CS323 SPL Compiler - Phase 1

GuTao
GuTaoZi@GitHub

Artanisax
Artanisax@GitHub

ShadowStorm
Jayfeather233@GitHub

1 Quick Start

1.1 Project Structure

```
Phase_1
├── Makefile
├── report
│   ├── Phase1.pdf
│   └── Requirements.pdf
├── src
│   ├── lex.l
│   ├── syntax.y
│   ├── treeNode.c
│   └── treeNode.h
├── test         // self-written testcases for basic features
├── test-ex      // self-written testcases for bonus features
├── test_ex.sh
├── test-others  // testcases by other students
├── test_others.sh
├── test_self.sh
├── test-std     // official testcases
└── test_std.sh
```

1.2 Environment

Tool	Version
C	C99
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

2.1 Lexical Part

In the file lex.l, we detect valid and invalid tokens and build nodes of the parse tree for them, and invalid tokens are used for error recovery for subsequent syntax analysis.

In this section, we define a has_error variable to record whether or not an error occurred during compilation. has_error is initialized to 0 and is set to 1 after either a TYPE A or TYPE B error. The parser will output a parse tree after analysis if has_error is 0, otherwise it will report the errors.

2.2 Syntactical Part

In the file syntax.y, we formulate the matching rules used to analyze the syntax and use the node pointers of the parse tree as nonterminal values, so that a complete parse tree can be constructed by Bison's syntax analysis.

2.3 Parse Tree

treeNode.c and treeNode.h define the structure treeNode of the syntactic analysis tree, as well as a set of functions necessary to build the tree. Each tree node holds the following variables:

- name: the name of the node of the (non)terminal
- val: the content of the terminal
- lineno: the line number in the source file that corresponds to this (non)terminal
- child: head of the list of child nodes
- nxt: pointer to a sibling node

3 Bouns Features

For bouns features, we implemented for statements, include statements, single-line and multi-line comments, and the parser can match string with multiple lines.

3.1 for Statements

We add extra patterns to Stmt in syntax.y:

```
Stmt:
  ...
  | FOR LP DecList SEMI Exp SEMI Exp RP Stmt %prec UPPER_FOR
  | FOR LP VarDec COLON Exp RP Stmt %prec LOWER_FOR
  | FOR DecList SEMI Exp SEMI Exp RP Stmt %prec UPPER_FOR
  | FOR LP DecList SEMI Exp SEMI Exp error Stmt %prec UPPER_FOR
  | FOR VarDec COLON Exp RP Stmt  %prec LOWER_FOR
  | FOR LP VarDec COLON Exp error Stmt  %prec LOWER_FOR
```

So the parser can recognize for(;;) and for(:) in C89 style.

3.2 File Inclusion

We modified the high-level specification:

```
Program : HeaderDefList ExtDefList
    ;
HeaderDefList :
    | Headers HeaderDefList
    ;
Headers : IncDef
```

```
    ;
IncDef : SHARP INCLUDE ABSTR
    | SHARP INCLUDE error
    ;
```

So the parser can recognize programs with/without file inclusions in format of `include <filename>`.

3.3 Comments

In `lex.l`, we designed regexps to match single-line comments or multi-line comments. The parser simply igores the comments during syntax analysis.

The parser will output line numbers of comments before printing the parse tree or reporting errors, if any comment recognized.

```
  "//".*$ {
    fprintf(yyout,"Single LINE COMMENT at Line %d\n", yylineno);
  }
  "/*"((("*"[^/])?)|[^*])*"*/" {
    fprintf(yyout,"MULTI LINE COMMENT at Line %d\n", yylineno);
  }
```

3.4 String

We designed regexps in `lex.l` to match strings, supporting ESC and C-style multi-line strings.

The parser will consider them similar to other constant types and can be reduced into `Exp`.

```
VALID_STRING_LINE ([^\\\"\n]*(\\.))*[^\\\"\n]*
STRING \"({VALID_STRING_LINE}\\{new_line})*{VALID_STRING_LINE}\"
```

4 Explanations for Test Cases

4.1 Basic tests

We designed 5 test cases for basic features under `./test` foder.

1. `test_12111624_1.spl`

   This piece of code has no syntax error. Basic variable defintions and numerical calculations.

2. `test_12111624_2.spl`

   This piece of code has no syntax error. Do nothing in `if` and `while`, with a useless semicolon.

3. `test_12111624_3.spl`

   This piece of code contains 2 TYPE B errors: `else` without an `if`.

4. `test_12111624_4.spl`

   This piece of code contains 3 TYPE B errors: unclosed `{}`,`[]`,`()`.

5. `test_12111624_5.spl`

   This piece of code contains 3 TYPE A errors: `1test_r05`, `@`, `998244353.2147483647.19260817`.

4.2 Extra tests

We designed one correct case and one error case for each bonus feature under `./test-ex` foder, 8 test cases in total.

1. `test_1.spl`

   Test for single-line and multi-line comments.

2. `test_2.spl`

   Test for multi-line comments with only `/*`.

3. `test_3.spl`

   Test for `for(;;)` and `for(:)`.

4. `test_4.spl`

   Test for missing closing curly bracket `for(;;){`

5. `test_5.spl`

   Test for `#include <FILENAME>`.

6. `test_6.spl`

   Test for `#include` error.

7. `test_7.spl`

   Test for `string` and `multiple-line-string`.

8. `test_8.spl`

   Test for string error due to wrong use of escape character.