



# SER 502 PROJECT SKRAV - Group 7

Rajat Yadav  
Varun Singh  
Abhishek Masetty  
Sahil Jambhulkar  
Konark Bhad



# IMPORTANT LINKS:

GITHUB: <https://github.com/raiat698/SER502-Spring2022-Team7>

YOUTUBE VIDEO LINK:



# ABOUT THE LANGUAGE

- The name of our Language is “SKRAV”. Which is derived from our names' initials.
- The extension of SKRAV is “.xxx”.
- Programing Languages used to make SKRAV are Python 3 and SWI-Prolog.

# COMPILATION PROCESS



It will include a program built in the SKRAV programming language syntax. The .xxx file extension will be used to save the source code file.

It is the compiler's first phase, which does lexical analysis. By removing any whitespace or comments from the source code, the lexical analyzer breaks these syntaxes down into a series of tokens. The lexical analyzer generates an error if a token is found to be incorrect.

It ensures that the source code adheres to the SKRAV language's stated syntax. A parse tree is created by reading tokens from the token list one by one. If all of the tokens aren't parsed, it throws a syntax error.

It accepts the parse tree as input and runs the program using operational and denotational semantics. Each node is parsed from the top down, with simultaneous evaluation.



# DETAILS

## Parsing Technique:

We used the same technique as the DCG rules in prolog, i.e., a collection of functions that consume tokens and recursively build the tree, and validate correctness by enforcing that all tokens are consumed.

## Data structures used:

The parse tree is implemented using classes specific to different nodes in the tree that have links to other nodes. Environment is implemented using hash maps (dictionary) with the identifier name as the key, and the value field consisting of the identifier's value, datatype and some other information associated with it.



# FEATURES:

Basic mathematical operations such as addition, subtraction, multiplication, division, and modulus are supported in our language.

Less than, greater than, less than equal, greater than equal are also assisted in our language.

The SKRAV language includes the feature of "print" followed by a space and a string to be written within the quotes to print any statement to the console in our language.

Precedence: Multiplication, division, and modulus operations are given more weight in the language than addition and subtraction operations.

Data Types: SKRAV supports different types of data type like: Int, string, float and boolean

Looping construct: while and for

Decision control statement: if-else, else if

Functions and nested functions are supported.

Syntactic sugar (+, -, \*, / =) is supported in our language.



# LEXICAL, ANALYSER AND PARSER

Removes spaces, tabs, and new lines from the input program file to create a list of tokens.

Tokens are passed to a parser.

The parser translates this to syntactic form using the grammar rules given, and then builds the syntax tree.

Parsed in a top down fashion



# EVALUATOR

The final component of our language design is the evaluator.

The evaluator will take the parse tree as input and evaluate the nodes in order to return the final answer to the input code.

The evaluator has been written in Python 3.

Hash maps (dictionaries) are used to implement the environment, with the identifier name as the key and the identifier's value and datatype as a value.

Evaluator is called recursively.





# GRAMMAR

# arithmetic expressions

$\langle \text{digit} \rangle := 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$\langle \text{number} \rangle := \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{number} \rangle$

$\langle \text{expr} \rangle := \langle \text{number} \rangle \mid \langle \text{id} \rangle \mid \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{expr} \rangle - \langle \text{expr} \rangle \mid \langle \text{expr} \rangle * \langle \text{expr} \rangle \mid \langle \text{expr} \rangle /$

$\langle \text{expr} \rangle \mid \langle \text{expr} \rangle \% \langle \text{expr} \rangle \mid ( \langle \text{expr} \rangle )$


# boolean expressions

$\langle \text{boolean} \rangle := \text{True} \mid \text{False}$

$\langle \text{bool\_expr} \rangle := \langle \text{boolean} \rangle \mid \langle \text{bool\_expr} \rangle \text{ and } \langle \text{bool\_expr} \rangle \mid \langle \text{bool\_expr} \rangle \text{ or } \langle \text{bool\_expr} \rangle \mid \text{not}$

$\langle \text{bool\_expr} \rangle \mid \langle \text{bool\_expr} \rangle == \langle \text{bool\_expr} \rangle \mid \langle \text{id} \rangle \langle \text{comparator} \rangle \langle \text{number} \rangle \mid \langle \text{id} \rangle == \langle \text{string} \rangle \mid ($   
 $\langle \text{bool\_expr} \rangle )$

$\langle \text{comparator} \rangle := > \mid < \mid \leq \mid \geq \mid ==$



## # strings and identifiers

<alphabet> := 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

<spec\_char> := ! | \ | # | \$ | % | & | \ | ( | ) | \* | + | , | - | . | / | : | ; | < | = | > | ? | @ | ~

<whitespace> := " "

<char> := <alphabet> | <digit> | <spec\_char> | <whitespace>

<char\_list> := <char> | <char><string>

<string> := " <char\_list> " | ""

<id> := <alphabet> | <alphabet><id\_suffix>

<id\_suffix> := <alphabet> | <digit> | <alphabet><id\_suffix> | <digit><id\_suffix>

## # declarations

<datatype> := int | str | bool

<stmt> := <datatype> <id> ; | <datatype> <id> = <value> ;

## # assignment

<value> := <expr> | <string> | <bool\_expr>

<stmt> := <id> = <value> ;



#if-else

<stmt> := <bool\_expr> ? <stmt> : <stmt>

<stmt> := if ( <bool\_expr> ) { <stmt> } else { <stmt> }

#for loop

<stmt> := for-loop(<stmt> <bool\_expr> ; <stmt>) { <stmt> }

<stmt>

:= for-loop(<id> in range(<number>, <number>)) { <stmt> }

#while loop


<stmt> := while( <bool\_expr> ) { <stmt> }

# print

<stmt> := display <id> ; | display <value> ;

# comment

<comment> := \$\$ <char\_list> \$\$



# function

<function> := func ( <parameter>\* ) { <stmt>\* };

#parameter

<parameter> := <datatype> <id>

#program

<program> := <func>\* shuru <stmt>\* khatam



THANK YOU