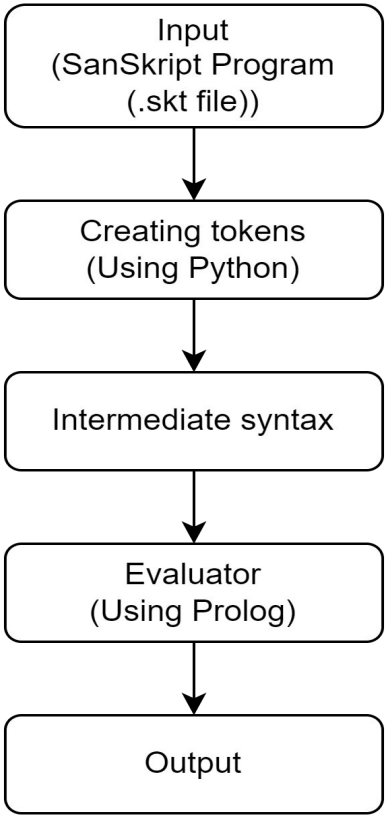
SanSkript Team 25

Team Members:

Milind Deshpande
Rushabh Gajab
Siddharth Sharma
Viresh Bhurke

Basic Execution Flow



Data Types supported in SanSkript

- int
- float
- boolean
- string

Conditional statements in SanSkript

1. If-then
2. If-then-else

Example program in SanSkript:

```
aarambh
int c = 3 ||
if (c > 5)
then (likhyam("c is greater than 5")||)
else (likhyam("c is less than 5")||)||
antah
```

Loops(Iterative Statements) in SanSkript

1. While loop
2. Traditional for loop
3. For in range loop

Example program in SanSkript (While loop):

```
aarambh
int b = 5 ||
while(b>0)
(
   likhyam(b)||
   b--||
)||
antah
```

Loops(Iterative Statements) in SanSkript

Example program (Traditional for loop):

```
aarambh
int i = 5 ||
for(i=0 || i<9 || i++) (
   likhyam(i)||
)||
antah
```

Example program (for in range loop):

```
aarambh
int i=1||
for g in range(1 -> 7)(
   likhyam(i)||
   i=i+1 ||
) ||
antah
```

Operators supported in SanSkript

➔ **Arithmetic Operators**:

- Addition
- Subtraction
- Multiplication
- Division

➔ **Logical Operators**:

- and
- or
- not

Operators supported in SanSkript

➔ **Comparison Operators:** This is in addition to the basic language design requirements as defined in the project deliverable document**.**

- greater than: (>)
- greater than or equal to: (>=)
- less than: (<)
- less than or equal to: (<=)
- equal to: (==)
- not equal to: (=/=)

Tools used for the project

- Python3
- Prolog(SWI Prolog)

SanSkript Grammar Snippet

```
:- discontiguous variable_declaration/3, i/3, conditional_block/3.
:- discontiguous statement/3, evaluator/3,condition/3.
%:- use_rendering(svgtree).

% Helper member predicate to check in list
mem(H,[H|_]).
mem(H,[_|T]):- mem(H,T).

%Program
program(prgm(B)) --> block(B).

%Block
block(blk(aarambh, D, '||', S, antah)) --> [aarambh], declarations(D),
statements(S), [antah].

% Declarations
declaration(decl(V)) --> variable_declaration(V).
declaration(decl(S)) --> string_declaration(S).
declarations(decls(V)) --> declaration(V).
declarations(decls(V,Vs)) --> declaration(V), declarations(Vs).
%string
%different statement rule defined.
statements(stats(S)) --> statement(S),['||'].
statements(stats(S,Ss)) --> statement(S),['||'],statements(Ss).
statement(cond_blk(S)) --> conditional_block(S).
%statements(stats(S,Ss)) --> conditional_block(S),statements(Ss).
%Added loops to statements.
statement(S) --> loops(S).
%statements(stats(S,Ss)) --> loops(S),statements(Ss).

statement(stat(S)) --> assignment(S).
statement(stat(S)) --> increment_operation(S).


% Print Statement
statement(print_stmt(S)) --> print_statement(S).

% create loops for nested
loops(lps(S,Ss)) --> loop(S), loops(Ss).
loops(L) --> loop(L).
% create loop
loop(lp(S)) --> traditional_whileloop(S).
%created for loop and for in range loop.
loop(lp(S)) --> traditional_forloop(S).
loop(lp(S)) --> range_forloop(S).
```

SanSkript Grammar Snippet(Contd..)

```
%conditional block
conditional_block(cond_blk(S)) --> if_then_block(S).
if_then_block(if_then_blk(Condition,S)) -->
[if],['('],condition(Condition),[')'],[then],['('],statements(S),[')'].
if_then_block(if_then_blk(Bool,S)) -->
[if],['('],bool(Bool),[')'],[then],['('],statements(S),[')'].

%conditional block
conditional_block(cond_blk(S)) --> if_then_else_block(S).
if_then_else_block(if_then_else_blk(Condition,S1,S2)) -->
[if],['('],condition(Condition),[')'],[then],['('],statements(S1),[')'],[else
],['('],statements(S2),[')'].
if_then_else_block(if_then_else_blk(Bool,S1,S2)) -->
[if],['('],bool(Bool),[')'],[then],['('],statements(S1),[')'],[else],['('],state
ments(S2),[')'].
```

```
traditional_whileloop(trd_while_blk(while,Condition,Ss)) -->
[while],['('],condition(Condition),[')'],['('],statements(Ss),[')'].
traditional_whileloop(trd_while_blk(while,Bool,Ss)) -->
[while],['('],bool(Bool),[')'],['('],statements(Ss),[')'].
% rules for traditional for loop and for in range loop.
traditional_forloop(trd_for_blk(for,I,V,Condition,I,Op,Ss)) -->
[for],['('],identifier(I),[=],value(V),['||'],condition(Condition),['||'],increment_
operation(Op),[')'],['('],statements(Ss),[')'].
traditional_forloop(trd_for_blk(for,I,V,Bool,I,Op,Ss)) -->
[for],['('],identifier(I),[=],value(V),['||'],bool(Bool),['||'],increment_operation(
Op),[')'],['('],statements(Ss),[')'].

range_forloop(rng_for_loop(for,I,in,range,N,->,M,Ss)) -->
[for],identifier(I),[in],[range],['('],value(N),[->],value(M),[')'],['('],statements(S
s),[')'].

%Ternary Operator:
conditional_block(cond_blk(S)) --> ternary_operator_block(S).
ternary_operator_block(tern_op_blk(Condition,?,S1,:,S2)) -->
condition(Condition),['?'],statements(S1),[':'],statements(S2).
ternary_operator_block(tern_op_blk(Bool,?,S1,:,S2)) -->
bool(Bool),['?'],statements(S1),[':'],statements(S2).
```

SanSkript Grammar Snippet(Contd..)

```
%Increment Operations
increment_operation(incr_op(I,++)) --> identifier(I),[++].
increment_operation(incr_op(I,--)) --> identifier(I),[--].

condition(cond(E1,Ri,E2))-->
expression(E1),relational_identifier(Ri),expression(E2).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%
condition(cond(true)) --> [true].
condition(cond(false)) --> [false].
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%

%Relational operators
relational_identifier(<) --> [<].
relational_identifier(<=) --> [<=].
relational_identifier(>) --> [>].
relational_identifier(>=) --> [>=].
relational_identifier(==) --> [==].
% using =/= imstead of != as it is giving oprerator
relational_identifier(=/=) --> [=/=].
```

```
%new
bool(bool_t(C1,and,C2))--> condition(C1),[and],condition(C2).
bool(bool_t(C1,or,C2))--> condition(C1),[or],condition(C2).
condition(cond(not,A))--> [not], condition(A).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%

% Print Statements For evaluation of expressions and Strings.
print_statement(print_stmt(P)) --> [likhyam],['('],expression(P),[')'].
print_statement(print_stmt(W)) --> print_statement_word(W).
print_statement_word(print_stmt_Word(X)) --> [likhyam, '(', X, ')'].

%terms for identifiers and values
term(term(I)) --> i(I).
term(val(N)) --> n(N).
%Handling brackets in terms
term(A) --> ['('], expression(A), [')'].
```

SanSkript Grammar Snippet(Contd..)

```
%Expressions defined
expression(E) --> expr_add_sub(E).
%Expression for addition and subtraction
expr_add_sub(A) --> term(A).
expr_add_sub(store(I,=,E)) --> expr_mul_div(I), [=], expression(E).
expr_add_sub(sub(A, -, B)) --> expr_mul_div(A), [-], expression(B).
expr_add_sub(add(A, +, B)) --> expr_mul_div(A), [+], expression(B).
expr_add_sub(A) --> expr_mul_div(A).
% Giving priority to multiplication and division
expr_mul_div(A) --> term(A).
expr_mul_div(mul(A, *, B)) --> term(A), [*], expr_mul_div(B).
expr_mul_div(div(A, /, B)) --> term(A), [/], expr_mul_div(B).

% Declared initial to initialize
i(store(a,=,E)) --> [a],[=], expression(E).
i(store(b,=,E)) --> [b],[=], expression(E).
i(store(c,=,E)) --> [c],[=], expression(E).
i(store(d,=,E)) --> [d],[=], expression(E).
i(store(e,=,E)) --> [e],[=], expression(E).
i(store(f,=,E)) --> [f],[=], expression(E).
i(store(g,=,E)) --> [g],[=], expression(E).
i(store(h,=,E)) --> [h],[=], expression(E).
i(store(i,=,E)) --> [i],[=], expression(E).
i(store(j,=,E)) --> [j],[=], expression(E).
i(store(k,=,E)) --> [k],[=], expression(E).
i(store(l,=,E)) --> [l],[=], expression(E).
i(store(m,=,E)) --> [m],[=], expression(E).
i(store(n,=,E)) --> [n],[=], expression(E).
i(store(o,=,E)) --> [o],[=], expression(E).
i(store(p,=,E)) --> [p],[=], expression(E).
i(store(q,=,E)) --> [q],[=], expression(E).
i(store(r,=,E)) --> [r],[=], expression(E).
i(store(s,=,E)) --> [s],[=], expression(E).
i(store(t,=,E)) --> [t],[=], expression(E).
i(store(u,=,E)) --> [u],[=], expression(E).
i(store(v,=,E)) --> [v],[=], expression(E).
i(store(w,=,E)) --> [w],[=], expression(E).
i(store(x,=,E)) --> [x],[=], expression(E).
i(store(y,=,E)) --> [y],[=], expression(E).
i(store(z,=,E)) --> [z],[=], expression(E).
```

SanSkript Grammar Snippet(Contd..)

identifier(I) --> i(I).

i(a) --> [a].
i(b) --> [b].
i(c) --> [c].
i(d) --> [d].
i(e) --> [e].
i(f) --> [f].
i(g) --> [g].
i(h) --> [h].
i(i) --> [i].
i(j) --> [j].
i(k) --> [k].
i(l) --> [l].
i(m) --> [m].
i(n) --> [n].
i(o) --> [o].
i(p) --> [p].
i(q) --> [q].
i(r) --> [r].
i(s) --> [s].
i(t) --> [t].
i(u) --> [u].
i(v) --> [v].
i(w) --> [w].
i(x) --> [x].
i(y) --> [y].
i(z) --> [z].

SanSkript Grammar Snippet(Contd..)

```
word(S) --> [S], {atom(S)}.

% Define numbers. Made easier for parsing in tree.
value(V) --> n(V).
n(0) --> [0].
n(1) --> [1].
n(2) --> [2].
n(3) --> [3].
n(4) --> [4].
n(5) --> [5].
n(6) --> [6].
n(7) --> [7].
n(8) --> [8].
n(9) --> [9].
n(N) --> [N], {integer(N)}.
```

```
%Variable Declaration and Assignment
variable_declaration(var_decl(T,I,=,V))--> type(T), identifier(I), [=],
value(V),['||'].
%assignment
assignment(assig(I,=,E))--> identifier(I), [=], expression(E).
assignment(assig(I,=,W))--> identifier(I), [=], word(W).
%string
string_declaration(str_decl(_,I,=,W))-->
[string],identifier(I),[=],['('],word(W),[')'],['||'].

%Declaring datatypes

type(typ(int))--> [int].
type(typ(flt))--> [float].
type(typ(vrbl))--> [bool].
type(typ(str))--> [string].
```

Tokenization

- The SanSkript program is broken down into tokens.
- Tokenization process is done in Python.
- This tokenized output is given to the parser.

Input program:

```
aarambh
int i = 10||
likhyam("Value of i: ")||
likhyam(i)||
antah
```

Output tokens:

```
aarambh
int
i
=
10
||
likhyam
(
"Value of i: "
)
||
likhyam
(
i
)
||
antah
```

Parsing tokens in Prolog

- The parser checks the grammar of the language from the input tokens provided to it.
- These tokens are then converted to an intermediate form for further processing.
- The parser is implemented in Prolog and DCG is used for checking the grammar.

Input tokens:

```
aarambh
int
i
=
10
||
likhyam
(
"Value of i: "
)
||
likhyam
(
i
)
||
antah
```

Output intermediate form:

[aarambh, int, i, =, 10, '||', likhyam, '(', "Value of i: ", ')', '||', likhyam, '(', i, ')', '||', antah]

Tokenize function

```
import re
import os
# Token specifications
token_specification = [
    ('SINGLE_QUOTE_STRING', r"'[^']*'"), # Single quoted string
    ('INCREMENT',  r'\+\+'),         # Increment operator
    ('DECREMENT',  r'--'),           # Decrement operator
    ('ID',         r'[A-Za-z]+'),    # Identifiers
    ('GTE',        r'>='),           # Greater than or equal to
    ('LTE',        r'<='),           # Less than or equal to
    ('ARROW',      r'->'),           # Arrow operator
    ('QUESTION',   r'\?'),           # Question mark for conditional
operations
    ('GT',         r'>'),            # Greater than
    ('LT',         r'<'),            # Less than
    ('EQ',         r'=='),           # Equal to
    ('NEQ',        r'=/='),          # Not equal to
    ('STRING',     r'"[^"]*"'),      # Double quoted string
    ('NUMBER',     r'\d+(\.\d*)?'),  # Integer or decimal number
    ('ASSIGN',     r'='),            # Assignment operator
    ('END',        r'\|\|'),         # Statement terminator '||'
    ('OP',         r'[+\-*/]'),      # Arithmetic operators
    ('BOOL_OP',    r'and|or|not'),   # Boolean operators
    ('BOOL',       r'true|false'),   # Boolean literals
    ('LPAREN',     r'\('),           # Left parenthesis
    ('RPAREN',     r'\)'),           # Right parenthesis
    ('LANG_START', r'aarambh'),      # Language start
    ('LANG_END',   r'antah'),        # Language end
    ('PRINT',      r'likhyam'),      # Print keyword
    ('COLON',      r':'),            # Colon, commonly used in ternary
operations or other constructs
    ('SKIP',       r'[ \t]+'),       # Skip over spaces and tabs
    ('NEWLINE',    r'\n'),           # Line endings
    ('MISMATCH',   r'.'),            # Any other character
]

# Building the regex
tok_regex = '|'.join('(?P<%s>%s)' % pair for pair in
token_specification)
```

Tokenize function

```
def tokenize(file_path):
    """
    Tokenizes the input file based on the predefined token specifications and writes the output to a file.
    """
    tokenized_output = []  # List to collect tokens
    with open(file_path, 'r') as file, open('tokens_output.txt', 'w') as token_file:
        code = file.read()
        line_num = 1
        for mo in re.finditer(tok_regex, code):
            kind = mo.lastgroup
            value = mo.group()
            if kind == 'NEWLINE' or kind == 'SKIP':
                line_num += 1 if kind == 'NEWLINE' else 0
                continue
            if kind == 'MISMATCH':
                raise RuntimeError(f'{value!r} unexpected on line {line_num}')
```

```
# Only put specific tokens in single quotes
            if kind == 'END':
                tokenized_output.append("'||'")
            elif kind == 'LPAREN':
                tokenized_output.append("'('")
            elif kind == 'RPAREN':
                tokenized_output.append("')'")
            elif kind == 'QUESTION':
                tokenized_output.append("'?'")
            elif kind == 'COLON':
                tokenized_output.append("':'")
            elif kind == 'SINGLE_QUOTE_STRING':
                tokenized_output.append("'")
            else:
                tokenized_output.append(value)
            token_file.write(f'{value}\n')

    return tokenized_output
```

Evaluator

```
% Return answer for evaluation

% Evaluator for evaluating
evaluator(Expr, Substitutions, Ans) :-
    evaluation(Expr, Substitutions, Ans).

% Check tree for substitutions
evaluation(term(I), Substitutions, Variable) :-
    mem((I, Variable), Substitutions).
evaluation(val(Value), _, Value).

% Addition operation
evaluation(add(Exp1, +, Exp2), Substitutions, Ans) :-
    evaluation(Exp1, Substitutions, Exp1_Ans),
    evaluation(Exp2, Substitutions, Exp2_Ans),
    Ans is Exp1_Ans + Exp2_Ans.

% Subtraction operation
evaluation(sub(Exp1, -, Exp2), Substitutions, Ans) :-
    evaluation(Exp1, Substitutions, Exp1_Ans),
    evaluation(Exp2, Substitutions, Exp2_Ans),
    Ans is Exp1_Ans - Exp2_Ans.

% Multiplication Operation
evaluation(mul(Exp1, *, Exp2), Substitutions, Ans) :-
    evaluation(Exp1, Substitutions, Exp1_Ans),
    evaluation(Exp2, Substitutions, Exp2_Ans),
    Ans is Exp1_Ans * Exp2_Ans.

% Division Operation
evaluation(div(Exp1, /, Exp2), Substitutions, Ans) :-
    evaluation(Exp1, Substitutions, Exp1_Ans),
    evaluation(Exp2, Substitutions, Exp2_Ans),
    Ans is Exp1_Ans / Exp2_Ans.

eval_var_declaration(var_decl(_, I, _, V), Env, NewEnv) :-

    % Add constant to environment
    NewEnv = [(I, V) | Env].
```

Evaluator

```
eval_str_declaration(str_decl(_,I,_,W), Env, NewEnv) :-
   % Add constant to environment
   NewEnv = [(I, W) | Env].

eval_decl(decl(V), Env, NewEnv) :-
   eval_var_declaration(V, Env, NewEnv).

eval_decl(decl(V), Env, NewEnv) :-
   eval_str_declaration(V, Env, NewEnv).

eval_decls(decls(V), Env, FinEnv) :-
   eval_decl(V, Env, FinEnv).

eval_decls(decls(V,Vs), Env, FinEnv) :-
   eval_decl(V, Env, Env1),
   eval_decls(Vs, Env1, FinEnv).

eval_assignment(assig(I,=,E),Env,NewEnv):-
   evaluator(E, Env, R),
   update(I,R,Env,NewEnv).
```

```
eval_assignment(assig(I,=,W),Env,NewEnv):-
   update(I,W,Env,NewEnv).

eval_stat(stat(S), Env, FinEnv):- eval_assignment(S, Env, FinEnv).
eval_stat(stat(S), Env, FinEnv):- eval_increment_operation(S, Env,
FinEnv).
eval_stat(print_stmt(S),Env,Env):- eval_print_statement(S,Env).
eval_stat(S,Env, FinEnv):- eval_loops(S,Env,FinEnv).
eval_stat(cond_blk(S),Env,FinEnv):-
eval_conditional_block(S,Env,FinEnv).

eval_stats(stats(S), Env, FinEnv):- eval_stat(S, Env, FinEnv).
eval_stats(stats(S,Ss), Env, FinEnv):- eval_stat(S, Env, Env1),
eval_stats(Ss, Env1, FinEnv).

eval_bool(cond(true), _, true).
eval_bool(cond(false), _, false).
```

Evaluator

```
eval_bool(cond(E1, ==, E2), Env, Result) :-
    % Evaluate the expressions
    evaluator(E1, Env, Val1),
    evaluator(E2, Env, Val2),
    % Check if the expressions are equal
    (Val1 =:= Val2 -> Result = true ; Result = false).

eval_bool(cond(E1, >=, E2), Env, Result) :-
    % Evaluate the expressions
    evaluator(E1, Env, Val1),
    evaluator(E2, Env, Val2),
    % Check if the expressions are equal
    (Val1 >= Val2 -> Result = true ; Result = false).

eval_bool(cond(E1, <=, E2), Env, Result) :-
    % Evaluate the expressions
    evaluator(E1, Env, Val1),
    evaluator(E2, Env, Val2),
    % Check if the expressions are equal
    (Val1 =< Val2 -> Result = true ; Result = false).
```

```
eval_while_loop(trd_while_blk(while,Condition,Ss),Env1,NewEnv).
eval_while_loop(trd_while_blk(while,Condition,_),Env,Env):-
    eval_bool(Condition,Env,false).

eval_increment_operation(incr_op(I,++),Env,NewEnv) :-
lookup(I,Env,V), K is V+1,
    update(I,K,Env,NewEnv).

eval_increment_operation(incr_op(I,--),Env,NewEnv) :-
lookup(I,Env,V), K is V-1,
    update(I,K,Env,NewEnv).

eval_traditional_forloop(trd_for_blk(for,I,V,Condition,I,Op,Ss),Env,
FinEnv):- update(I,V,Env,Env1), eval_bool(Condition,Env1,true),
    eval_stats(Ss, Env1,
Env2),eval_increment_operation(Op,Env2,Env3), lookup(I,Env3,V1),
eval_traditional_forloop(trd_for_blk(for,I,V1,Condition,I,Op,Ss),Env
3, FinEnv).
eval_traditional_forloop(trd_for_blk(for,I,V,Condition,I,_,_),Env,
Env):- update(I,V,Env,Env1), eval_bool(Condition,Env1,false).
```

Evaluator

```
eval_loop(lp(S),Env,NewEnv):-
eval_traditional_forloop(S,Env,NewEnv).
eval_loop(lp(S),Env,NewEnv):- eval_while_loop(S,Env,NewEnv).
eval_loop(lp(S),Env,NewEnv):- eval_range_for(S,Env,NewEnv).

eval_loops(S,Env,NewEnv):- eval_loop(S,Env,NewEnv).
eval_loops(lps(S,Ss),Env,NewEnv):- eval_loop(S,Env,Env1),
eval_loops(Ss,Env1,NewEnv).
```

```
%conditional_block(cond_blk(S)) --> if_then_else_block(S).
%if_then_else_block(if_then_else_blk(Condition,S1,S2)) -->
[if],['('],condition(Condition),[')'],[then],['('],statements(S1),[')'],[else
],['('],statements(S2),[')'].
eval_if_then_else_block(if_then_else_blk(Condition,S1,_),Env,New
Env):-
   eval_bool(Condition,Env,true), eval_stats(S1, Env, NewEnv).
eval_if_then_else_block(if_then_else_blk(Condition,_,S2),Env,New
Env):-
   eval_bool(Condition,Env,false), eval_stats(S2, Env, NewEnv).
```

Evaluator

```
eval_conditional_block(cond_blk(S),Env,NewEnv):-
eval_if_then_block(S, Env, NewEnv).
eval_conditional_block(cond_blk(S),Env,NewEnv):-
eval_if_then_else_block(S, Env, NewEnv).
eval_conditional_block(cond_blk(S),Env,NewEnv):-
eval_tern_operator(S, Env, NewEnv).

eval_block(blk(aarambh, D, '||', S, antah),Env,FinEnv):-
eval_decls(D, Env, Env1), eval_stats(S, Env1, FinEnv).

eval_program(prgm(B),Env,FinEnv):- eval_block(B,Env,FinEnv).

%evaluator for ternary operator conditional block:
eval_tern_operator(tern_op_blk(Condition,?,S1,:,_),Env,FinEnv):-
   eval_bool(Condition,Env,true),eval_stats(S1,Env,FinEnv).
eval_tern_operator(tern_op_blk(Condition,?,_,:,S2),Env,FinEnv):-
   eval_bool(Condition,Env,false),eval_stats(S2,Env,FinEnv).
```

```
eval_print_statement_word(print_stmt_Word(X),Env):-
mem((X,_),Env),lookup(X,Env,V),write(V).
eval_print_statement_word(print_stmt_Word(X),Env):- \+
mem((X,_),Env),write(X).
eval_print_statement(print_stmt(W),Env):-
eval_print_statement_word(W,Env).
eval_print_statement(print_stmt(W),Env):- evaluator(W, Env, Ans),
write(Ans),nl.

%Update value in Environment
update(Id, Val, [], [(Id,Val)]).
update(Id, Val, [(Id,_)|T], [(Id,Val)|T]).
update(Id, Val, [H|T], [H|R]) :- H\=(Id,_),update(Id,Val,T,R).
%Lookup value in environment
lookup(_,[],_).
lookup(Id,[(Id,Val)|_],Val).
lookup(Id,[_|T],Val):- lookup(Id,T,Val).
```

Evaluating expressions in Prolog

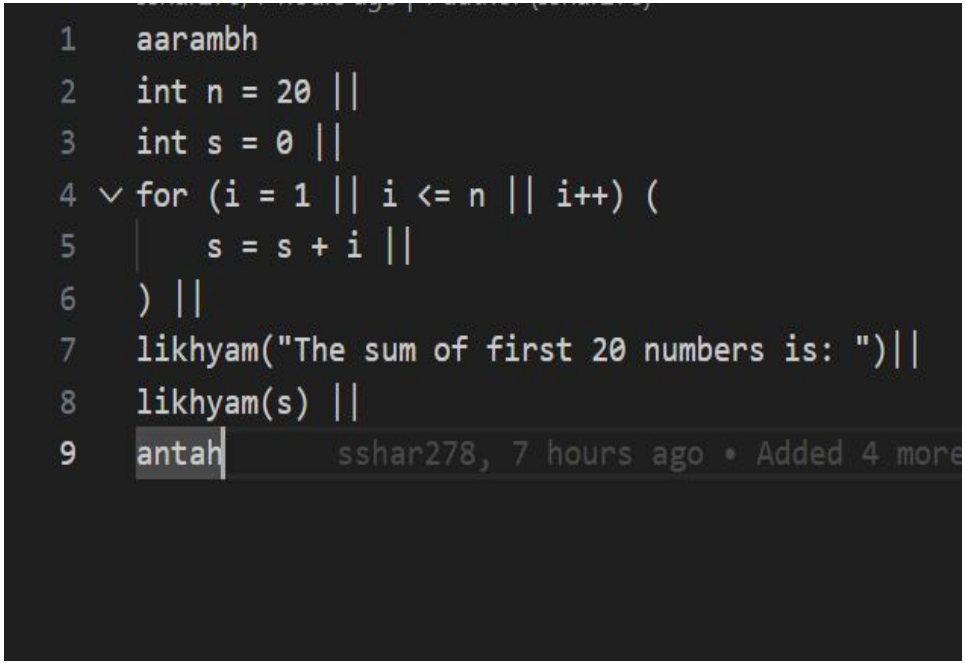
- The evaluators work using pattern matching.
- It takes the input as expressions and produces resulting values and a set of values.
- This is implemented in DCG and Prolog.

➔ **Key concepts**:

1. Lookup table
2. Update table

Program Execution screenshots

SanSkript Program



```
aarambh
int n = 20 ||
int s = 0 ||
for (i = 1 || i <= n || i++) (
    s = s + i ||
) ||
likhyam("The sum of first 20 numbers is: ")||
likhyam(s) ||
antah
```

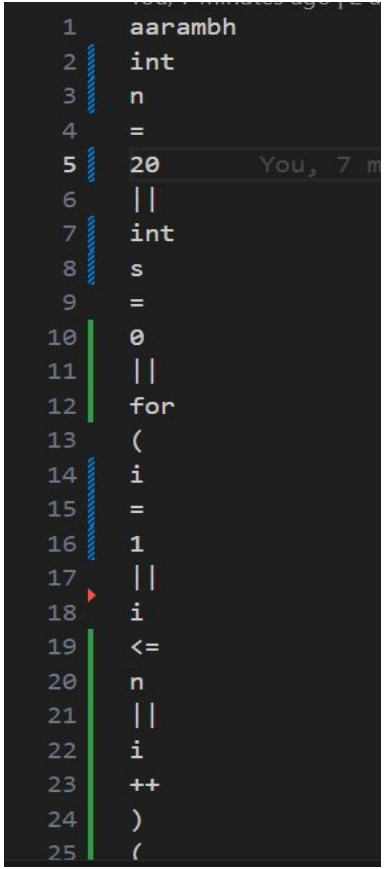
sshar278, 7 hours ago • Added 4 more

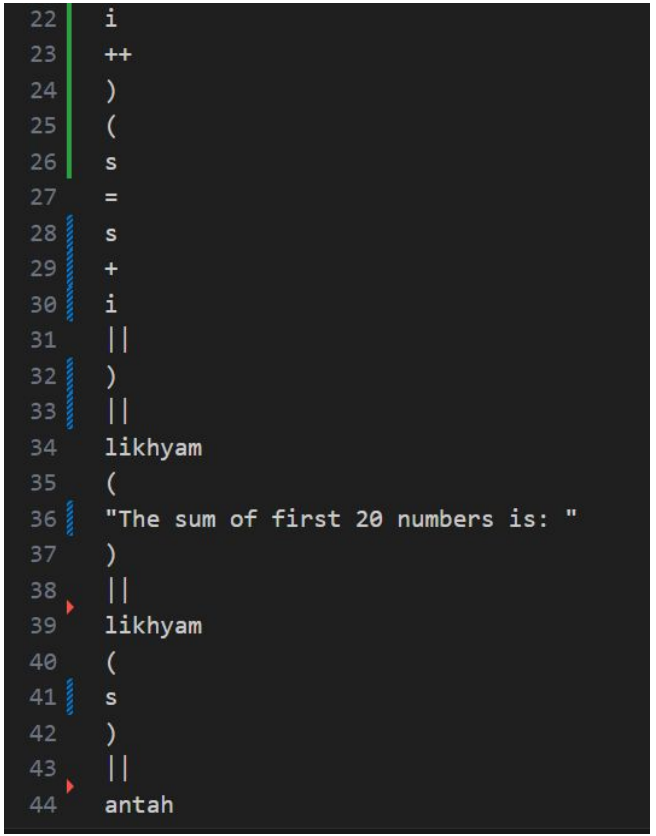
Terminal commands

```
C:\Users\milin\OneDrive\Documents\SER502 - Languages and Programming Paradigms\Project\SER502-SanSkript-
Team25>python runskript.py natural_nums.skt
The sum of first 20 numbers is: 210

Execution completed.
```

Intermediate token files





Intermediate syntax

src > runtime > temp > natural_nums.sktc

sshar278, 8 hours ago | 1 author (sshar278)

```
1  [aarambh, int, n, =, 20, '||', int, s, =, 0, '||', for, '(', i, =, 1, '||', i, <=, n, '||', i, ++,
   ')', '(', s, =, s, +, i, '||', ')', '||', likhyam, '(', "The sum of first 20 numbers is: ", ')', '||
   ', likhyam, '(', s, ')', '||', antah]      sshar278, 8 hours ago • Added 4 more sample programs
```

Future Scope

- Reading input provided by the user.
- Support for writing comments.
- More datatypes like long, double.
- Support for string operations like append, find a character, reverse.
- Support for functions in SanSkript.
- Support for Data Structures.

THANK YOU