Lexical Analysis & Syntax Analysis

1 Introduction

This report aims to introduce the compiler development work we have completed in the CS323 course project. Our task was to write a compiler for a toy programming language called SPL. In the first phase, we utilized two powerful open-source tools, Flex and Bison, to implement the parser and wrote it using the C programming language. Our parser performs lexical analysis and syntax analysis on SPL source code, identifying tokens and checking the code structure.

This report will first introduce the characteristics and requirements of the SPL language, followed by a detailed description of the design and implementation of our parser. Finally, we will discuss the challenges and issues we encountered in the first phase and provide suggestions for future improvements.

2 Deveplop Tools

- GCC version 11.4.0
- GNU Make version 4.3
- GNU Flex version 2.6.4
- GNU Bison version 3.8.2

3 SPL Grammar

3.1 Lexical

Here is all of the valid tokens in SPL:

- Keywords: int char float **string** struct if else while return
- Brackets: Curly brackets { }, Square brackets [ ], Round brackets ( ).
- Operators and Separators: = < <= > >= != == + - * / && || ! . , ;
- INT, FLOAT, ID: the implementation and requirements are consistent with the requirements.
- CHAR: We have two types of character literals, decimal (base 10) and hexadecimal (base 16) forms. A hexadecimal integer lexeme always begin with "\x", followed by a sequence of hex-digits (0-9, a-f),
- **String: multiple character contained in a pair of double-quotes. when there are nested double-quotation mark in the lexeme, user can use \" to mark.**

3.2 Syntax

This part is designed as required.

3.3 Comments

We have implemented a commenting feature in our system, which includes both single-line comments using "//" and multi-line comments using "/* */". This feature allows developers to add comments within the code to provide explanations, notes, or documentation.

4 Implementation And Result

4.1 Lexical

This section focuses on the implementation of int, float, char, and id. We achieved their recognition, validation, and other functionalities by setting up corresponding regular expressions and creating nodes.

```
// int
0(x|X){hex_digit}+ {
    if (yyleng < 35)// handle hexadecimal int
    else // length exceed limit, report error
}
{digit}+ {
    if (yyleng < 33) // handle decimal int
    else // length exceed limit, report error
}

// char
'[^']' { // handle decimal char }
'\\(x|X){hex_digit}+' { // handle hexadecimal char}

// float
{digit}+{DOT}{digit}+ { // handle float }

// ID
{letter_}({letter_}|{digit})* { // valid ID, handle ID }
({letter_}|{digit})+{letter_}+ { // invalid ID, report error. }

// string
\"[^\"]*\" {
    if(yytext[yyleng-2] == '\\') {
        yyless(yyleng-1);
        yymore();
    } else // handle string
}
```

4.2 Syntax

We have designed a tree node structure to store the parse tree information in a format that is readily printable. The line number and respective token values from the lex file are returned and stored in the corresponding tree node. Nonterminals act as parent nodes for both terminal and nonterminal children. Once the entire code has been parsed, we print the tree in a top-down manner. Additionally, a boolean flag named "error" is implemented to switch to a true value if any error occurs, resulting in the tree not being printed.

```
// tree_node.h
#ifndef TREE_NODE_H
#define TREE_NODE_H
#include <stdbool.h>
typedef struct TreeNode {
    char* type;
    char* value;
    int line;
    bool empty;
    struct TreeNode** children;
    int numChildren;
} TreeNode;
#endif

// example:
Args : Exp COMMA Args {
    $$ = createNode("Args", "", $1->line, 3, $1,
                createNode("COMMA", "", 0, 0), convertNull($3));
}
| Exp {
    $$ = createNode("Args", "", $1->line, 1, $1);
}
;
```

4.3 Comments

In the implementation of comments, we recognize two types of comments: single-line and multi-line. If "//" is recognized, the program will advance to the end of the line and increase the line number. If "/*" is recognized, the program will enter the comment mode, where any character except newline and "*/" will be ignored. When encountering a newline character "\n", the line number will be increased. The program will exit the comment mode when "*/" is encountered.

```
%x COMMENT
%%
"//" { char c; while((c=input()) != '\n'); line++;}
"/*" { BEGIN(COMMENT); }
<COMMENT>{
    "*/" { BEGIN(INITIAL); }
    \n {line++;}
    . {}
}
%%
```

4.4 Conflict reducing

In this section, we have avoided all conflicts by assigning priorities and associativity to various terminals based on their own priorities, as well as explicitly defining priorities for non-terminals.

```
%nonassoc<node> LOWER
%nonassoc<str_line.line> ELSE
%nonassoc<str_line.line> ASSIGN
%left<str_line.line> OR
%left<str_line.line> AND
%nonassoc<str_line.line> LT LE GT GE NE EQ
%left<str_line.line> PLUS MINUS
%left<str_line.line> MUL DIV
%left<str_line.line> NOT
%left<str_line.line> LP RP LB RB DOT

Stmt: ...
|IF LP Exp RP Stmt %prec LOWER {
    // something...
}
| IF LP Exp RP Stmt ELSE Stmt {
    // something...
}
```

5 Test

Open the project and run:

make clean
make splc
bin/splc "testcase_address".spl

the result will be in "testcase_address".out

6 Conclusion

In the first phase of our CS323 course project, we successfully implemented a parser for the SPL (SUSTech Programming Language) programming language using Flex and Bison. Our SPL parser was able to analyze the source code and identify tokens and their relationships according to the specified grammar rules. This achievement sets the stage for the successful completion of the rest of the compiler.

Throughout this process, we encountered numerous challenges, including but not limited to our unfamiliarity with the C language, lack of understanding certain default settings in Flex and Bison, which leading to confusing bugs, and so on. We faced some difficulties and experienced moments of frustration, but ultimately, we gained a lot precious experience, as well as a better partnership of our group.

Overall, this phase provided us with practical experience in lexical and syntax analysis, as well as an understanding of how to automate these processes using powerful tools like Flex and Bison. We are now well-equipped to proceed with the next stages of the project and continue building a fully functional compiler for the SPL language.