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

# **Team 11 – Milestone II:**

* 1. **GitHub Repository URL:** <https://github.com/AmudhanManisekaran/SER502-Spring2020-Team11>
  2. **Language Name:** CodeZilla
  3. **Language Design :**
  4. **A close up of a piece of paper

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

* 1. **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
  1. **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
  1. **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
  1. The Parser and Interpreter rules will be written in **SWI-Prolog**.
  2. 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.

* 1. **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\_lessthanequal(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.

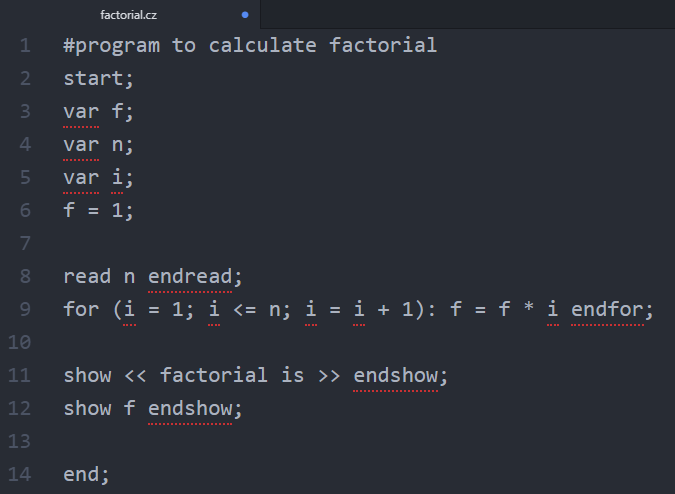
factorial\_is

f = 120

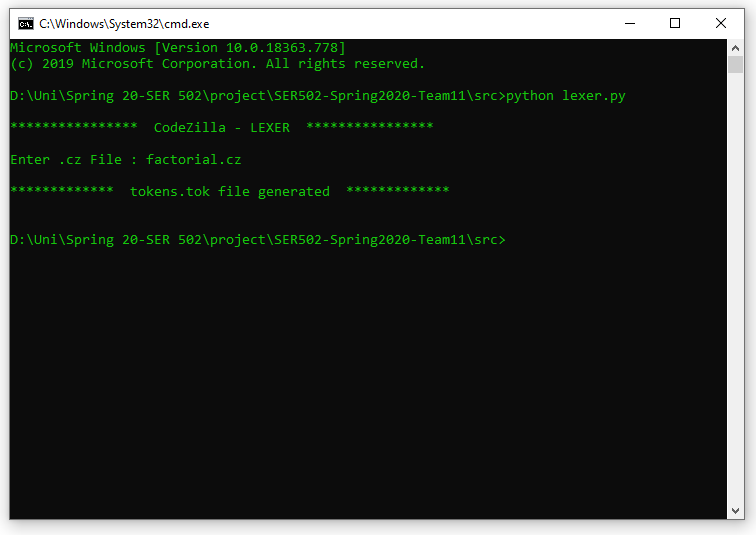
true.

**Screenshots:**

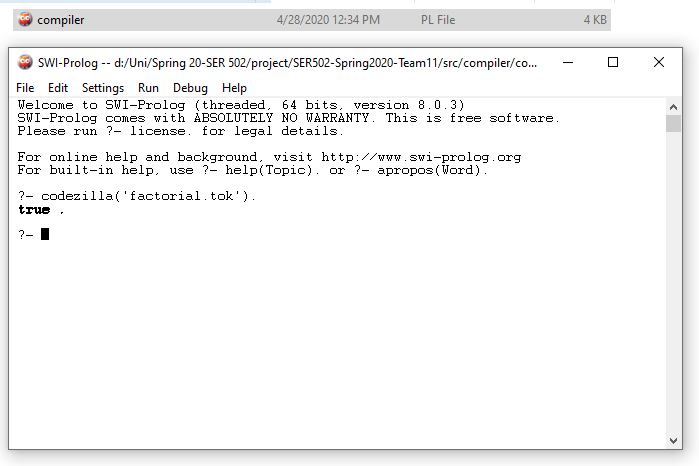
*CodeZilla (.cz) file*

****

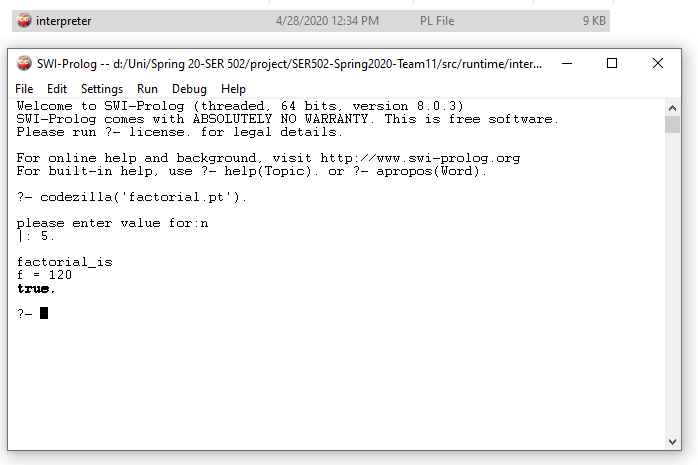
*Lexer*



*Parser*



*Interpreter*



**Changes in Grammar from Milestone 1 :**

* For loop (traditional and range) structure changed from feedback in Milestone1 from TA and Professor.
* Changed bool structure in grammar for easy access.
* Newly added assignment inside expressions, boolean value assignment to variable.
* Newly added endshow, endread, endternary, endfor for readability.
* Space keyword in grammar removed since it was inherently checked by lexer.
* Regular expression for id and string modified.
* Double datatype, block inside a command line, constant declaration removed since they were not compulsory and we had less time.