SER 502 Project: Compiler and Virtual Machine for a Programming Language

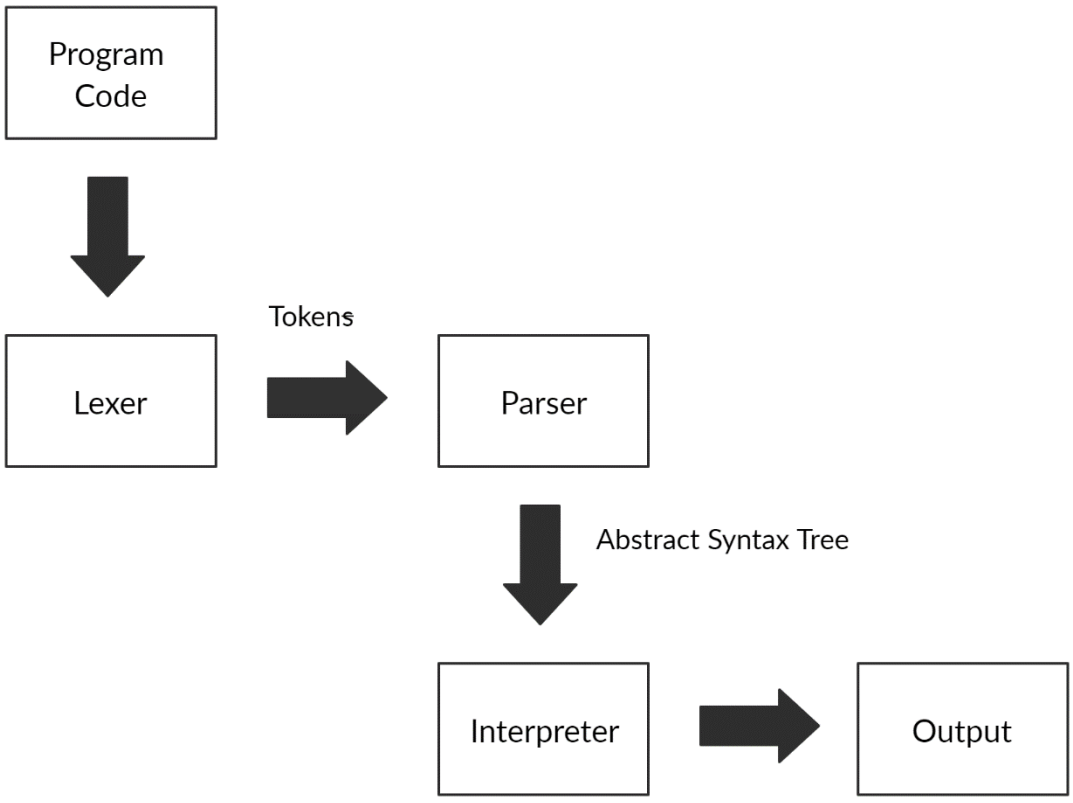
Team 11 – Milestone II:

GitHub Repository URL: https://github.com/AmudhanManisekaran/SER502-Spring2020-Team11

YouTube URL: https://www.youtube.com/watch?v=pqQUxUurmqc&t=470s

Language Name: CodeZilla

Language Design :



We have implementd the lexical analyzer in Python, Parser, and Interpreter using Definite Clause Grammar (DCG) in Prolog, a logic programming language.

Lexer (Lexical Analyser): Python program will be written to read the user's code. This code will be converted to a set of tokens, which will be stored in a token file**.**

- **Input:** program.cz
- **Output:** program.tok

Parser: The tokens from the Lexer will be passed through the Parser containing Prolog rules. These rules will generate the equivalent Abstract syntax trees. These syntax trees will act as the intermediate code.

- **Input:** program.tok
- **Output:** program.pt

Interpreter: The interpreter will evaluate the parse tree obtained from the previous step using Syntax-directed semantics. At each node in the parse tree, the environment will be updated.

- **Input:** program.pt
- **Output:** program.cz code's output

The Parser and Interpreter rules will be written in **SWI-Prolog**.

The **data types** supported by CodeZilla are:

- Integer
- Float
- String
- Boolean

The **operators** supported by CodeZilla are:

- Addition
- Subtraction
- Multiplication
- Division
- Comparison (<, >, <=, >=, =:=)
- AND, OR, NOT for Boolean expressions.
- Ternary ($)
- Assignment (=)

The **conditionals** supported by CodeZilla are:

- Standard ‘if’
- If then else

The **loops** supported by CodeZilla are:

- While
- Traditional For
- For

The **terminators** and **delimiters** used are:

- Start
- End
- Semi-colon (;)

CodeZilla allows the user to print the numbers, boolean values and strings using **show** keyword.

User input can be obtained by using the **read** keyword.

Grammar Rules:

PROGRAM ::= start; BLOCK end;

BLOCK ::= DECLARELIST; CONDITIONLIST;

DECLARELIST ::= DECLARE; DECLARELIST | DECLARE

DECLARE ::= var ID | str ID

CONDITIONLIST ::= CONDITION; CONDITIONLIST | CONDITION

CONDITION ::= ASSIGN | IF | TERNARY | LOOPS | show DATA endshow| read ID endread

Conditional Operations:

ASSIGN ::= ID = EXPSET | ID = STRING | ID = BOOL

IF ::= if BOOL then CONDITIONLIST endif | if BOOL then CONDITIONLIST else CONDITIONLIST endif

TERNARY ::= ID = BOOL $ EXPSET / EXPSET endternary

LOOPS ::= while BOOL do CONDITIONLIST endwhile | for (ID = VALUE ; BOOL ; EXPSET): CONDITIONLIST endfor | for ID inrange (VALUE : VALUE): CONDITIONLIST endfor

Boolean Expressions:

BOOL ::= true | false | BOOL and BOOL | BOOL or BOOL | not (BOOL) | BOOL LESSTHAN BOOL | BOOL GREATERTHAN BOOL | BOOL LESSTHANEQUAL BOOL | BOOL GREATERTHANEQUAL BOOL | BOOL EQUAL BOOL.

Arithmetic Expressions:

EXPSET ::= EXP | ID = EXPSET

EXP ::= EXP + TERM | EXP – TERM | TERM

TERM ::= TERM / FACTOR | TERM * FACTOR | FACTOR

FACTOR ::= VALUE | (EXPSET)

VALUE ::= ID | INT | FLOAT

Identifiers:

ID ::= ^ [a-z] $

Data types:

INT ::= ^ [0-9]+ $

FLOAT ::= ^ [0-9]+ \ . [0-9] $

STRING ::= << ^[a-z] [a-zA-Z]+ $ >>

Keywords:

DATA ::= STRING | ID

LESSTHAN ::= <

LESSTHANEQUAL ::= <=

GREATERTHAN ::= <

GREATERTHANEQUAL ::= <=

EQUAL ::= ==

Sample run:

- ***factorial.cz***

```
#program to calculate factorial
start;
var f;
var n;
var i;
f = 1;
read n endread;
for (i = 1; i <= n; i = i + 1): f = f * i endfor;
show << factorial is >> endshow;
show f endshow;
end;
```

- ***factorial.tok***

[start, semicolon, var, f, semicolon, var, n, semicolon, var, i, semicolon, f, equal, 1, semicolon, read, n, endread, semicolon, for, open_para, i, equal, 1, semicolon, i, less_thanequal, n, semicolon, i, equal, i, +, 1, close_para, colon, f, equal, f, *, i, endfor, semicolon, show, less_than, less_than, factorial_is, greater_than, greater_than, endshow, semicolon, show, f, endshow, semicolon, end, semicolon].

- ***factorial.pt***

t_parser(t_program(t_block(t_declarationLINE(t_declarationVAR(t_id(f)),t_declarationLINE(t_declarationVAR(t_id(n)),t_declarationLINE(t_declarationVAR(t_id(i))))),t_commandLINE(t_command_assign(t_assign_var(t_id(f),t_expr(t_fact(t_value(t_int(1)))))),t_commandLINE(t_command_read(t_read(t_id(n))),t_commandLINE(t_command_loops(t_loops(t_trad_for(t_id(i),t_value(t_int(1)),t_lessth

anequal(t_expr(t_fact(t_value(t_id(i)))),t_expr(t_fact(t_value(t_id(n)))),t_assign(t_id(i),t_expr(t_add(t_fact(t_value(t_id(i))),t_fact(t_value(t_int(1)))))),t_commandLINE(t_command_assign(t_assign_var(t_id(f),t_expr(t_mul(t_fact(t_value(t_id(f))),t_fact(t_value(t_id(i))))))))),t_commandLINE(t_command_show(t_show(t_data(t_str(factorial_is)))),t_commandLINE(t_command_show(t_show(t_data(t_id(f))))))).

- ***Output***

  ?- codezilla('factorial.pt').

  please enter value for:n

  |: 5.

  factorial_is

  f = 120

  true.

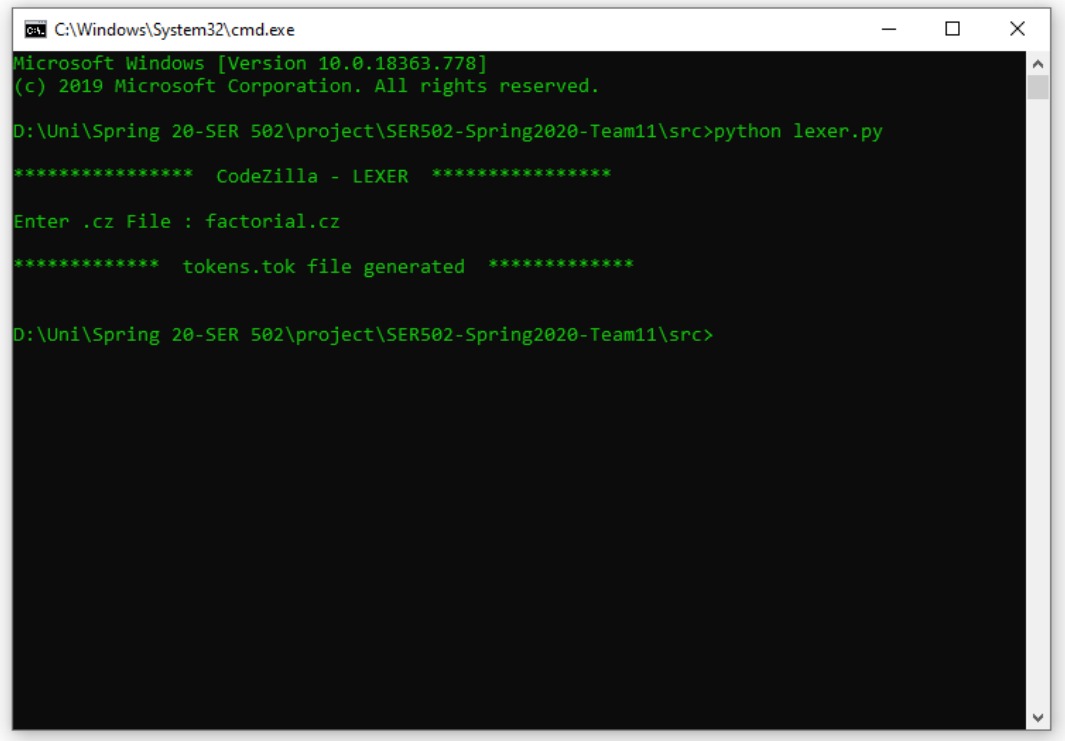
Screenshots:

CodeZilla (.cz) file

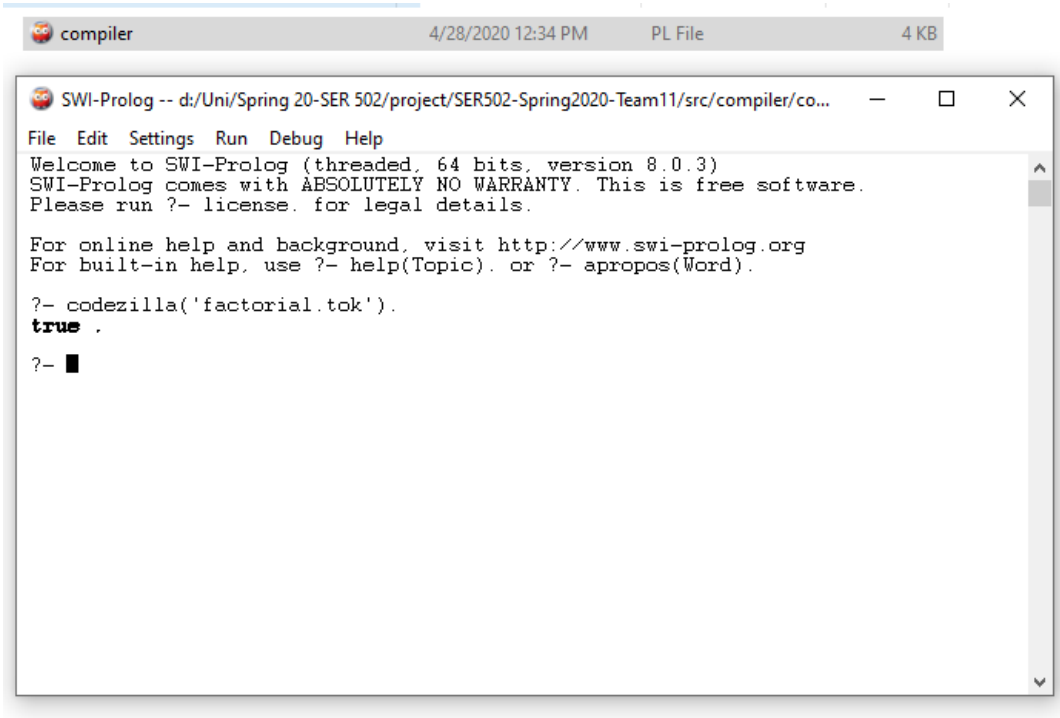
factorial.cz

```
#program to calculate factorial
start;
var f;
var n;
var i;
f = 1;

read n endread;
for (i = 1; i <= n; i = i + 1): f = f * i endfor;

show << factorial is >> endshow;
show f endshow;

end;
```

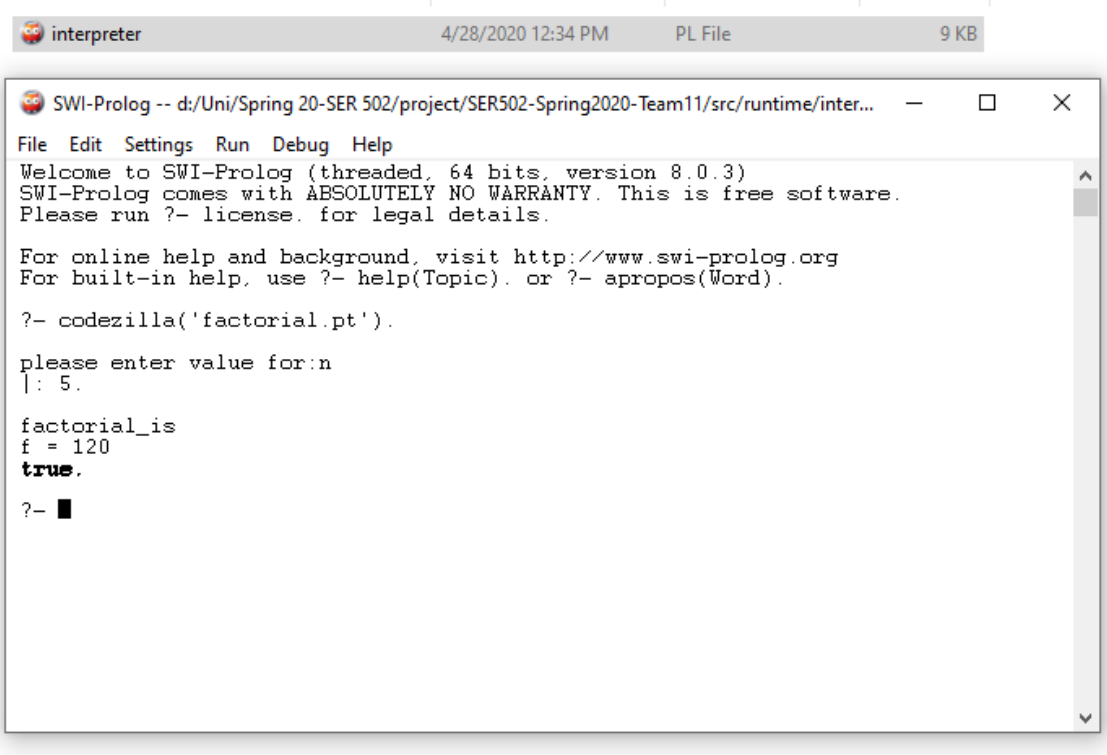
Lexer



Parser



Interpreter



Changes in Grammar from Milestone 1 :

- For loop (traditional and range) structure changed based on feedback from TA and Professor.
- Changed bool structure in grammar for easy access.
- Newly added assignment inside expressions, boolean value assignment to variables.
- Newly added endshow, endread, endternary, endfor to identify termination.
- Space keyword in grammar removed since Lexer was made to check it inherently.
- Regular expression for id and string modified.