IMPETUS Programming Language

Purushothama Shathappa, Praneeth Kumar Reddy Kotha, Rahul Ghanghas

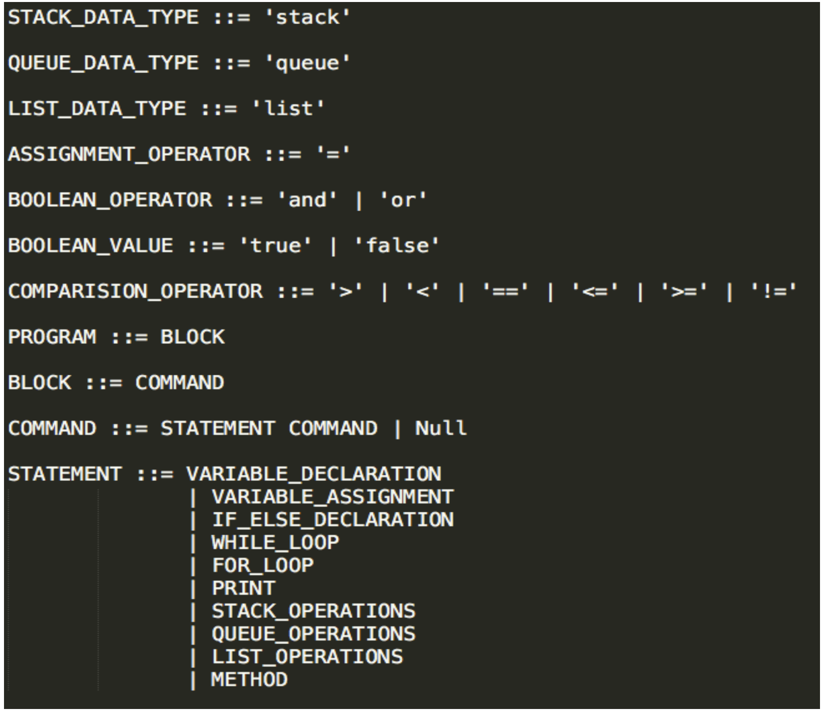
Language Features

Features	Types
Datatype	num, boolean, string
Boolean operator	and, or, not
Relational operator	<, >, <=, >=, ==, !=
Arithmetic operator	+, -, /, *, (,)
Assignment operator	=
Conditional operator	if else, ternary
Looping Construct	traditional for and while loop, for in range(x,y)
Printing	print()

Language Features

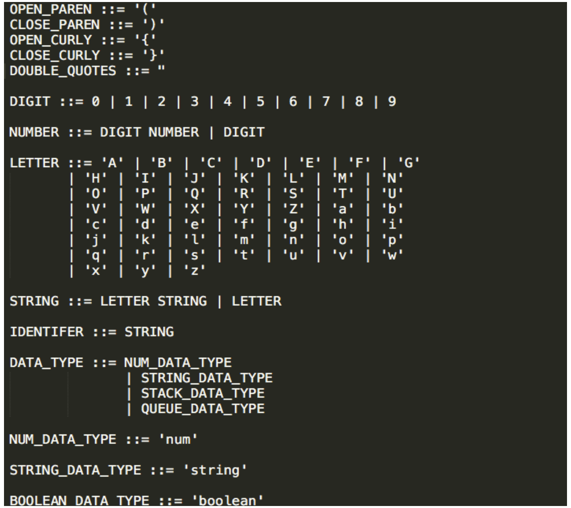
- Data structure – Stack, Queue and List
- String Concatenation operation
- Variable Scope Checking
- Type Checking during Parsing
- Functions

Grammar

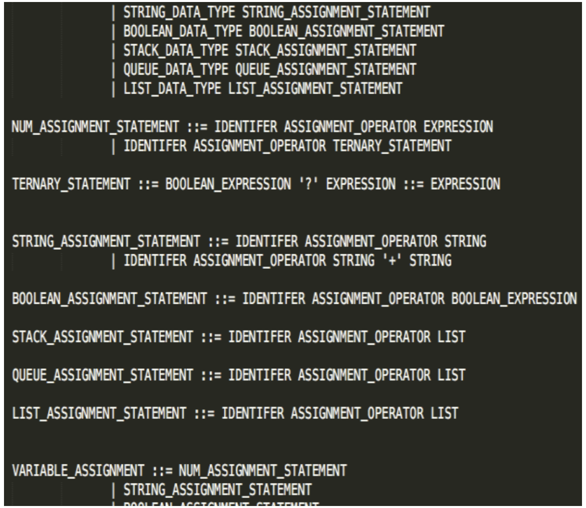


```
STACK_DATA_TYPE ::= 'stack'

QUEUE_DATA_TYPE ::= 'queue'

LIST_DATA_TYPE ::= 'list'

ASSIGNMENT_OPERATOR ::= '='

BOOLEAN_OPERATOR ::= 'and' | 'or'

BOOLEAN_VALUE ::= 'true' | 'false'

COMPARISION_OPERATOR ::= '>' | '<' | '==' | '<=' | '>=' | '!='

PROGRAM ::= BLOCK

BLOCK ::= COMMAND

COMMAND ::= STATEMENT COMMAND | Null

STATEMENT ::= VARIABLE_DECLARATION
              | VARIABLE_ASSIGNMENT
              | IF_ELSE_DECLARATION
              | WHILE_LOOP
              | FOR_LOOP
              | PRINT
              | STACK_OPERATIONS
              | QUEUE_OPERATIONS
              | LIST_OPERATIONS
              | METHOD
```

Grammar

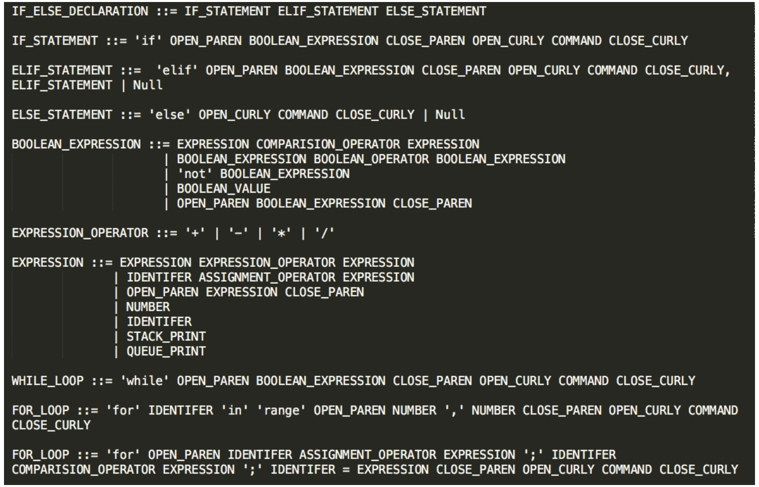


```
OPEN_PAREN ::= '('
CLOSE_PAREN ::= ')'
OPEN_CURLY ::= '{'
CLOSE_CURLY ::= '}'
DOUBLE_QUOTES ::= "

DIGIT ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

NUMBER ::= DIGIT NUMBER | DIGIT

LETTER ::= 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
       | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N'
       | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U'
       | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a' | 'b'
       | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i'
       | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p'
       | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w'
       | 'x' | 'y' | 'z'

STRING ::= LETTER STRING | LETTER

IDENTIFER ::= STRING

DATA_TYPE ::= NUM_DATA_TYPE
            | STRING_DATA_TYPE
            | STACK_DATA_TYPE
            | QUEUE_DATA_TYPE

NUM_DATA_TYPE ::= 'num'

STRING_DATA_TYPE ::= 'string'

BOOLEAN_DATA_TYPE ::= 'boolean'
```

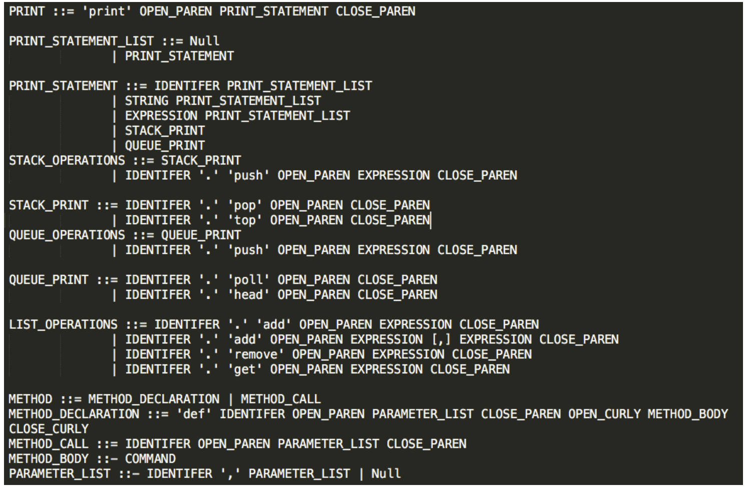


```
            | STRING_DATA_TYPE STRING_ASSIGNMENT_STATEMENT
            | BOOLEAN_DATA_TYPE BOOLEAN_ASSIGNMENT_STATEMENT
            | STACK_DATA_TYPE STACK_ASSIGNMENT_STATEMENT
            | QUEUE_DATA_TYPE QUEUE_ASSIGNMENT_STATEMENT
            | LIST_DATA_TYPE LIST_ASSIGNMENT_STATEMENT

NUM_ASSIGNMENT_STATEMENT ::= IDENTIFER ASSIGNMENT_OPERATOR EXPRESSION
            | IDENTIFER ASSIGNMENT_OPERATOR TERNARY_STATEMENT

TERNARY_STATEMENT ::= BOOLEAN_EXPRESSION '?' EXPRESSION ::= EXPRESSION


STRING_ASSIGNMENT_STATEMENT ::= IDENTIFER ASSIGNMENT_OPERATOR STRING
            | IDENTIFER ASSIGNMENT_OPERATOR STRING '+' STRING

BOOLEAN_ASSIGNMENT_STATEMENT ::= IDENTIFER ASSIGNMENT_OPERATOR BOOLEAN_EXPRESSION

STACK_ASSIGNMENT_STATEMENT ::= IDENTIFER ASSIGNMENT_OPERATOR LIST

QUEUE_ASSIGNMENT_STATEMENT ::= IDENTIFER ASSIGNMENT_OPERATOR LIST

LIST_ASSIGNMENT_STATEMENT ::= IDENTIFER ASSIGNMENT_OPERATOR LIST


VARIABLE_ASSIGNMENT ::= NUM_ASSIGNMENT_STATEMENT
            | STRING_ASSIGNMENT_STATEMENT
            | BOOLEAN_ASSIGNMENT_STATEMENT
```

Grammar

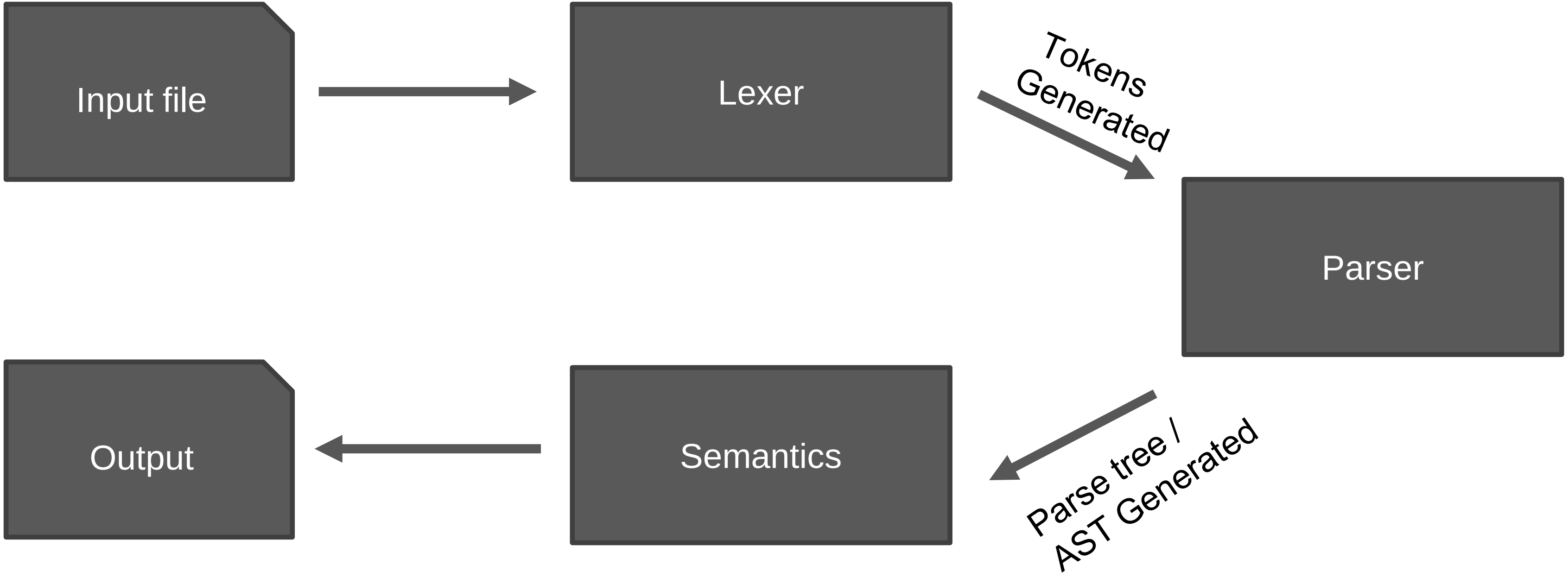


```
IF_ELSE_DECLARATION ::= IF_STATEMENT ELIF_STATEMENT ELSE_STATEMENT

IF_STATEMENT ::= 'if' OPEN_PAREN BOOLEAN_EXPRESSION CLOSE_PAREN OPEN_CURLY COMMAND CLOSE_CURLY

ELIF_STATEMENT ::=  'elif' OPEN_PAREN BOOLEAN_EXPRESSION CLOSE_PAREN OPEN_CURLY COMMAND CLOSE_CURLY,
ELIF_STATEMENT | Null

ELSE_STATEMENT ::= 'else' OPEN_CURLY COMMAND CLOSE_CURLY | Null

BOOLEAN_EXPRESSION ::= EXPRESSION COMPARISION_OPERATOR EXPRESSION
                     | BOOLEAN_EXPRESSION BOOLEAN_OPERATOR BOOLEAN_EXPRESSION
                     | 'not' BOOLEAN_EXPRESSION
                     | BOOLEAN_VALUE
                     | OPEN_PAREN BOOLEAN_EXPRESSION CLOSE_PAREN

EXPRESSION_OPERATOR ::= '+' | '-' | '*' | '/'

EXPRESSION ::= EXPRESSION EXPRESSION_OPERATOR EXPRESSION
              | IDENTIFER ASSIGNMENT_OPERATOR EXPRESSION
              | OPEN_PAREN EXPRESSION CLOSE_PAREN
              | NUMBER
              | IDENTIFER
              | STACK_PRINT
              | QUEUE_PRINT

WHILE_LOOP ::= 'while' OPEN_PAREN BOOLEAN_EXPRESSION CLOSE_PAREN OPEN_CURLY COMMAND CLOSE_CURLY

FOR_LOOP ::= 'for' IDENTIFER 'in' 'range' OPEN_PAREN NUMBER ',' NUMBER CLOSE_PAREN OPEN_CURLY COMMAND
CLOSE_CURLY

FOR_LOOP ::= 'for' OPEN_PAREN IDENTIFER ASSIGNMENT_OPERATOR EXPRESSION ';' IDENTIFER
COMPARISION_OPERATOR EXPRESSION ';' IDENTIFER = EXPRESSION CLOSE_PAREN OPEN_CURLY COMMAND CLOSE_CURLY
```



```
PRINT ::= 'print' OPEN_PAREN PRINT_STATEMENT CLOSE_PAREN

PRINT_STATEMENT_LIST ::= Null
              | PRINT_STATEMENT

PRINT_STATEMENT ::= IDENTIFER PRINT_STATEMENT_LIST
              | STRING PRINT_STATEMENT_LIST
              | EXPRESSION PRINT_STATEMENT_LIST
              | STACK_PRINT
              | QUEUE_PRINT
STACK_OPERATIONS ::= STACK_PRINT
              | IDENTIFER '.' 'push' OPEN_PAREN EXPRESSION CLOSE_PAREN

STACK_PRINT ::= IDENTIFER '.' 'pop' OPEN_PAREN CLOSE_PAREN
              | IDENTIFER '.' 'top' OPEN_PAREN CLOSE_PAREN
QUEUE_OPERATIONS ::= QUEUE_PRINT
              | IDENTIFER '.' 'push' OPEN_PAREN EXPRESSION CLOSE_PAREN

QUEUE_PRINT ::= IDENTIFER '.' 'poll' OPEN_PAREN CLOSE_PAREN
              | IDENTIFER '.' 'head' OPEN_PAREN CLOSE_PAREN

LIST_OPERATIONS ::= IDENTIFER '.' 'add' OPEN_PAREN EXPRESSION CLOSE_PAREN
              | IDENTIFER '.' 'add' OPEN_PAREN EXPRESSION [,] EXPRESSION CLOSE_PAREN
              | IDENTIFER '.' 'remove' OPEN_PAREN EXPRESSION CLOSE_PAREN
              | IDENTIFER '.' 'get' OPEN_PAREN EXPRESSION CLOSE_PAREN

METHOD ::= METHOD_DECLARATION | METHOD_CALL
METHOD_DECLARATION ::= 'def' IDENTIFER OPEN_PAREN PARAMETER_LIST CLOSE_PAREN OPEN_CURLY METHOD_BODY
CLOSE_CURLY
METHOD_CALL ::= IDENTIFER OPEN_PAREN PARAMETER_LIST CLOSE_PAREN
METHOD_BODY ::- COMMAND
PARAMETER_LIST ::- IDENTIFER ',' PARAMETER_LIST | Null
```

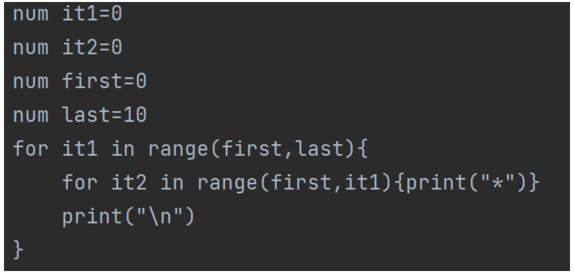
Compiler Design Process



Lexer

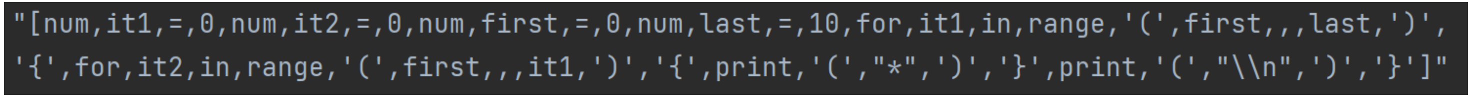
- Lexer reads the characters from source program and groups them into lexemes (sequence of characters that “go together”). Each lexeme corresponds to a token.

Lexer Input:



```
num it1=0
num it2=0
num first=0
num last=10
for it1 in range(first,last){
    for it2 in range(first,it1){print("*")}
    print("\n")
}
```

Lexer Output:



```
"[num,it1,=,0,num,it2,=,0,num,first,=,0,num,last,=,10,for,it1,in,range,'(',first,,,last,')',
'{',for,it2,in,range,'(',first,,,it1,')','{',print,'(',"*",')','}',print,'(',"\\n",')','}']"
```

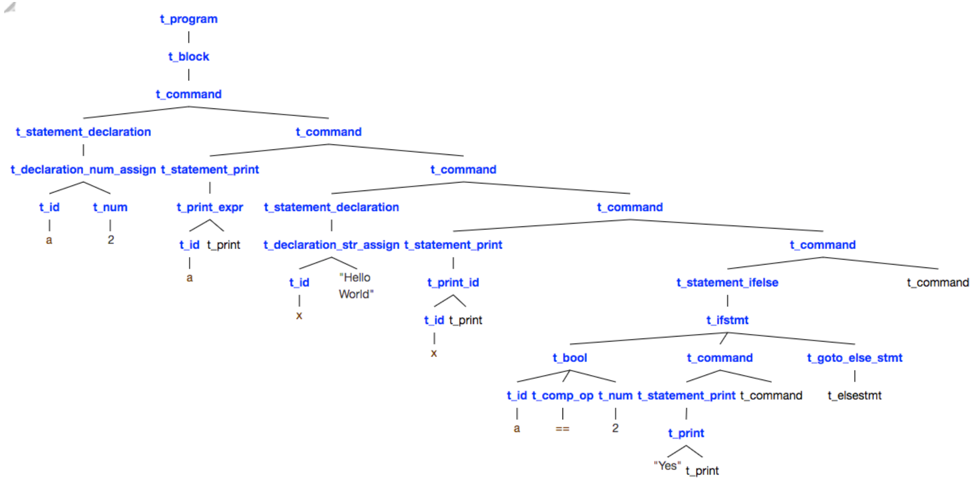
Parser

- It takes all the tokens one by one and constructs the parse tree.
- Symbol Table Data Structure and its Uses in Parser.

Symbol Table

Identifier	Type
x	num
y	stack
add	method

Parser Output



Semantics

- Giving meaning to the parse tree
- Symbol table Data Structure

  - [(Identifier, Value, Type)]
- O/P – Execution of I/P file

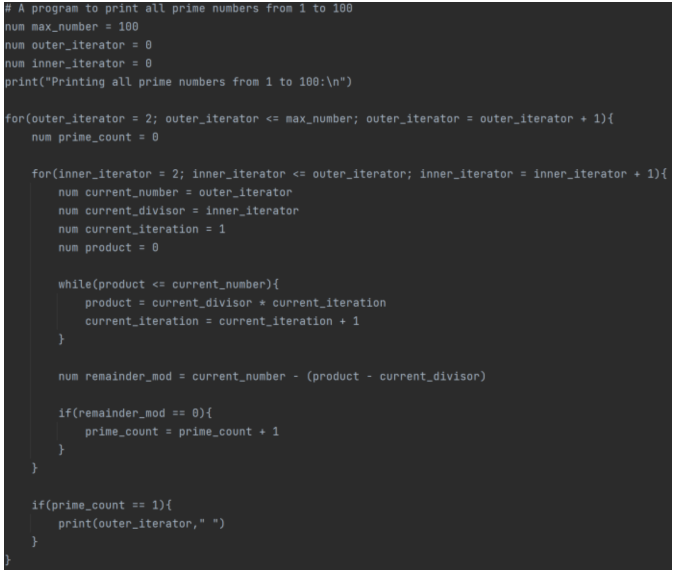
Identifier	Value	Type
x	5	num
A	[10,20]	stack
add	*((t_formal_parameter(t_id(x), t_formal_parameter(t_id(y), t_formal_parameter())),t_body(t_command(t_statement_print(t_print_expr(t_add(t_id(x), t_id(y)), t_print())), t_command())))))*	method

MyProgram.py

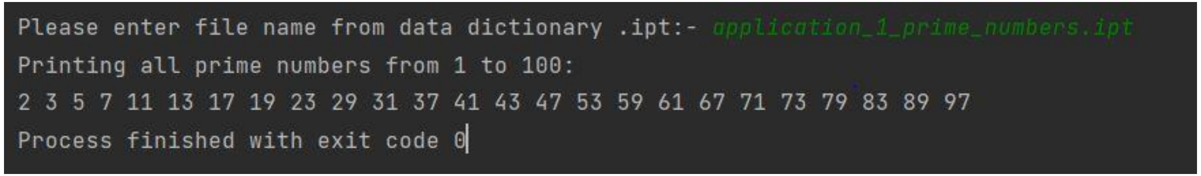
Running your main program

python main.py <inputfile.rch>

Snapshot of the demonstration of the language

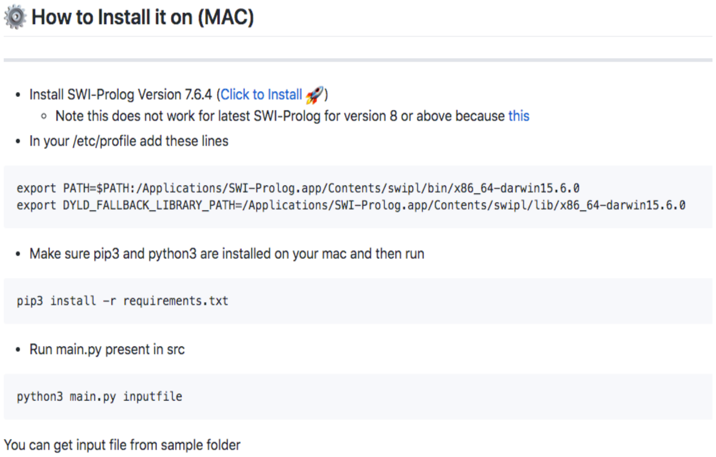


```
# A program to print all prime numbers from 1 to 100
num max_number = 100
num outer_iterator = 0
num inner_iterator = 0
print("Printing all prime numbers from 1 to 100:\n")

for(outer_iterator = 2; outer_iterator <= max_number; outer_iterator = outer_iterator + 1){
    num prime_count = 0

    for(inner_iterator = 2; inner_iterator <= outer_iterator; inner_iterator = inner_iterator + 1){
        num current_number = outer_iterator
        num current_divisor = inner_iterator
        num current_iteration = 1
        num product = 0

        while(product <= current_number){
            product = current_divisor * current_iteration
            current_iteration = current_iteration + 1
        }

        num remainder_mod = current_number - (product - current_divisor)

        if(remainder_mod == 0){
            prime_count = prime_count + 1
        }
    }

    if(prime_count == 1){
        print(outer_iterator," ")
    }
}
```



```
Please enter file name from data dictionary .ipt:- application_1_prime_numbers.ipt
Printing all prime numbers from 1 to 100:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Process finished with exit code 0
```

Installation Demonstration



⚙️ How to Install it on (MAC)

- Install SWI-Prolog Version 7.6.4 (Click to Install 🚀)
    - Note this does not work for latest SWI-Prolog for version 8 or above because this
- In your /etc/profile add these lines

```
export PATH=$PATH:/Applications/SWI-Prolog.app/Contents/swipl/bin/x86_64-darwin15.6.0
export DYLD_FALLBACK_LIBRARY_PATH=/Applications/SWI-Prolog.app/Contents/swipl/lib/x86_64-darwin15.6.0
```

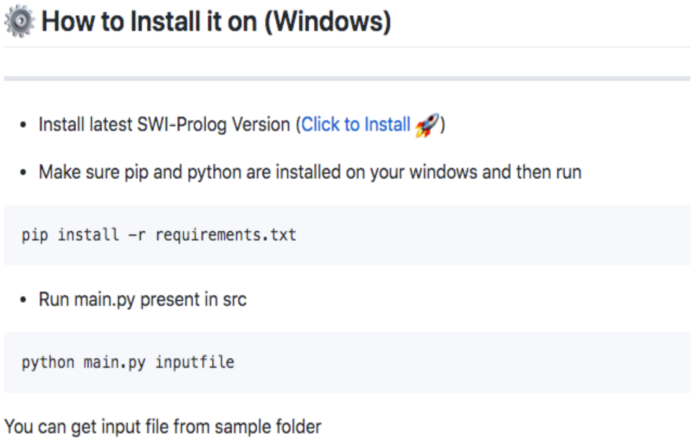
- Make sure pip3 and python3 are installed on your mac and then run

```
pip3 install -r requirements.txt
```

- Run main.py present in src

```
python3 main.py inputfile
```

You can get input file from sample folder



⚙️ How to Install it on (Windows)

- Install latest SWI-Prolog Version (Click to Install 🚀)
- Make sure pip and python are installed on your windows and then run

```
pip install -r requirements.txt
```

- Run main.py present in src

```
python main.py inputfile
```

You can get input file from sample folder

Future Scope

- Advanced Data Structure
- User Defined Data types
- Import Multiple files and functions
- Recursion
- Multi-threading

Sample Code Demonstration