FORMAL LANGUAGES

DOCUMENTATION

FOR THE *JANE COMPILER*

Technical University of Kosice, KPI Korotetskyi Oleksandr 2020

TABLE OF CONTENTS:

1. INTRODUCTION

Jane – is an educational programming language using which you can perform various *arithmetic* (addition, subtraction, multiplication, division, exponentiation) and *logical* (greater, less, equal, greater-equal, less-equal, *AND, OR, NOT – are experimental*) operations and solve different algebraic equations. Moreover, *IF* and *WHILE* statements are supported in the future versions.

Jane is easy to learn and to use. Grammar and semantics of this language are small due to it's only purpose – education. Students may use this project to practice their knowledge in development of programming languages and compilers. The compiled input is headed to COMPUTRON VM which provides the user with the possibility of testing it practically.

The project was created in order to the subject "Formal Languages" in the Technical University of Kosice, Slovakia, by *Oleksandr Korotetskyi*.

The project consists of 4 coherent modules:
1] **Ifunc** module, header file
2] **Generator** module, header file
3] **Lexer** module, header file
4] **Interpret** module

The compiler was derived from the subject contents. While working process, firstly lexer operaions are performed to obtain a metadata about the input. Secondly, the interpreter proceeds with obtained data to generate machine instructions – result.

2. ENTRY CODE SPECIMEN

Jane, as an every programming language has it's own grammar. The grammar is *partially* described below:

```
/* G R A M M A R:
* Term -> VALUE | "(" Expr ")" | ID
* Power -> Term { ("^") Term }
* Mul -> Power {("*"|"/") Power}
* Expr -> Term {("+"|"-") Term}
* Eql -> Expr ( LEQ | LT | GT | LEQ | GEQ | ET | NET ) Expr
* Print_Expr -> "print" Expr ";"
```

```
* Print_Logic -> "logic" Expr ";"
* Scan -> "scan" ID {"," ID} ";"
* Var -> "var" ID {"," ID} ";"
* Let -> "let" ID ":=" Expr ";"
* Analyze -> Scan | Print | Var | Let | If | While
* Magic -> { Analyze }
*/
```

The example of an entry code:

```
var x,y;
var z;
scan z;
let x := 4;
let y := 5;
let x := x + y;
let y := x - y;
let x := x - y;
print x;
print y;
print z * x + y;
logic 4 > 9;
```

While the code is being compiled, the compiler produces the next output into command line (terminal) environment:

```
Lexical analysis results
  [18] VAR
  [ 1] ID <0> -> x
  [11] COMMA
  [ 1] ID <1> -> y
  [27] SEMICOL
  [18] VAR
  [ 1] ID <2> -> z
  [27] SEMICOL
  [ 2] SCAN
  [ 1] ID <2> -> z
  [27] SEMICOL
  [19] LET
  [ 1] ID <0> -> x
  [23] SETVAL
  [ 0] VALUE <4>
  [27] SEMICOL
  [19] LET
  [ 1] ID <1> -> y
  [23] SETVAL
  [ 0] VALUE <5>
  [27] SEMICOL
  [19] LET
  [ 1] ID <0> -> x
  [23] SETVAL
  [ 1] ID <0> -> x
```

```
  [ 4] PLUS
  [ 1] ID <1> -> y
  [27] SEMICOL
  [19] LET
  [ 1] ID <1> -> y
  [23] SETVAL
  [ 1] ID <0> -> x
  [ 5] MINUS
  [ 1] ID <1> -> y
  [27] SEMICOL
  [19] LET
  [ 1] ID <0> -> x
  [23] SETVAL
  [ 1] ID <0> -> x
  [ 5] MINUS
  [ 1] ID <1> -> y
  [27] SEMICOL
  [ 3] PRINT
  [ 1] ID <0> -> x
  [27] SEMICOL
  [ 3] PRINT
  [ 1] ID <1> -> y
  [27] SEMICOL
  [ 3] PRINT
  [ 1] ID <2> -> z
  [ 6] MUL
  [ 1] ID <0> -> x
  [ 4] PLUS
  [ 1] ID <1> -> y
  [27] SEMICOL
  [31] LOGIC
  [ 0] VALUE <4>
  [25] GEQ
  [ 0] VALUE <9>
  [27] SEMICOL
  [12] SEOF
Beginning of interpretation...
z = 0
Result:   5
Result:   4
Result:   4
Logical expression: false
```

The provided code is used only for showing the compiler is functional an may be easily substituted by user.

3. GENERATOR

Generator is used for generating the machine language instructions on the base of the code which is being proceeded by the compiler. These instructions may be used in the COMPUTRON VM later. This module contains a plenty of functions which are responsible for a data transfer to a binary file named “magic.bin”. These functions are the next:

- `void init_generator(FILE *output);`

- `void generate_output();`
- `short get_address();`
- `void write_begin(short num_vars);`
- `void write_end();`
- `void write_result();`
- `void write_number(short value);`
- `void write_var(short index);`
- `void write_add();`
- `void write_sub();`
- `void write_mul();`
- `void write_div();`
- `void write_pow(int value);`
- `void write_EQUAL();`
- `void write_NET();`
- `void write_LT();`
- `void write_GT();`
- `void write_LEQ();`
- `void write_GEQ();`

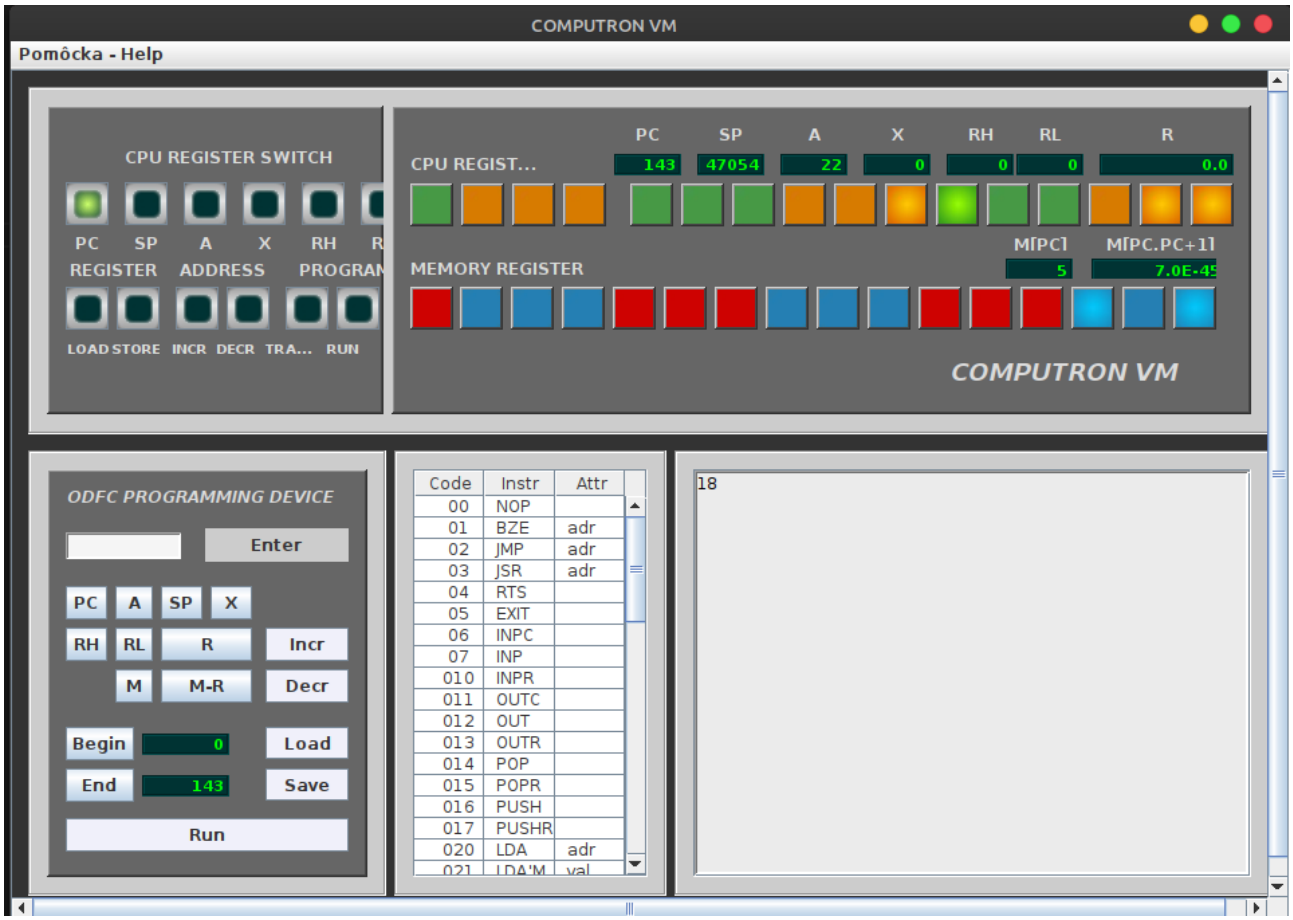
Almost all these functions are partially commented in the *generator.c* and *generator.h* files.

The instructions generated by the code from the example:

```
// atributy su uvedene v 8-kovej sustave (ako v Computrone)
// pred kazdou instrukciou je v [] uvedena pamatova adresa
// (je to adresa pamati v Computrone, ak je program nacitany od 0)
[00] JMP 06
// nerozpoznana instrukcia
[02] NOP
[03] NOP
[04] NOP
[05] LDS 02
[07] LDAM 04
[011] PUSH
[012] LDA 00
[014] PUSH
[015] LDAM 05
[017] PUSH
[020] LDA 01
[022] PUSH
[023] LDA 00
[025] PUSH
[026] LDA 01
[030] PUSH
[031] POP
[032] STA 02
[034] POP
[035] ADD 02
[037] PUSH
[040] LDA 00
[042] PUSH
```

```
[043] LDA 00
[045] PUSH
[046] LDA 01
[050] PUSH
[051] POP
[052] STA 02
[054] POP
[055] SUB 02
[057] PUSH
[060] LDA 01
[062] PUSH
[063] LDA 00
[065] PUSH
[066] LDA 01
[070] PUSH
[071] POP
[072] STA 02
[074] POP
[075] SUB 02
[077] PUSH
[0100] LDA 00
[0102] PUSH
[0103] LDA 00
[0105] PUSH
[0106] LDA 01
[0110] PUSH
[0111] LDA 02
[0113] PUSH
[0114] LDA 00
[0116] PUSH
[0117] POP
[0120] STA 02
[0122] POP
[0123] MUL 02
[0125] PUSH
[0126] LDA 01
[0130] PUSH
[0131] POP
[0132] STA 02
[0134] POP
[0135] ADD 02
[0137] PUSH
[0140] LDAM 04
[0142] PUSH
[0143] LDAM 011
[0145] PUSH
[0146] POP
[0147] STA 02
[0151] POP
[0152] EQ 02
[0154] PUSH
```

```
[0155] POP
[0156] OUT
[0157] EXIT
```



4. LEXER

Lexer is used to define keywords and analyze the input file content to classify it. This module consists of two files: *lexer.c* and *lexer.h*. In it, we prevent the recognition of spaces, as well as define punctuation by type plus, minus, multiplication, division, exponentiation and other operations. Also in the lexical analyzer, we define names for each punctuation mark or keyword to write these indicators in the lexical analysis in the future.

Those functions are the basement of the lexer (every function is *fully* commented in the coherent project files):

- `void init_lexer(char *string)`
- `int store_id(char *id)`
- `void next_symbol ()`
- `const char * symbol_name (Symbol symbol)`
- `void print_tokens ()`

Moreover, the module contains the enumeration type for all the possible lexical meanings of the encountered symbol or keyword to use them in the future (also described in the coherent project files) as well as other necessary dependencies.

5. INTERPRETER

The **interpreter** is the main module of the project which is responsible for handling all the symbols from the logical point of view; it responsible for defining and performing actions according to the keywords and symbols that are specified in entry code. The keywords themselves are recognized in the lexer file, and this module works on data obtained from lexer performance. In addition, it contains the main function as well as functionality for handling error cases.
Description of the some of important functions of module:

- `int main(int argc, char** argv)` - The main function is the starting point for executing the program. It controls the execution of a program, invoking other functions;
- `void analyze(KeySet K)` – a function to define functions according to the specified keywords. (set, print, read, var, logic);
- `void error(const char *msg, KeySet K)` - writing errors related to arithmetic and logical operations, also performs the function of skipping non-keywords (read, print, var, if, etc.);
- `void check(const char *msg, KeySet K)` - a review of the word in case of an inaccuracy in the writing of the indemnity;
- `int match(const Symbol expected, KeySet K)` - Verify the symbol on the input and read the next one. Returns the attribute of a verified symbol;
- `int term(KeySet K)` - one of the functions that is associated with the grammatical verification of the code you specify. Determines the priority of parentheses in the arithmetic example you specify. Checks for characters. In case of malfunction will write out various errors;
- `int ipower(KeySet K)` - one of the functions that is associated with the grammatical verification of the code you specify. Performs elevation actions if the '^' character is present in your code. Writes errors related to the action of elevation;
- `int mul(KeySet K)` - one of the functions that is associated with the grammatical verification of the code you specify. Performs division and multiplication operations if there are '*' or '/' characters in your code. Displays errors related to multiplication and division;
- `int expr(KeySet K)` - one of the functions that is associated with the grammatical verification of the code you specify. Performs addition and subtraction operations if there are '+' or '-' characters in your code. Writes errors related to addition or subtraction;
- `int logical_expr(KeySet K)` - one of the functions that is associated with the grammatical verification of the code you

specify. Performs the actions related to logical operations (greater, less, equal, greater-equal, less-equal, logical operations AND, OR, NOT). Returns true or false. Writes errors related to the use of logical operations;

- `void print_Expr(KeySet K)` – a function for writing the results of algebraic expressions. To use, enter the keyword print Expr; (for example: print 2 + 2;);

- `void print_Logic(KeySet K)` – a function deriving the results from logical expressions. To use, enter the keyword logic Expr; (Example: logic 3 <= 2;);

- `void scan(KeySet K)` – a function to scan data from the console. Uses the void scan_var(int id_idx) function. (Example: var x; scan x; print x;);

- `void c_var(KeySet K)` – a function for declaring variables. Is needed in order to perform let, read and print operations on variables. (Example: var x; let x := 4; print x;);

- `void let(KeySet K)` - a function for assigning values to a variable. To use, you need to declare the variable and use the keyword let: = VALUE. (Example var x; let x: = 3; print x;);

6. MODULE IFUNC

Ifunc is responsible for getting the input from file and keyboard. The obtained input is stored in a string named *source*. Moreover it includes the enumeration type describing the error cases which may occur during working with input file. There are two possible scenarios: the program is run and provided with an input file filename given as an argument in command line (terminal); the program is provided with nothing. In the second case user is asked whether he desire to quit, to enter the input file filename, to enter the code manually.

All the functions are described in *ifunc.h.*

7. CONCLUSION

During the development process I faced a plenty of problems coherent to different fields: from health to university. That is why a project itself was a great challenge for me to complete. From the very beginning, I was guided by the project the students are provided with in the subject web pages, but not all contents were clear to me. Frankly speaking I even wanted to create the project from scratch fully by myself, because I understood the idea of how it should work, but did not understood how to implement that idea in the project I was provided with. After spending hours on studying lectures and reading literature I started to understand all the basic principles and concepts of implemenation, and finally, I managed to create a simple compiler by myself.

The very adaptation of the project to the task conditions began with lexer. I have added some keywords, arithmetic signs and other functionality. Then I rearranged everything in the interpreter. Unfortunately, not all tasks were implemented due to the lack of knowledge and practice in the compiler design: IF and WHILE statements were not fully implemented. The biggest number of difficulties I have encountered was in

Interpreter module. The difficulties were coherent with bitwise operators and bit functionality used in functions design.

In conclusion, I think my personal opinion is worth mentioning. I really liked the challenge the compiler was to me at the very beginning. I eager for the new knowledge and I think that write a compiler is always a good idea. But, on the other hand, the information we are provided with was not as useful as it could be. There were many unclear points that had no explanation in the course materials. For example, we could have been provided with the *for* cycle implementation as an example to further development of *while* and *do-while* cycles. Also, I did not really understood how to assign a value to a variable as was described in the task. That is why, the *let* keyword appeared: to make the interpretation easier. All in all I am grateful that I managed to implement a tiny compiler by myself.

Additional completed tasks:

Task 5: A1, A2

What development environments did I use?

Linux Mint OS, Windows 10 OS; Microsoft Visual Studio Code, Linux Terminal

8. LITERATURE

1) Basics of Compiler Design, Torben Ægidius Mogensen, (DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF COPENHAGEN)
2) Ján Kollár, PREKLADAČE