JANE

Programming language

Yaroslav Melnychyn

2020

Documentation

Description of language grammar

JANE - is a powerful programming language with which you can perform various algebraic equations (addition, subtraction, multiplication, division, exponentiation), perform logical operations (more, less, equal, more-equal, less-equal, logical operations AND, OR, NOT).

It is not difficult to study, but in order to understand it, you need to understand the project in which it was created. It was invented in 2020 by a second-year student Yaroslav Melnychyn of the Technical University in Kosice on the basis of a formal languages.

So the project consists of 3 parts, which play a separate role in the whole compiler:

1. interpret.c
2. lexer.c
3. generator.c

Example of an introductory code

```
var x,y;
set x := 4;
set y := 5;
set x := x + y;
set y := x – y;
set x := x – y;
print x;
print y;
```

Syntactic analyzer or Interpreter

The interpreter is the main file in my project, it contains the main function, and is responsible for defining actions according to the keywords that we specify in our code. The keywords themselves are recognized in the lexer file, which I'll talk about later. In the interpreter, I describe the main actions with keywords, that is, I show what each word should do, for example: "var x;" must work variables. Also in this file I make a list of the main errors related to grammar, i.e. incorrect use of keywords.

Description of the following functions in the interpreter:

- int main(int argc, char** argv) - The most important function in the project. The main function is the starting point for executing the program. It controls the execution of a program, invoking its other functions;
- void readLine() - this function acts as a read from the console, it allows you to read not only the string, but reads to the EOF mark;
- int exponentiation(int number, int exponentiation) - performs the function of a simple elevation to the degree;
- void scan(int id_idx) - used in the read function to read and write data to a variable.
- 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 of the case in question in the event 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 power(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 logic(KeySet K) - one of the functions that is associated with the grammatical verification of the code you specify. Performs actions related to logical operations (more, less, equal, more-equal, less-equal, logical operations AND, OR, NOT). Returns 0 or 1 (true or false). Writes errors related to the use of logical operations;
- void print(KeySet K) - 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) - function for counting the results of logical expressions. To use, enter the keyword logic Expr; (for example: logic 3> 2;);

- void read(KeySet K) - function for reading data from the console. Uses the void scan (int id_idx) function. To work, you must first declare a variable. (Example: var x; read x; print x;);

- void var(KeySet K) - function for declaring variables. Needed to use in order to then perform set, read, print operations on variables. (For example: var x; set x: = 4; print x;);

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

- void stat(KeySet K) - function to define functions according to the specified keywords. (set, print, read, var, logic);

Lexical analyzer or Lexer

Lexer is used to define keywords using switch-case. In it, we prevent the recognition of spaces, as well as define punctuation by type plus, minus, multiplication, division, exponentiation and many 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.

Description of the following functions in the lexer:

- void init_lexer(char *string) - Initialization of the lexical analyzer. The parameter is the input string.
- int store_id(char *id) - Store the identifier `id` in the identifier table if it is not already there. Returns the index on which the identifier is stored.
- void next_symbol () - Reads the next character. Defines keywords and calls enum for easier use. Also defines the main arithmetic operations by reading the introductory line.
- const char * symbol_name (Symbol symbol) - Returns the name of the lexical unit recognized by the next_symbol () function. An array (SYM_NAMES) is required for operation, which writes the name of each unit, according to the value in enum.
- void print_tokens () - List of all lexical units from the introduction

Type of basic units:

- Types of symbols - lexical units:

```
typedef enum {
    VALUE, ID, READ, PRINT,LOGIC, PLUS, MINUS, MUL, DIV, POWER,LPAR,
    RPAR, COMMA,SERROR, IF, THEN, ELSE, WHILE, VAR, SET, LT,EQUAL,
    GT, SETVAL, LTEQUAL, GTEQUAL, NOT, SEMCOL, SEOF, AND, OR
} Symbol;
```

- Symbol names:

```
const char *SYM_NAMES[] = {
       [VALUE] = "VALUE", [ID] = "ID", [READ] = "READ",
       [PRINT] = "PRINT", [LOGIC] = "LOGIC",
       [PLUS] = "PLUS", [MINUS] = "MINUS", [MUL] = "MUL",
       [DIV] = "DIV", [POWER] = "POWER",
       [LPAR] = "LPAR", [RPAR] = "RPAR", [COMMA] = "COMMA",
       [SEOF] = "SEOF", [SERROR] = "SERROR",
       [IF] = "IF", [THEN] = "THEN", [ELSE] = "ELSE",
       [WHILE] = "WHILE", [VAR] = "VAR", [SET] = "SET",
       [SEMCOL] = "SEMCOL", [EQUAL] = "EQUAL",
       [LT] = "LESS", [GT] = "GREATER", [SETVAL] = "SETVAL",
       [LTEQUAL] = "LESSEQUAL", [GTEQUAL] = "GREATEREQUAL",
       [NOT] = "NOT",[AND]= "AND",[OR] = "OR"
};
```

Generator

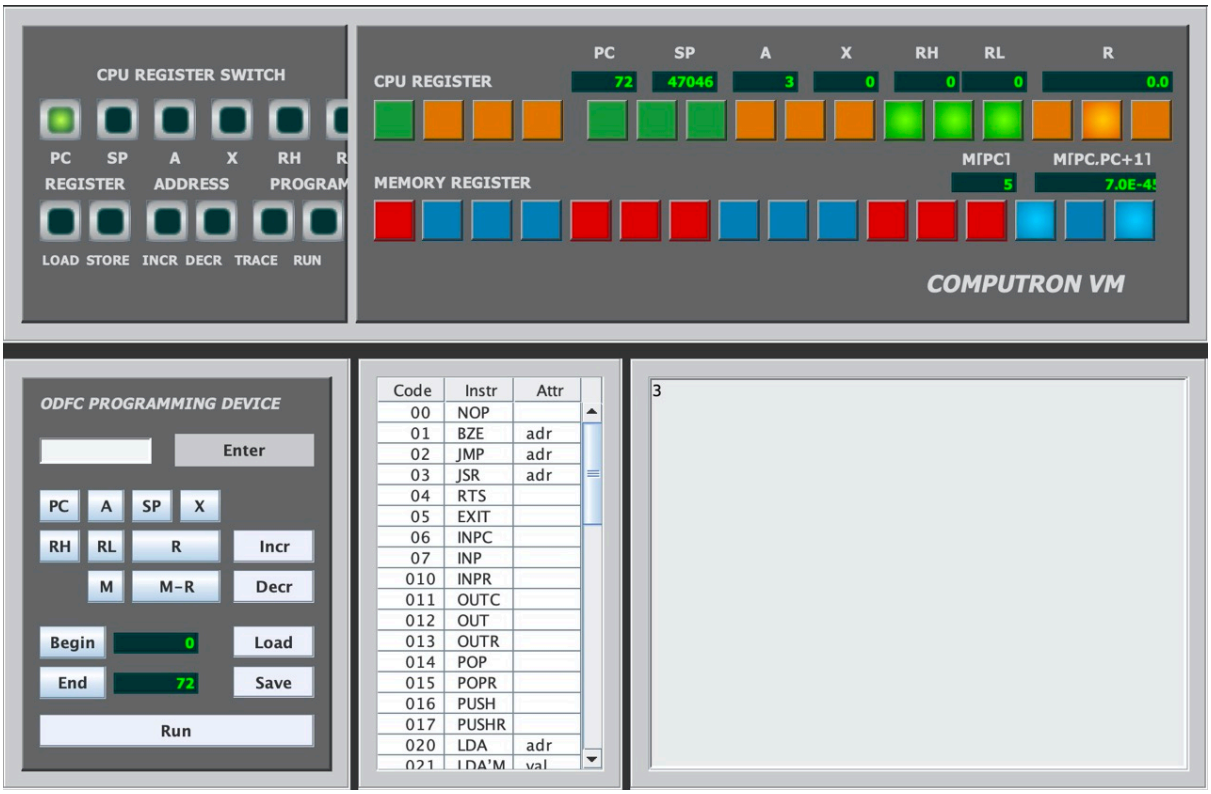
Generator generates code for us, which we then write to a binary file and use in Computron VM. Contains many functions that are responsible for a data transfer to a binary file. Here they are:

- 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_ask_var(short index, char *name);
- void write_pow(int value);
- void write_EQUAL();
- void write_NOT();
- void write_LT();
- void write_GT();
- void write_LESSEQUAL();
- void write_GTEQUAL();

Description of the code from the disassembler:

// 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 03
// nerozpoznana instrukcia
[02] LDS 02
[04] LDAM 04
[06] PUSH
[07] LDAM 05
[011] PUSH
[012] LDA 00
[014] PUSH
[015] LDA 01
[017] PUSH
[020] POP
[021] STA 02
[023] POP
[024] ADD 02
[026] PUSH
[027] LDA 00
[031] PUSH
[032] LDA 01
[034] PUSH
[035] POP
[036] STA 02
[040] POP
[041] SUB 02
[043] PUSH
[044] LDA 00
[046] PUSH
[047] LDA 01
[051] PUSH
[052] POP
[053] STA 02
[055] POP
[056] SUB 02
[060] PUSH
[061] LDA 00
[063] PUSH
[064] LDA 01
[066] PUSH
[067] POP
[070] OUT
[071] EXIT

Computron VM after passing through the binary code



Conclusion

- How the project was created:
  This project took me a long time. From the beginning, I was guided by the project, which was provided to us on the flowers, then began to adapt my project to the conditions that sets for me.

- What development environments did I use:
  I also changed the IDE in which I worked. At first I worked in the terminal, and then when I passed all the lessons, I switched to CLion, because it is much easier to edit the project.

- How the project changed:
  The very adaptation of the project to the task conditions began with lexer. I added some keywords, arithmetic signs and more. Then rearranged everything in the interpreter.

- Problems encountered while working on the project:
  There were very few problems. I think this is due to good project planning, because I was not confused and did not irritate anything. The first difficulties were in the interpreter, when I started creating new functions for logical and arithmetic operations.

- What I liked and what I didn't:
  At first I didn't like the fact that I understood almost nothing. I did not quite understand how everything works in the skeleton of the project given to us in the lessons. I even wanted to create the whole project from scratch. Then I started to rebuild this project and at the end of the task began to seem very easy, I began to navigate the project and understand where the mistakes are, and most importantly, how to fix them.

- Additional completed tasks:
  - Task 5: A1, A2;