Lexical Analysis & Syntax Analysis

1 Overview

In the CS323 course, we will provide a series of tutorials to help you write a compiler for a toy programming language called SPL, the abbreviation for SUSTech Programming Language. SPL is a C-like programming language that removes most advanced features in C standard, such as macros or pointers. It is strongly typed, with several primitive types and structure type (as we borrowed from C), and supports C-like control flow such as if-statement and while-statement. Also, it supports basic user interaction by providing two built-in function `read` and `write`, which makes it Turing-complete!

Following the tutorials, you will learn how to build your own compiler from scratch. Our compiler compiles a SPL program into MIPS32 assembly. You will be guided to realize it through lexical and syntax analysis, semantics checking, intermediate code and target code generation. At the end of this course, your compiler will be able to translate a SPL program into MIPS32 code, which can be run on a **REAL** MIPS simulator.

You are suggested to configure the development environment by yourself. Detailed information can be found in Section 2.

In the first phase, you are going to implement a SPL parser. Your parser will perform lexical analysis and syntax analysis on the SPL source code. You will use two powerful open-source tools, Flex[1] and Bison[2], to realize your parser. you will implement it by C programming language. In modern compiler design and implementation, lexical analysis and syntax analysis can be totally automated (so you don't need to implement your NFA/DFA, or parsing algorithms), typical tools are Lex/Flex for generating lexical analyzer, and Yacc/Bison for generating parser. Though you can complete phase 1 without too much theoretical knowledge, it doesn't mean the theory is not important. You will learn how to recognize tokens using regular expressions and how to check code structures using context-free grammars in the lecture. As for this phase, what you need to do is just specifying the regular expressions and grammar rules.

One thing to note is that, you will finish the subsequent parts of your compiler on top of this work, thus it is important to keep your code maintainable and extensible.

Lastly, for our course project, you are highly recommended to design your own programming language, choose an appropriate combination of intermediate and target language, and implement a compiler for it. This series of tutorials based on the SPL language will be helpful to you. Besides the Flex/Bison tools, you are welcome to use other tools (e.g., Antler[3]) to help your build your compiler.

[1] https://github.com/westes/flex
[2] https://www.gnu.org/software/bison/
[3] https://www.antlr.org/

2 Development Environment

To set up your own development environment, we list the necessary dependencies as follow (we have tested the suggested versions and newer versions may also work):

- GCC version 7.4.0
- GNU Make version 4.1
- GNU Flex version 2.6.4
- GNU Bison version 3.0.4
- Python version 3.6.8
- urwid (Python module) 2.0.1
- Spim version 8.0

3 `Flex` for Lexical Analysis

Flex is a fast lexical analyzer (or *lexer*) generator. You should specify the token patterns to match and actions to apply for each token. Flex takes your specification and generates a combined NFA that recognizes all your patterns, then converts it to an equivalent DFA, minimizes the automaton as much as possible, finally generates C code that implements the lexer. Flex is similar to its predecessor, Lex, designed by Lesk and Schmidt. While we will use Flex to build our compiler, almost all features we use are present in Lex.

This section is designed to give you a quick introduction to the Flex tool. We hope it serves as a useful reference for the phase 1. To learn more about Flex, run `info flex` in the command line, or read the documentation at `http://dinosaur.compilertools.net/flex/`.

3.1 Coding with Flex

Flex is designed for use with C code and generates a lexer/scanner written in C. The lexer is specified using regular expressions for *patterns* and C code for the *actions*. The lexical specification files are identified by a `.l` extension. You invoke `flex` on a `.l` file and it creates `lex.yy.c`, a source file containing C code that implements a finite automaton encoding all your rules and corresponding actions you specified. The file provides an `extern` function `yylex()` that will read the from input stream pointed by `yyin`. You compile that C source file normally, link with the Flex library (by GCC option `-lfl`), then you have built a lexer!

```
%{
Declarations
%}
```

```
Definitions
%%
Rules
%%
User subroutines
```

The optional `Declarations` and `User subroutines` sections are used for ordinary C code that you want copied verbatim to the generated C file. `Declarations` are copied to the top of the file, and `user subroutines` to the bottom. The optional `Definitions` section is where you specify options for the scanner and can set up definitions to give names to regexes as a simple substitution mechanism that allows for more readable code in the `Rules` section that follows. The *required* `Rules` section is where you specify the patterns that identify tokens and corresponding actions to perform upon recognizing each token. Each section is separated by two `%` marks. If the code contains only the required `Rules` section, it should also begin with a line `%%`.

To see how Flex works, we have a simple application that includes all aforementioned sections in a `.l` source file. For those optional sections, Flex has default routine to handle with their absence, which will be introduced later.

Listing 1: A `wc` application in Flex

```
%{
    // just let you know you have macros!
    #define EXIT_OK 0
    #define EXIT_FAIL 1
    // and global variables
    int chars = 0;
    int words = 0;
    int lines = 0;
%}
letter [a-zA-Z]
%%
{letter}+ { words++; chars+=strlen(yytext); }
\n { chars++; lines++; }
<<EOF>> { if(yyleng > 1){ lines++; } return 0; }
. { chars++; }
%%
int main(int argc, char **argv){
    char *file_path;
    if(argc < 2){
        fprintf(stderr, "Usage: %s <file_path>\n", argv[0]);
        return EXIT_FAIL;
    }
    else if(argc == 2){
        file_path = argv[1];
        if(!(yyin = fopen(file_path, "r"))){
            perror(argv[1]);
            return EXIT_FAIL;
        }
        yylex();
        printf("%-8d%-8d%-8d%s\n", lines, words, chars, file_path);
```

```
            return EXIT_OK;
        }
        else{
            fputs("Too much arguments!\n", stderr);
            return EXIT_FAIL;
        }
    }
```

The program above works like a `wc` utility presented in most Unix systems. In the `Declarations` section, we define two macros `EXIT_OK` and `EXIT_FAIL`, and three global variables that store the number of characters, words and lines, respectively. Just like coding with C, you can also include libraries, declare external variables in this section. Then the `Definitions` section follows. The general format of this section is:

`name definition`

which means you *name* a regex specified by the *definition*. You can define multiple definitions here. The `Definitions` section is optional, since you can always write the full regexes in the `Rules` section. A single rule is defined as:

`pattern {action}`

As the lexer reads characters from the input stream, it will gather them until it forms the longest possible match for any of the available patterns. If two or more patterns match an equally long sequence, the pattern listed first in the source code is applied. The *action* is written in C, it depends on what procedure you are trying to do with each token. For a lexer designed to be used by a compiler, the action will usually record the token attributes and return a code that identifies the token type.

In code listing 1, we define the name `letter`, which corresponds to the alphabetic characters, then we specify four action rules that reference global variables defined by both Flex library and ourselves. The global variables `yytext` and `yyleng` store the matched lexeme and its length. Action for each pattern is intuitive. In Section 3.2, we will show more available patterns supported by Flex. The third pattern is a special pattern provided by Flex, it is matched when the input stream reaches the end, it will return 0 by default. In the customized rule, 0 should also be returned to signal the end-of-file, otherwise the lexer will never stop scanning.

We defined our own `main` function at the bottom. By default, Flex provides a simple `main` that will repeatedly calls `yylex` until it reaches EOF. You should compile and link the lexer into your project and use your own `main` to control the behavior whenever tokens are matched. In the main routine, we reference the variable `yyin` and invoke the library function `yylex`. The former is a pointer pointing to the input file descriptor and is `stdin` by default; the latter is the entry point to our lexing procedure. More library-defined global variables and functions will be introduced later in Section 3.3.

3.2 Patterns in Flex

We show you some useful patterns supported by Flex in Table 1, while not yet complete, they are sufficient for your compiler construction. For more details about the supported regex

Table 1: Patterns supported by Flex (not complete)

Pattern	**Regex**	**Description**
Character classes	`[0-9]`	This means alternation of the characters in the range listed (in this case: `0|1|2|3|4|5|6|7|8|9`). More than one range may be specified, e.g. `[0-9A-Za-z]` as well as specifying individual characters, as with `[aeiou0-9]`.
Character exclusion	`^`	The first character in a character class may be `^` to indicate the complement of the set of characters specified. For example, `[^0-9]` matches any non-digit character.
Arbitrary character	`.`	Matches any single character **except newline**.
Selection	`x|y`	Either `x` or `y` can be matched.
Single repetition	`x?`	0 or 1 occurrence of `x`.
Nonzero repetition	`x+`	`x` repeated one or more times; equivalent to `xx*`.
Specified repetition	`x{n,m}`	x repeated between n and m times.
Beginning of line	`^x`	Match `x` at beginning of line only.
End of line	`x$`	Match `x` at end of line only.
Context-sensitivity	`ab/cd`	Match `ab` but only when followed by `cd`. The lookahead characters are left in the input stream to be read for the next token.
Literal strings	`"x"`	This means `x` even if `x` would normally have special meaning. Thus, `"x*"` may be used to match `x` followed by an asterisk. You can turn off the special meaning of just one character by preceding it with a backslash, .e.g., `\.` matches exactly the period character and nothing more.
Definitions	`{name}`	Replace with the earlier defined pattern called `name`. This kind of substitution allows you to reuse pattern pieces and define more readable patterns.

patterns, please refer to the Flex manual.

3.3 More Flex Features

In this section, we will introduce more features supported by Flex, which can reduce your workload during coding. However, they are totally optional since you can still complete your lexer without these advanced features.

1. **The `yylineno` option**

A compiler often needs to record information of line number of the source file, in case of error reporting, debugging, etc. To support this functionality, you can maintain a global variable `lineno` with initial value 1, and increment it when a new line character is encountered, i.e., `lineno++` for pattern `"\n"`.

However, Flex has already provided built-in mechanism for this, by a global variable `yylineno` which will automatically increase by 1 after scanning each line. To use this feature, you should add a single line in the `Definitions` section:

```
%option yylineno
```

2. **The library function `input` and `unput`**

Flex library function `input()` takes no input, which will read a single character from the input buffer and return it. By this function, you are able to realize some actions without specifying the concrete regex. For example, you can discard all characters behind `"//"` in the line by the following code:

```
"//" { char c; while((c=input()) != '\n'); }
```

Inversely, the function `unput(char c)` puts `c` back into the input buffer. You can realize multiple-lines comment with this function. Note that pushing characters back should be done reversely.

3. **The library function `yyless` and `ymore`**

`yyless(int n)` returns the `yyleng-n` characters in the postfix of current lexeme back to the input buffer, where they will be rescanned when the lexer looks for the next match. `yytext` and `yyleng` are adjusted appropriately (e.g., `yyleng` will now be equal to `n`).

`yymore()` tells the lexer that the next time it matches a rule, the corresponding token should be appended onto the current value of `yytext` rather than replacing it.

If you want to support string literal, you should consider the case when there are nested double-quotation mark in the lexeme, for example, `"And God said, \"Let there be light,\" and there was light."` By cooperating `yyless` and `yymore`, your lexer is able to recognize string literals:

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

3.4 Writing Your Lexer

To realize your lexer, you should firstly (and carefully) read the token specification of SPL in Appendix A and the detailed explanation. You should write regular expressions for the first four tokens, while the remaining part of the specification is rather simple. In this stage, your job is only to recognize each token, hence you can simply print out the tokens in their corresponding actions:

```
"TYPE" { printf("TYPE %s\n", yytext); }
```

To report lexical error, you can add a fallback rule:

```
. { printf("Error type A at Line %d: Unknown characters \'%s\'\ n",
          yylineno, yytext); }
```

Here is a sample program of SPL:

Listing 2: A sample SPL source code

```
int test_spl()
{
    int i = 0, j = 1;
    float i = 1;
}
```

For this source code, your lexer should print:

```
TYPE int
ID test_spl
LP
RP
LC
TYPE int
ID i
ASSIGN
INT 0
COMMA
ID j
ASSIGN
INT 1
SEMI
TYPE float
ID i
ASSIGN
INT 1
SEMI
RC
```

4 `Bison` for Syntax Analysis

Bison is a parser generator. You provide the input of a grammar specification and Bison will generate an LALR(1) parser to recognize sentences in that grammar. The name is a great example of a CS in-joke. Bison is an upgraded version of the older tool Yacc ("Yet Another Compiler Compiler"). It is probably the most common LALR tool. In our course, we will use the updated version Bison, a close relative of the yak, but all features we use are also present in the original tool. So this handout serves as a brief overview of both. For those who really want to dig deep and learn everything about parser generators, please refer to Bison's official manual: `https://www.gnu.org/software/bison/manual/bison.html`.

Bison generates a parser, which accepts the input token stream from Flex, to recognize code in the specified context-free grammar (CFG). By default, Bison generates a LALR(1) parser. Note that the programmer is always shielded from the parsing algorithm applied by the compiler. For example, JTB[4] by Java generates an LL(1) parser for the JavaCC. As a compiler designer, what you are going to do is writing the grammar/syntax specification, and the action for each production. In theory, the output of a parser is a parse tree, though we are able to go beyond, i.e., analyzing the code semantics with Bison.

4.1 Interacting Flex and Bison

Bison source code has a similar structure as Flex source code, which also contains optional `Declarations`, `Definitions` and `User routines` sections.

```
%{
Declarations
%}
Definitions
%%
Productions
%%
User subroutines
```

In Bison, just as you utilize Flex, you can associate an action with each pattern (this time a production), which allows you to do whatever processing as you reduce using that production. Unlike in Flex, the optional `Definitions` section is where we configure various parser features such as defining token codes, establishing operator precedence and associativity, and setting up the global variables used to communicate between the lexer and parser.

To see how Bison compiles a CFG into an executable, we provide example Bison code in Listing 3. It implements a simple calculator with conventional precedence for each operation:

Listing 3: The Bison source code `syntax.y` for `calc`

[4] `http://compilers.cs.ucla.edu/jtb/`

```
 1  %{
 2      #include"lex.yy.c"
 3      void yyerror(const char*);
 4  %}
 5  %token INT
 6  %token ADD SUB MUL DIV
 7  %%
 8  Calc: /* empty */
 9      | Exp { printf("= %d\n", $1); }
10      ;
11  Exp: Factor
12      | Exp ADD Factor { $$ = $1 + $3; }
13      | Exp SUB Factor { $$ = $1 - $3; }
14      ;
15  Factor: Term
16      | Factor MUL Term { $$ = $1 * $3; }
17      | Factor DIV Term { $$ = $1 / $3; }
18      ;
19  Term: INT
20      ;
21  %%
22  void yyerror(const char *s){
23      fprintf(stderr, "%s\n", s);
24  }
25  int main(){
26      yyparse();
27  }
```

We define our `main` function at the end of the source file. We invoke the Bison function `yyparse`, in which the token stream will be read via `yylex`. The routine `yyerror` is called when parser encounters an error, in which the next token cannot be shifted during parsing. Though `yyerror` is invoked every time an error occurs, it should be defined by ourselves. You should be able to report more than general syntax error messages in the `yyerror` function.

We include the Flex-generated `lex.yy.c` file at the `Declarations` section, since we will apply `yylex` defined in it. It's worth mentioning that, the `Declarations` section will be copied verbatim into the generated `C` file, rather than statically linked during Bison compilation, hence we are able to compile our `.y` file successfully with the absence of `lex.yy.c` file. Lines 5–6 define tokens of the language. In this case, they are number literals and arithmetic operators. These tokens can be returned by Flex actions.

All defined tokens are terminals in the language, where the nonterminals are specified in the `Productions` section. The first nonterminal presented is assumed to identify the start symbol for the grammar. For each rule, a colon is used in place of the arrow, a vertical bar separates the various productions, and a semicolon terminates the rule. The action for each production is enclosed by braces. If no rules are assigned, Bison will insert a default action `{ $$ = $1 }`. The variables whose name starts with the dollar sign (`` `$' ``) are associated with each variable's attribute value at each production and the order increases from left to right. As an example, in Listing 3, Line 12 evaluates the result of add operator. The `` `$$' `` is the attribute of the left-side of the production, i.e., `Exp`, while `` `$1' `` and `` `$3' `` correspond to the attributes of right-side nonterminals `Exp` and `Factor`.

Since we read the token stream from lexer-defined functions, we should update Flex source code accordingly:

Listing 4: The Flex source code `lex.l` for `calc`

```
%{
    #include"syntax.tab.h"
%}
%%
[0-9]+ { yylval = atoi(yytext); return INT; }
"+" { return ADD; }
"-" { return SUB; }
"*" { return MUL; }
"/" { return DIV; }
```

In each action, we return the tokens defined in `syntax.tab.h`, which is generated from Bison source code. `yylval` is a predefined Flex variable, which indicates the attribute of the current token, while the return value tells the parser what kind of token is scanned at the moment.

In order to compile the calculator, we should firstly compile the Bison source code into C code, and split it into header file and implementation file (by Bison "`-d`" option):

```
bison -d syntax.y
```

This command will produce two files, `syntax.tab.h` and `syntax.tab.c`. Then we can compile the Flex code:

```
flex lex.l
```

Next we put them together and compile the source files with GCC:

```
gcc syntax.tab.c -lfl -ly -o calc.out
```

The options `-lfl` and `-ly` tell GCC to include Flex and Bison/Yacc library, respectively. After that, we have an executable, which accepts a single arithmetic expression from standard input and outputs its result to the console. You can validate the calculator by feeding simple test cases:

```
  echo "1 + 1" | ./calc.out
> = 2
  echo "10 - 2*3" | ./calc.out
> = 4
  echo "92+1c" | ./calc.out
> syntax error
```

4.2 More on Bison

1. **Token/nonterminal attributes**

In the above, we mentioned that, the attribute of the symbol is assigned to the global variable `yylval`, but we did not clarify its data type. This variable is declared as of type `YYSTYPE`, which has a default value `int`. That's why we are able to assign a integer value to `yylval` in the aforementioned Flex source code. However, in our parser, the attribute of each nonterminal/token should be abstracted into a syntax tree node. How can we implement it?

One way to do so is to explicitly redefine the `YYSTYPE` macro. In other words, you can insert the following line in the Bison `Declarations`:

```
#define YYSTYPE float
```

to ensure each production results in a floating-point value. To support multi-typed attribute, you may consider `union/struct` type.

However, there are more flexible approaches for this goal. Bison provides the `%union` directive to indicate the token's data type, which goes to the `Definitions` section. For example, the following `union` type will be copied as the value of `YYSTYPE`:

```
%union{
    int int_value;
    float float_value;
    char *string_value;
}
```

To indicate the token type, we should modify the token definition. In the `Definitions` section, by preceding the token name with the union field name enclosed in a pair of angle brackets, we are able to assign type to a particular token. The same usage applies to nonterminal symbols, which should be specified by the `%type` directive.

```
%token <int_value> INT
%token ADD SUB MUL DIV
%type <float_value> Exp Factor
```

As an example, the code above assigns an integer value to the `INT` token, and a floating-point value to `Exp` and `Factor` nonterminals. The arithmetic operators are defined without data type, since their attributes are not significant during expression parsing.

2. **Symbol position**

At the beginning of this section, we have shown that we can access the attribute of each symbol (token or nonterminal) via `$n`, where `n = $` refers to the left-side nonterminal of production, while numerical `n` refers to the `n`-th symbol at the right side.

Similarly, we are able to access the location information of each symbol with `@n` notation. These two notations are totally in parallel, i.e.,

- `$n` refers to a `YYSTYPE` variable `yylval`, which is the attribute of the symbol
- `@n` refers to a `YYLTYPE` variable `yylloc`, which is the position of the symbol

The default definition of `YYLTYPE` is a structure containing the first line/column and the last line/column of the grammar symbol. The information provided is already sufficient. However, if you directly read the information via `@n` notation, you will find that the location value never gets updated! The reason is that, the parser only reduces the token stream, while the position information is gathered by the lexer. So, in order to update the location, you have to manage it in the Flex code.

Firstly, you should insert the code below at the header sections in the `.l` file:

```
%{
    /* library inclusions */
    int yycolno = 1;
    #define YY_USER_ACTION \
        yylloc.first_line = yylineno; \
        yylloc.first_column = yycolno; \
        yylloc.last_line = yylineno; \
        yylloc.last_column = yycolno + yyleng; \
        yycolno += yyleng;
%}
```

The variable `yycolno` is managed by yourself, which should be reset every time a new line occurs:

```
"\n" { yycolno = 1; }
```

The macro `YY_USER_ACTION` is executed at each token recognized by Flex, which is an empty macro by default.

3. **Conflict resolution**

The grammar can be ambiguous, and Bison will not throw exception under such a situation, but will record all potential conflicts in the table, and handle conflicts in such ways:

- For a shift/reduce conflict, always choosing shift
- For a reduce/reduce conflict, reducing with the rule declared first in the `.y` file

These strategies may change the accepted language, which is undesirable. Even if everything happens to work out, it is not recommended to adopt this default mechanism. You should resolve conflicts by specifying the precedence and associativity for the operators. The precedence and associativity for SPL operators are enumerated in Appendix C.2.

Consider our calculator example. If we modify the grammar to make it ambiguous:

```
Exp: INT
    | Exp ADD Exp
```

```
        | Exp SUB Exp
        | Exp MUL Exp
        | Exp DIV Exp
        ;
```

When you compile the grammar, Bison will report:

`syntax.y: warning: 16 shift/reduce conflicts [-Wconflicts-sr]`

The parser generated from this grammar may then exhibit strange behavior:

```
  echo "3 * 4 + 5" | ./calc.out
> = 27
  echo "5 + 3 * 4" | ./calc.out
> = 17
```

To understand the reason, consider why shift/reduce conflict occurs. For the first expression, when token stream `3 * 4` is on the stack, followed by the tokens `+ 5`, the parser can choose either reducing with `Exp -> Exp MUL Exp` or shifting the next token. For such a shift/reduce conflict, Bison will always shift, hence the parser shifts `+ 5` and reduces with `Exp -> Exp ADD Exp`. To resolve the conflict, we should explicitly assign precedence and associativity to each operator. See the following token definition code:

```
%left ADD SUB
%left MUL DIV
```

The code states the associativity for arithmetic operators, i.e., all of them associate from left to right. Similarly, you can apply `%right` directive to indicate a right-to-left operator, and `%nonassoc` for a non-associative operator. The code also assigns precedence for the operators by ordering the token definition: the lower-precedence operator appears earlier, so `MUL` has a higher precedence than `ADD` and `SUB`.

Assigning precedence and associativity resolves the conflicts. For a shift/reduce conflict, if the precedence of the token to be shifted is higher than that of the rule to reduce, it chooses to shift and vice versa. The precedence of a rule is determined by the precedence of the rightmost terminal on the right-hand side. So if a `5 + 3` is on the stack and `*` is coming up, the `*` has a higher precedence than the `5 + 3`, so it shifts. If `3 * 4` is on the stack and `+` is coming up, it reduces. If `5 + 3` is on the stack and `+` is coming up, the associativity breaks the tie: a left-to-right associativity will reduce the rule and then go on; a right-to-left associativity will shift and postpone the reduction.

The associativity of the production can also be explicitly set with the `%prec` directive. When `%prec` is placed at the end of a production with a terminal as argument, it explicitly sets the precedence of the production to the same precedence as that terminal. This is useful where the production's right side has no terminals, or when you want to explicitly assign precedence to the production. This feature may help you to eliminate the shift/reduce conflict warning for a *dangling-else ambiguity* (it is present in our SPL specification), even if Bison's default strategy for this conflict is consistent with the C language standard.

4. **Error recovery**

Bison supports error recovery by providing a special `error` token. When the Bison-generated parser encounters an error, it triggers the following error recovery routine:

1. Invoke `yyerror` reporting function (which you may overwrite)
2. Pop out all un-reduced tokens, until the `error` can be shifted
3. Shift `error`, then drop tokens until one can be pushed after `error` (re-synchronization)
4. If three symbols can be successfully shifted, continue parsing, otherwise, back to step 2

To recover from an error state, you should place the `error` in a correct context. For example, the following recovery patterns help the parser recover from programs that have missing semicolons, right curly brackets or right parentheses.

```
Stmt: Exp error
CompSt: LC DefList StmtList error
Exp: ID LP Args error
```

However, where to put the `error` token is more of an art than a science: putting error at fairly high levels tends to protect you from all sorts of errors, while putting error at a lower level can help minimize the number of tokens to be discarded. For compiler construction, one reasonable strategy is to put it before ending the statement, and use punctuation as the synchronizing tokens. Another practice is to add incorrect productions into the syntax specification, which allows your parser to recognize these illegal forms.

Error recovery is largely a trial-and-error process, you can experiment with real-world compilers to see how good they are at recognizing errors (they are expected to hit this one perfectly), reporting errors clearly (less perfect), and gracefully recovering (far from perfect).

4.3 Parser Implementation

Realizing a parser is more complicated than a lexer. You have to carefully read and understand each production to handle potential syntax conflicts/errors as much as possible.

We highly recommend you to build your parser without any action for all productions at the beginning. For a syntactically correct SPL program, this parser should run smoothly and exit without any output or error message; as for a program with syntax errors, your parser at this stage will only output `"syntax error"`. Then you can handle general issues for parser, such as conflict resolution and error recovery.

To ensure that your parser can be compiled smoothly and run correctly, we provide you some hints for debugging your Bison programs:

- Compile your `.y` file with `-v` option, which generates a `syntax.output` text file that illustrates the LALR automaton, about its states, transitions, etc.
- Compile `.y` with `-t`, and add `yydebug = 1;` before invoking `yyparse`, which enables your parser with verbose output.

After that, you should construct a parse tree in the action of each production. As we've done in Listing 3, for a nonterminal symbol, you can assign a tree node to attribute `$$` and insert its children nodes one-by-one by some sort of `insert($$, $n)` (alternatively, consider variadic function for node constructor). For a terminal symbol, assign it with tree node via `yylval` in the Flex code, the usage of which is already shown in Listing 4.

Printing out the parse tree can be implemented as a pre-order traversal from the root node. In a object-oriented design paradigm, this method should be a member of type node, rather than the routine completed by the `main` function.

5 Requirements

5.1 Parser Requirements

You are required to implement a parser that accepts a single command line argument (the SPL file path), and outputs the parse tree for a syntactically valid SPL program, or the message reporting all existing lexical/syntax errors in the code. You should compile your parser as the `splc` target in `make`, and move the executable to the `bin` directory.

For example, the `Makefile` is placed under the project root directory, then we make the `splc` target by:

```
make splc
```

which generates the parser executable. Then we run the parser by:

```
bin/splc test/test_1_r01.spl
```

the parser should output the corresponding result specified in the text file `test/test_1_r01.out`, and we will verify the output by the `diff` utility.

5.1.1 Basic features

Your parser should be able to recognize the following errors:

- Lexical error (error type A) when there are undefined characters or tokens in the SPL program, or identifiers starting with digits.
- Syntax error (error type B) when the program has an illegal structure, such as missing closing symbol. Please find as many syntax errors as possible.

For a syntactically correct SPL program, your parser should print out its parse tree. Consider the following SPL code:

Listing 5: SPL test case 1

```
int test_1_r01(int a, int b)
{
   c = 'c';
   if (a > b)
   {
      return a;
   }
   else
   {
      return b;
   }
}
```

This program is free from syntax errors, while there is a semantic error: the local variable `c` is referenced without a definition, which is not allowed in C standard. Your parser should print its corresponding parse tree as follows (a caliper is helpful):

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: test_1_r01
        LP
        VarList (1)
          ParamDec (1)
            Specifier (1)
              TYPE: int
            VarDec (1)
              ID: a
          COMMA
          VarList (1)
            ParamDec (1)
              Specifier (1)
                TYPE: int
              VarDec (1)
                ID: b
        RP
      CompSt (2)
        LC
        StmtList (3)
          Stmt (3)
            Exp (3)
              Exp (3)
```

```
                ID: c
              ASSIGN
              Exp (3)
                CHAR: 'c'
            SEMI
          StmtList (4)
            Stmt (4)
              IF
              LP
              Exp (4)
                Exp (4)
                  ID: a
                GT
                Exp (4)
                  ID: b
              RP
              Stmt (5)
                CompSt (5)
                  LC
                  StmtList (6)
                    Stmt (6)
                      RETURN
                      Exp (6)
                        ID: a
                      SEMI
                  RC
              ELSE
              Stmt (9)
                CompSt (9)
                  LC
                  StmtList (10)
                    Stmt (10)
                      RETURN
                      Exp (10)
                        ID: b
                      SEMI
                  RC
        RC
```

Listing 6: SPL test case 2

```
int test_1_r03()
{
  int i = 0, j = 1;
  float i = $;
  if(i < 9.0) {
    return 1
  }
```

```
        return @;
    }
```

The program in Listing 6 contains two lexical errors and one syntax error, so your parser should report all of them:

```
Error type A at Line 4: Mysterious lexeme $
Error type B at Line 7: Missing semicolon ';'
Error type A at Line 8: Mysterious lexeme @
```

Note that the syntax error is reported at Line 7, while, intuitively, it appears to be at the end of Line 6. We do not require the exact position for a syntax error. In such a case, either Line 6 or Line 7 is fine.

Additionally, you should implement these capabilities:

- Supporting hexadecimal representation of integers such as `0x12`. You should be able to detect illegal form of hex-int, like `0x5gg`, and report lexical errors.
- Supporting hex-form characters, such as `\x90`, also, you need to detect its illegal form like `\xt0` and report lexical errors.

5.1.2 Extended features

You are encouraged to extend SPL language specification and implement other features that are not mentioned before. To show your parser's ability, you should provide your own testcases in the submitted files, and clearly introduce them in the report. The grading of this part will be based on the difficulty of the additionally supported features. To start, you may consider the following features adopted by most general-purpose programming languages:

- single- and/or multi-line comment (C.3)
- macro **preprocessor**
- file inclusion
- `for` statements
- ...

6 Resources

Here we list some links that you may find useful information during this course.

- C language tutorial: `http://www.stat.cmu.edu/~brian/cprog.html`
- Simple Makefile tutorial: `http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/`
- Compiler tools (Flex & Bison) introduction: `http://dinosaur.compilertools.net/`

- Flex & Bison introduction: `https://aquamentus.com/flex_bison.html`
- Stanford University CS143 Compilers: `https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/`
- GNU C Compiler Internals/GNU C Compiler Architecture: `https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture`
- Buffer Overflow Exploit - Dhaval Kapil:
  `https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/`
- Assemblers, Linkers, and the SPIM Simulator: `http://pages.cs.wisc.edu/~larus/HP_AppA.pdf`
- Kempe's graph-coloring algorithm `https://www.cs.princeton.edu/~appel/Color.pdf`

Appendix

A Token Specification

The following token specification defines the valid token in SPL. You can easily make a lexer with Flex by following the specification. However, we do not specify the patterns of the first four tokens: `INT`, `FLOAT`, `CHAR` and `ID`, it is your job to carefully design and write their regular expressions.

```
INT     ->  /* integer in 32-bits (decimal or hexadecimal) */
FLOAT   ->  /* floating point number (only dot-form) */
CHAR    ->  /* single character (printable or hex-form) */
ID      ->  /* valid identifier */
TYPE    ->  int | float | char
STRUCT  ->  struct
IF      ->  if
ELSE    ->  else
WHILE   ->  while
RETURN  ->  return
DOT     ->  .
SEMI    ->  ;
COMMA   ->  ,
ASSIGN  ->  =
LT      ->  <
LE      ->  <=
GT      ->  >
GE      ->  >=
NE      ->  !=
EQ      ->  ==
PLUS    ->  +
MINUS   ->  -
MUL     ->  *
DIV     ->  /
AND     ->  &&
OR      ->  ||
NOT     ->  !
LP      ->  (
RP      ->  )
LB      ->  [
RB      ->  ]
LC      ->  {
RC      ->  }
```

B Grammar Specification[5]

```
/* high-level definition */
Program -> ExtDefList
ExtDefList -> ExtDef ExtDefList
    | $
ExtDef -> Specifier ExtDecList SEMI
    | Specifier SEMI
    | Specifier FunDec CompSt
ExtDecList -> VarDec
    | VarDec COMMA ExtDecList

/* specifier */
Specifier -> TYPE
    | StructSpecifier
StructSpecifier -> STRUCT ID LC DefList RC
    | STRUCT ID

/* declarator */
VarDec -> ID
    | VarDec LB INT RB
FunDec -> ID LP VarList RP
    | ID LP RP
VarList -> ParamDec COMMA VarList
    | ParamDec
ParamDec -> Specifier VarDec

/* statement */
CompSt -> LC DefList StmtList RC
StmtList -> Stmt StmtList
    | $
Stmt -> Exp SEMI
    | CompSt
    | RETURN Exp SEMI
    | IF LP Exp RP Stmt
    | IF LP Exp RP Stmt ELSE Stmt
    | WHILE LP Exp RP Stmt

/* local definition */
DefList -> Def DefList
    | $
```

[5]The dollar sign “$” represents the empty string terminal.

```
Def -> Specifier DecList SEMI
DecList -> Dec
    | Dec COMMA DecList
Dec -> VarDec
    | VarDec ASSIGN Exp

/* Expression */
Exp -> Exp ASSIGN Exp
    | Exp AND Exp
    | Exp OR Exp
    | Exp LT Exp
    | Exp LE Exp
    | Exp GT Exp
    | Exp GE Exp
    | Exp NE Exp
    | Exp EQ Exp
    | Exp PLUS Exp
    | Exp MINUS Exp
    | Exp MUL Exp
    | Exp DIV Exp
    | LP Exp RP
    | MINUS Exp
    | NOT Exp
    | ID LP Args RP
    | ID LP RP
    | Exp LB Exp RB
    | Exp DOT ID
    | ID
    | INT
    | FLOAT
    | CHAR
Args -> Exp COMMA Args
    | Exp
```

C Additions

C.1 Tokens

We have already listed the token specification of SPL in Appendix A. You need to write regular expressions for the first four tokens, the remaining tokens correspond to the lexemes specified at the arrows' right side. (except the `TYPE` which can be one of the `int`, `float` or `char`) Here, we give more details about the tokens.

- `INT` represents an unsigned-integer literal. You can assume that they are always in 32-bits range. We have two types of integer literals, decimal (base 10) and hexadecimal (base 16) forms. An integer lexeme in decimal form is a consecutive sequence of digits 0-9, and it cannot start with "`0`", except 0 itself. A hexadecimal integer lexeme always starts with "`0x`" or "`0X`", followed by a sequence of hex-digits (0-9, a-f), again, the digit sequence cannot start with "`0`", except 0x0 itself.

- `FLOAT` represents an unsigned floating-point[6] number in IEEE-754 standard. You can assume that they are always in valid single-precision format. We do not consider the scientific notation, which is represented by a number's base-10 mantissa and exponent. A valid floating-point number always contains a single dot character (i.e., "`.`"), and there must be always digit sequence on the dot's both side.

- `CHAR` represents a single character contained in a pair of single-quotes. We do not consider those characters represented by escape sequence, e.g., "`\n`" (the newline character) or "`\t`" (the horizontal tab character).

- `ID` stands for *identifier*, which consists of 3 types of characters: the underscore (`_`), digits (0-9), and letters (A-Z and a-z). A valid identifier cannot start with digit. You can assume that the length of identifiers will not exceed 32.

C.2 Grammar rules

Here we explain the general definition of each set of productions. You should understand the meaning of each production before implementing the parser.

- **High-level definition** specifies the top-level syntax for a SPL program, including global variable declarations and function definitions.

- **Specifier** is related to the type system, in SPL, we have primitive types (`int`, `float` and `char`) and structure type.

- **Declarator** defines the variable and function declaration. Note that the array type is specified by the declarator.

- **Statement** specifies several program structures, such as branching structure or loop structure. They are mostly enclosed by curly braces, or end with a semicolon.

- **Local definition** includes the declaration and assignment of local variables.

- **Expression** can be a single constant, or operations on variables. Note that these operators have their precedence and associativity, as shown in Table 2.

[6] http://754r.ucbtest.org/background/

Table 2: Operators in SPL

Precedence	**Operator**	**Associativity**	**Description**
1	( )	left to right	parenthesis or function invocation
	[ ]		array indexing
	.		structure member accessing
2	-	right to left	negative number
	!		logical NOT
3	*	left to right	multiplication
	/		division
4	+		addition
	-		subtraction
5	<		less than
	<=		less than or equal to
	>		greater than
	>=		greater than or equal to
	==		equal to
	!=		not equal to
6	&&		logical AND
7	||		logical OR
8	=	right to left	assignment

C.3 Comments

You can optionally implement your compiler to support comments. There are two types of comment styles for SPL, single-line and multi-line comments.

A single-line comment starts with two slashes “`//`”, all symbols followed until a newline character will be dropped by the lexer.

A multi-line comment starts with “`/*`”, and ends with the first “`*/`” after that. Note that multi-line comments cannot be nested, i.e., you cannot put another multi-line comment within “`/*`” and “`*/`”.