler in Phase 2 supports semantic checking of the break and continue keywords. If `break` or `continue` occurs out of a `for` or `while` loop, then the compiler will raise a semantic error.

This is done by maintaining a global variable `loop_cnt` in `syntax.y` to to keep track of the number of levels in the loop, and raise an error if `break` or `continue` occurs when `loop_cnt=0`, i.e., outside of any loop context.

3.2 Local Scopes

To maintain what variables are abailable in different scopes, we implement an orthogonal list to record the variables and structures defined/declared in a local scope.

The orthogonal list has two dimensions: **Hash table** and **Stack**.

The stack is introduced to represent scopes. When entering a new block surrounded by curly brackets `{}`, we push a new stackNode onto the stack top. While leaving a scope, we free all variables in the top scope, and then pop the stack top so that variables defined in this scope is no longer accessible later.

The hash table is introduced to find the variables/functions with given name in the “nearest” scope. A linked list can be obtained by looking up the hash table, with all variables/functions with the same name, and variable within the “nearest” scope is stored as the head of this linked list, so we can retrieve the corresponding variable in the correct scope with a given variable name.

Therefore, each node, representing a variable or a function, is connected to the hash table and the stack by two linked lists. When defining/declaring a new variable (since function must be declared in the global scope, according to the syntax rules), we add this variable to the head of the linked list according to the hash table by its name, and add this variable to the head of the linked list pointed by the stack top representing the current scope. When leaving a scope, the compiler traverse the linked list representing variables in the current scope and free them from both the stack and the hash table.

For more detail, please check `ortho.h` and `ortho.c`, where this feature is implemented.

3.3 Structural Equivalence

Our compiler also supports structural equivalence rather than name equivalence, i.e., when checking if two structures are the same, check the types of their member variables are consistent, not just that their structure names are the same.

At the same time, our compiler supports the idea that structures have struct members (which is also a C feature), which raises the difficulty of implementing structural equivalence checking. To implement this, we check the types of the member variables **recursively**, by checking if the member struct has members of the same types.

3.4 Implicit Type Conversion

We introduce implicit type conversion mechanism, allowing assignment and calculation between different primitive types. So our compiler will not raise any error when doing assigning or calculation between primitive types(except `char`).

We implement this feature by invoking `checkPrimEqual()` in the beginning of `checkTypeEqual()`. This feature is extendable, so we can implement type narrowing and widening in Phase3.

4 Explanations for Test Cases

4.1 Basic tests

1. ttest_12111624_1.spl

   Test for array in structure.

2. test_12111624_2.spl

   Test for function useage.

3. test_12111624_3.spl

   Test for recursively function useage and mismatch of parameters.

4. test_12111624_4.spl

   Test for array in struct and struct array.

5. test_12111624_5.spl

Test for struct array and multiple assign(=).

4.2 Extra tests

1. test_1.spl

   Test for `continue` and `break` in `while` loops.

2. test_2.spl

   Test for nested structures.

3. test_3.spl

   Test for multi-level scope.

4. test_4.spl

   Test for structure equivalence.