Report for CS323-Project1

Jin Zhaoxuan
11911413

Xu Tianqi
11912921

Zhao Yunlong
11911309

I. Base requirement

For syntax valid SPL source codes, we use *bison* and *flex* to achieve the requirements. Besides, to reach the feature of printing token tree, we write a tree structure as a extra c lib named *tokentree.c*.

For syntax invalid situations, we achieved error recovery for almost all errors in SPL.

- Missing open, close or both parenthesises, brackets or curly braces.
- Missing expression after operator.
- Missing or redundant semicolon.
- Missing declaration content.
- Missing statement content.
- Redundant type specifiers.
- Misplaced definition.
- Invalid array declaration.

In error recovery implementation, we use c macro to reduce code redundancy and enhance project maintainability.

II. Bonus feature: preprocessor

Our preprocessor will mainly do three jobs.

- Link the source file with included files, and generate a linked file.
- Remove comments, including single-line and multi-line comments from the linked file.
- Expand all the macros in the linked file and generate a preprocessed file for main compiler to analyze.

A. File inclusion

In our implementation, we can use the following instruction to include other files.

```
#include "test_2.spl"
```

Note that our implementation only accept filenames inside double quotes. For filenames included in angle brackets, i.e. '<' and '>', we won't recognize it as a valid include directive.

The function below is used to implement file inclusion recursively.

```
void link_include(IncludedNode *included_list,
     const char *include_file_path, FILE *fd)
```

In this function, `included_list` is a list of filenames that are already included, `included_file_path` is the path of the file being analyzed now and `fd` is a file pointer pointing to the generated linked file.

Every time the function starts to analyze one file, it will first check if the file exists. If the file doesn't exist, it will report an error and ignore this file.

Then the function will check if the file is already included by checking if the filename is contained in `included_list`. If the file is already included, then the function will directly return to avoid loop include, which may cause a program crash. Else, if the file is not included, then add it to the `included_list`.

```
if (is_included(included_list,
    include_filename))
{
    fprintf(stderr, "[Ignored] Duplicate
    included file: %s\n", included_filename);
    return;
}
add_include(included_list, include_filename);
```

Then this function will analyze the included file line by line. If one line is a simple line of the source code, we directly output it to the intermediate file. Else, if this line starts with `#include`, the preprocessor will try to parse this line into a include directive.

In the parsing process, the preprocessor will first check if the inclusion grammar is leagal. If this line has more or less than two double quotes, the preprocessor will report an error, and ignore this included file by default.

```
> ./preprocess ../test-ex/test_2_include01.spl >> test_2_include01.out
[Ignored] Duplicate included file: test_2_include01.spl
[Ignored] Include error at file "../test-ex/test_2_macro01.spl" line 4: #include "test_2_macro05.spl
```

Fig. 1. Error detection in file inclusion

If the inclusion grammar is legal, then the preprocessor will extract the filename from the directive, and call function *link_include()* again to analyze the included file.

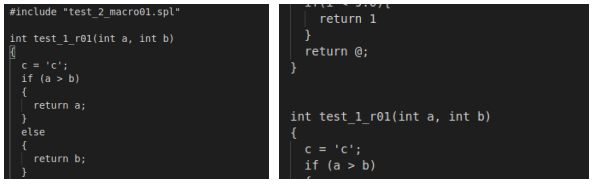

Fig. 2. Before and after file inclusion

This is a recursive process. When all the functions are returned, all the legal `#include` lines should be replaced with the content of the included files in the linked file, which is shown in figure 2 (the corresponding test case is `test_1.spl`).

B. Single-line and multiple-line comment

After file inclusion preprocess, the preprocessor will send the linked file into *flex* to perform a lexeme analysis only used in preprocess. In this analysis, the preprocessor will remove all the comments in the file.

For single-line comment, we use *input()* function provided by *flex* to capture characters until it reaches '\n'.

For multiple-line comment, we also use *input()* function. Besides, to detect invalid nested multiple-line comments, we design a simple finite state machine as figure 3.

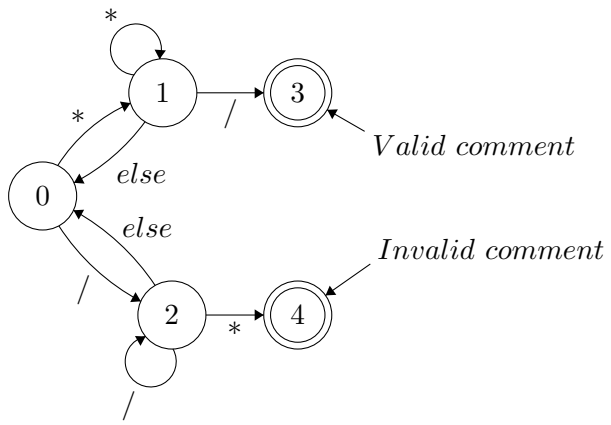

Fig. 3. Automata for multiple-line comment. State 3 shows the multiple-line comment ended and it is valid. State 4 shows the comment is invalidly nested with another multiple-line comment. Notice that the state machine is only entered after we getting a "/*".

C. Macro expansion

Our implementation of macro expansion can only expand object-like macros, so it will regard function-like macros and some special characters like # (stringlize) and ## (glue) as illegal grammar.

The lexeme analysis will not only remove the comments in the linked file, but will also break down the text that not inside comments into tokens at the same time. After the analysis, we will have a token sequence corresponding to the source code in linked file, then the token sequence will be send to function *expand()* for macro expansion.

```
void expand(Token *token_sequence)
```

Each token has a hide set which is initially empty. This set is used to record the macros that have generated this token before. For example, if a macro `TEST` expands to three tokens `a+3`, then `TEST` should be added to each of the three token's hide set. We use hide set to ensure that every macro is expanded only once, so if there are loop definitions, no matter directive (e.g. `#define TEST TEST + 3`) or indirective (e.g. `#define TEST TEST1` and `#define TEST1 TEST`), there will not be infinite expansion. The loop definition can be tested with test case `test_7.spl`.

The function will first initialize a macro set, which will contain all the macros in effect, and elements will be added and deleted from it during the execution of *expand()* (This will be introduced in detail in the following paragraphs). Then the function will go through all the tokens in the sequence and do process according to the attributes of the current token.

If it find that the current token is a token represents `#define`, which means this line is a macro definition directive, then the function will try to extract the macro and the substitute token, and then add the macro into the macro set. If the function detects an error, such as incomplete macro or duplicate definition, it will report the error and ignore the definition by default, which is shown in figure 4 and figure 5.

```
#define DUPLICATE int
#define DUPLICATE float

DUPLICATE test_1_r07()
{
  DUPLICATE a, b, c;
  a = 1;
  b = a && 2;
  c = b || 3;
  a = c | 4;
  b = a & 5;
}
```

Fig. 4. Duplicate definition

After that, this line will be removed from the token sequence since macro directives should not exist in the

preprocessed file.

```
> ./preprocess ../test-ex/test_2_macro03.spl >> test_2_macro03.out
[Ignored] Macro error at line 7: DUPLICATE can not be defined twice
```

Fig. 5. Duplicate macro error detection

Else, if the function find that the current token is a token represents `#undef`, which means this line is a macro undefinition directive, it will try to extract the macro and delete the macro from the macro set. Similarly, there will be some errors like undefining a not defined macro, and the function will report the error and ignore the directive by default.

If the function find that the token is a simple token, then the function will first check if the token is inside its own hide set. If the result is true, then the function will simply pass the current token and go to check the next token, else the function will check the macro set to see if the token is a macro in effect. If the token is a macro in effect, then replace it with the substitute token sequence of the macro, add the macro into each sub-token's hide set, and continue analyze the token sequence from the head of the substitute token sequence.

After the macro expansion process, we will have a modified token sequence representing the preprocessed source code, which have all the include files linked, comment removed, macro expanded and all the preprocess directives removed. At this time all the preprocess jobs are done, the token sequence will then be converted back into text and be sent to the main compiler for lexeme analysis and syntax analysis.

D. Generate intermediate file

The preprocessor is already integrated in the compiler `splc`. By default `splc` will only generate a `.out` file. However, user can use flag `-i` to tell `splc` to also generate the intermediate file, which is the preprocessed source code file.

```
bin/splc -i [intermediate file path] [spl file
    path]
```

Then `splc` will generate the `.out` file as well as the intermediate file.

E. Extra test cases

The test files from `test_1.spl` to `test_7.spl` in folder `test-ex` are test cases used to test the preprocessor. We have written comments in these files to explain the purpose of each test case. Except from the `.out` files, we also provide a `.itm` file for each of the seven test cases, which are the preprocessed source code file generated by adding flag `-i`. The performance of the preprocessor can be evaluated by checking these `.itm` files.

III. Bonus feature: "for" statement

We imitate c-style "for" statement in our SPL compiler, which looks like:

```
for (i=0;i<b;i=i+1) {}
```

The tokens between parenthesises of for must have exactly two semicolons. The semicolons divide the sentence into three expressions. The first expression is initial expression executed before for loop. The second expression is condition which control if the loop should continue. The third expression is executed after each cycle. Every expression can be empty if necessary.

A. Implementation

1) Lexical analysis: We use a token "FOR" in lexical analysis part, which only matches string "for".

2) Syntax analysis: We add a non-terminal "ForArgs" to represent tokens between parenthesises. Therefore, we add "for" statement in *Stmt* as:

```
Stmt: FOR LP ForArgs RP Stmt
```

While "ForArgs" have to match eight situations of sentences because each of three expression may be empty.

```
ForArgs: Exp SEMI Exp SEMI Exp
| Exp SEMI Exp SEMI Exp
| SEMI Exp SEMI Exp
| Exp SEMI SEMI Exp
| Exp SEMI Exp SEMI
| SEMI SEMI Exp
| SEMI Exp SEMI
| Exp SEMI SEMI
| SEMI SEMI
```

B. Error recovery

Except basic errors such as missing parenthesises or missing statement, we implement an *Invalid for statement error* for "for" statement only. which means error occurs in *ForArgs* nonterminal. For example, Missing semicolon or duplicated semicolon will raise a *Invalid for statement error*.

C. Extra test cases

The test files from `test_8.spl` to `test_10.spl` in folder `test-ex` are test cases used to test the for statement. `test_8.spl` gives some valid examples of using for statement while `test_9.spl` gives some invalid examples. We implement a bubble sort in `test_10.spl` to show that for statement can be used with any other statements correctly.