Compilers - Part 2: Scanner

Diogo César Ferreira
11/0027931

University of Brasília

September 12, 2019

1 Introduction

In this report we will describe the implementation of the scanner, or lexical analyzer, which has the responsibility to scan the input text and split it into tokens. First we define what patterns are to be recognized by the scanner and then use the Fast Lexical Analyzer (FLEX) [1] software to generate the lexical analyzer's main functionality. Additional functions for input, output and error handling were also implemented and will be explained in detail later.

1.1 Program usage

The program takes one argument which should be a path to the input file.

```
$ ./a.out <input_file>
```

The program will then output a list of all recognized tokens, their position in the file and their value to the standard output. Should there be invalid tokens, the program will instead list the encountered errors (invalid tokens) present in the input text.

1.2 Compilation instructions

1. (Optional) Generate the lexical analyzer .c source code through FLEX

   ```
   $ flex -osrc/lex.yy.c --header-file=src/lex.yy.h part2.lex
   ```

2. Compile the program

   ```
   $ gcc src/main.c src/scanner.c src/lex.yy.c -lfl -std=c11 -m64 -O3
   ```

3. Alternatively, a makefile has also been provided which will run both commands

   ```
   $ make part2
   ```

Operating system: Debian GNU/Linux 10 (buster)

Compiler version: gcc (Debian 8.3.0-7) 8.3.0

FLEX version: flex 2.6.4

1.3 Attached files

```
./a.out              - Precompiled x64 binary
./makefile           -
./part2.lex          - FLEX definitions file
./report.pdf         - This document
./src/lex.yy.c       - FLEX generated source file
./src/lex.yy.h       - FLEX generated header file
./src/main.c         - Main and I/O functions
./src/token.c        - Auxiliary data structures source
./src/token.h        - Auxiliary data structures header
./tests/             - Sample test files
./tests/test_error1.c -
./tests/test_error2.c -
./tests/test_valid1.c -
./tests/test_valid2.c -
./tests/test_valid3.c -
```

2 Language grammar

The following table summarizes what patterns are part of this language, which are combined into an input file for FLEX (see attached file "part2.lex"):

Data types	void	char	int	float			
Keywords	if	else	do	while	return		
Arithmetical operators	+	−	*	/	%	++	−−
Comparison operators	<	>	<=	>=	==	!=	
Logical operators	&&	||					
Character/string delimiters	'	"					
Comment delimiters	/*	*/	//				
Other symbols	(	)	[	]	{	}	; , =

Character constants	text delimited by single quotation marks (')
String literals	text delimited by double quotation marks (")
Integer decimal constants	1, 2, 100, -5, +7
Integer hexadecimal constants	0x01, -0xA, +0x123
Floating-point constants	0.0, -1.234, +3.14
Comments and whitespace	are ignored.

We present a corrected version of this symbols and keywords table, where the assignment operator (=) and the "not equal" comparison operator (! =) were mistakenly not listed in the previous report, but are actually part of the language. No changes were made in the language's grammar.

1. program → function_definition
   | declaration

2. function_definition → type_specifier IDENTIFIER **(** parameter_list **)** compound_statement
   | type_specifier IDENTIFIER **( )** compound_statement

3. parameter_list → parameter_declaration
   | parameter_list , parameter_declaration

4. parameter_declaration → type_specifier declarator

5. compound_statement → { }
   | { statement_list }
   | { declaration_list }
   | { declaration_list statement_list }

6. declaration_list → declaration
   | declaration_list declaration

7. declaration → type_specifier init_declarator_list ;

8. init_declarator_list → init_declarator
   | init_declarator_list , init_declarator

9. init_declarator → declarator
   | declarator = initializer

10. declarator → IDENTIFIER
   | (declarator)
   | declarator [logical_or_expression]
   | declarator [ ]
   | declarator (identifier_list)
   | declarator (parameter_list)
   | declarator ( )

11. statement_list → statement
   | statement_list statement

12. statement → compound_statement
   | expression_statement
   | selection_statement
   | iteration_statement
   | jump_statement

13. selection_statement → IF (expression) statement
   | IF (expression) statement ELSE statement

14. iteration_statement → WHILE (expression) statement
   | DO statement WHILE (expression) ;

15. jump_statement → RETURN ;
   | RETURN expression ;

16. expression_statement → ;
   | expression ;

17. identifier_list → IDENTIFIER
   | identifier_list , IDENTIFIER

18. initializer → assignment_expression
   | { initializer_list }
   | { initializer_list , }

19. initializer_list → initializer
| initializer_list , initializer

20. expression → assignment_expression
| expression , assignment_expression

21. argument_expression_list → assignment_expression
| argument_expression_list , assignment_expression

22. assignment_expression → logical_or_expression
| postfix_expression = assignment_expression

23. logical_or_expression → logical_and_expression
| logical_or_expression || logical_and_expression

24. logical_and_expression → equality_expression
| logical_and_expression && equality_expression

25. equality_expression → relational_expression
| equality_expression == relational_expression
| equality_expression != relational_expression

26. relational_expression → additive_expression
| relational_expression < additive_expression
| relational_expression > additive_expression
| relational_expression <= additive_expression
| relational_expression >= additive_expression

27. additive_expression → multiplicative_expression
| additive_expression + multiplicative_expression
| additive_expression −− multiplicative_expression

28. multiplicative_expression → postfix_expression
| multiplicative_expression * postfix_expression
| multiplicative_expression / postfix_expression
| multiplicative_expression % postfix_expression

29. postfix_expression → primary_expression
| postfix_expression [expression]
| postfix_expression ( )
| postfix_expression (argument_expression_list)
| postfix_expression ++
| postfix_expression −−

30. primary_expression → IDENTIFIER
| CONSTANT
| STRING_LITERAL
| (expression)

31. type_specifier → VOID
| CHAR
| INT
| FLOAT

3 Error handling

The program scans the input file and splits it into tokens. Each token is then analyzed by the error handlers and stored in either a list of valid tokens or a list of invalid tokens. Every input token is analyzed, regardless of whether or not errors were found during the process, which effectively allows for every (handled) lexical error to be reported to the user. The use of the lists as buffers for the tokens also avoids clampering the output when errors are found, such that the valid token list is only displayed when no errors are found, and only errors are displayed otherwise.

Error handling is defined by additional functions on the `part2.lex` file, which will be explained in more detail on the next Section.

4 Implementation details

A separatate `main` function as well as additional functions for auxiliary data structures, error and input/output handling were implemented.

```
int  main(int argc, char** argv);      // (main.c) Main function, handles files and the program
     arguments
void print_tokens(struct tokenlist*); // (main.c) Formats the list of tokens
void print_errors(struct tokenlist*); // (main.c)  Formats the list of errors
void scan(void);                      // (main.c)  Acts as a wrapper for yylex()

// (token.h) Enumerates all possible token classes:
// keywords/types/operators/punctuation/constants/identifiers/errors
enum tokenclass {...}

// (token.h) Holds data about each token
struct token {
        int line;
        int column;
        enum tokenclass tc;
        char * text;
        struct token * next;
};

// (token.h) Linked list for storing tokens
struct tokenlist {
        struct token * first;
        struct token * last;
};

// (token.h) list manipulation and element initialization
const char* tok_to_str(enum tokenclass);
struct token * new_token(int, int, enum tokenclass, const char*);
void add_token(struct tokenlist*, struct token*);
void clear_tokens (struct tokenlist*);
```

```
// (lex.yy.c) Updates the line and column counters
static void update_position(void) {
        for(int i = 0; i < yyleng; ++i) {
                if (yytext[i] == '\n') {
                        ++nline;
                        ncolumn = 1;
                } else {
                        ++ncolumn;
                }
        }
}

// (lex.yy.c) Handles identifiers
// and its associated error (length above 32)
static int identifier(void) {
        if (yyleng > 32) {
                return ERROR_INVALID_IDENTIFIER;
        } else {
                return IDENTIFIER;
        }
}

// (lex.yy.c) handles character constants
// and its associated error (malformed char constant)
static int character_const(void) {
        if (yytext[1] != '\\' && yyleng >= 4) {
                return ERROR_INVALID_CHAR_CONST;
        } else {
                return CONSTANT;
        }
}
```

Lexical errors are treated by the functions defined in `lex.yy.c` (and `part2.lex`). When the errors are detected, `yylex()` returns a negative value associated with each error. There are three of such errors: identifiers above 32 characters in length (in `static int identifier(void)`), character constatns containing more than one character (in `static int character_const(void)`) and unrecognized symbols (dot pattern in the lexical definitions file: `. { update_position(); return ERROR_UNRECOGNIZED_TOKEN; }`, `part2.lex`, line 99, which encompasses the cases where the input did not match any other pattern).

5 Tests

Five sample test files are provided in the folder ./tests.

test_valid1.c is valid, tests for most recognized keywords and punctuation;

test_valid2.c is valid, tests for numerical values and loops;

test_valid3.c is valid, tests for character and string literals;

test_error1.c is invalid, tests for invalid tokens and malformed character literals and should detect the following errors:

- Unrecognized token at line 2, column 19 (?)
- Unrecognized token at line 2, column 23 (:)
- Invalid character constant at line 2, column 25 ('abcdef')

test_error2.c: is invalid, invalid tokens and long identifiers and should detect the following errors:

- Unrecognized token at line 1, column 1 (#)
- Unrecognized token at line 1, column 16 (.)
- Invalid identifier at line 4, column 9 (abcdefghijabcdefghijabcdefghijabcdefghij)

Both on valid and invalid inputs, the program outputs the position where it found each token or error. Lines and columns are 1-indexed and the columns should point to where the token begins. Note, however, that tabs (\t) count as if they had a length of one character.

References

[1] Paxson Vern et al. *The Fast Lexical Analyzer*. URL: `https://github.com/westes/flex`.