SER 502 - Team 10

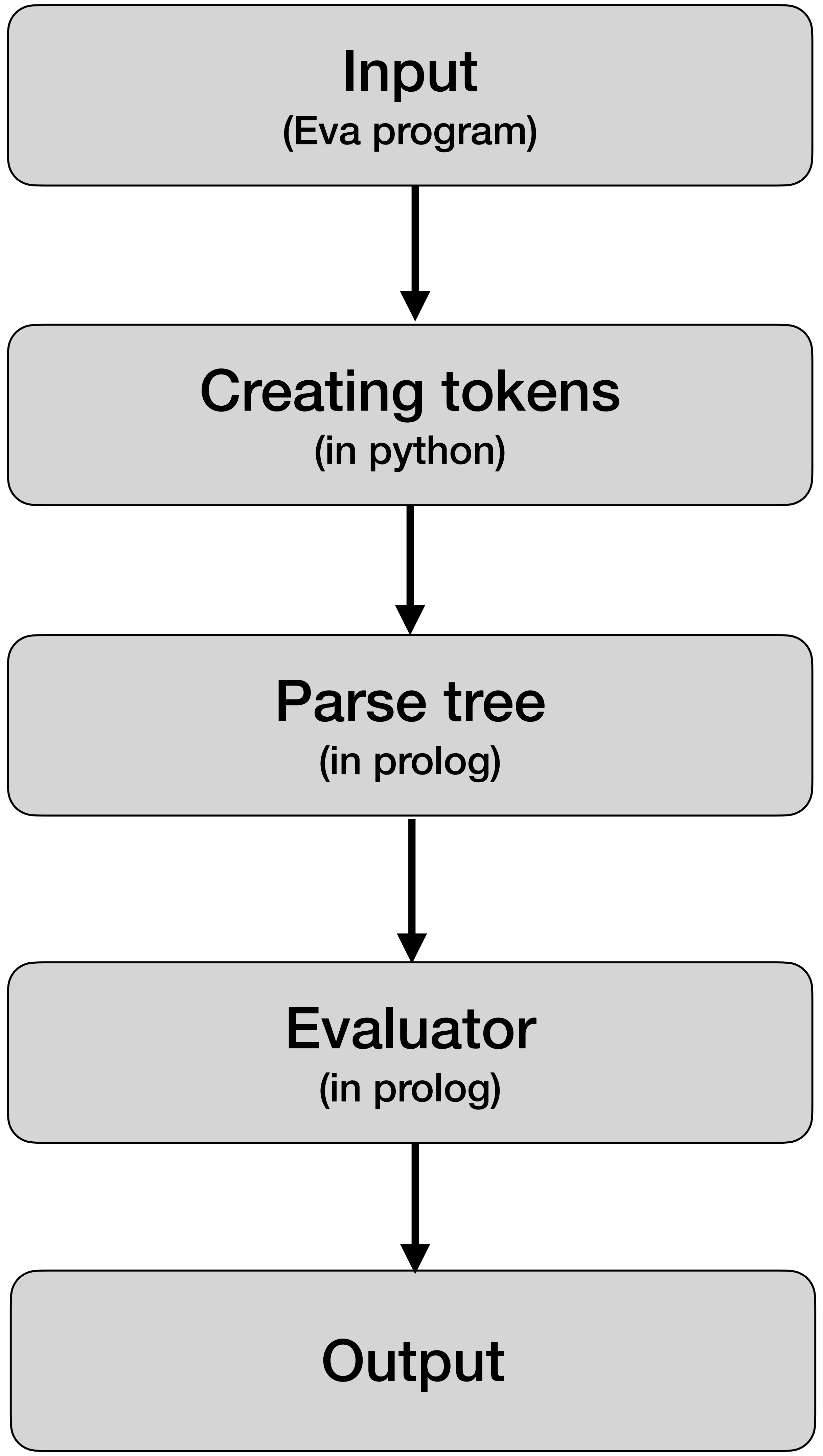
EVA

Sanket Surendra Kapse
Soham Prabhakar Patil

Sanika Yatin Gandhe
Sambhav Kapoor

Umang Sahastransu

Basic flow of execution



Input
(Eva program)

Creating tokens
(in python)

Parse tree
(in prolog)

Evaluator
(in prolog)

Output

Data Types in EVA

1. int

2. string

3. bool

Conditional Statements in EVA

1. If

2. If..else

Example in Eva:

```
program {
    int x;
    x = 12;
    if (x < 10) {
        print("Less than 10");
    } else if (x > 10) {
        print("Greater than 10");
    } else {
        print("Equal to 10");
    }
}
```

Iterative statements in EVA

1. for loop

2. while loop

3. for _ in range

Example in Eva:

```
program {
    int x;
    x = 6;
    int i;
    int res;
    res = 1;
    for i in range(1; x) {
        res = res * i;
    }
    print("Factorial of 5 is", res);
}
```

Operators in EVA

1. Arithmetic Operators

- Multiplication - (*)
- Division - (/)
- Addition - (+)
- Subtraction - (-)

2. Logical Operators

- AND - (&&)
- OR - (II)

Operators in EVA

3. Comparison operators

- greater than (>)
- greater than or equal to (>=)
- less than - (<)
- less than or equal to - (<=)
- equal to (==)
- Not equal to (!=)

Tools used

- Python3
- Prolog (SWIPL)

Grammar snippet

```
:- table block/2.
:- table expression/2.
:- table term/2.
:- table factor/2.

:- table block/3.
:- table condition/3.

program --> [program], ['{'], ['}'].
program --> [program], ['{'], block , ['}'].

% <block> ::= <statement> | <statement> <block>
block --> statement.
block --> block, statement.

% <statement> ::= <declaration> ";" | <assignment> ";" |
<if_statement> |
% <while_loop> | <for_loop> | <for_range> | <print_statement> ";"
statement --> declaration, [;].
statement --> assignment, [;].
statement --> if_statement.
statement --> while_loop.
statement --> for_loop.
statement --> for_range.
statement --> print_statement, [;].

% <declaration> ::= <type> <variable>
declaration --> type, variable.
```

```
% <type> ::= "int" | "string" | "bool"
type --> [int].
type --> [string].
type --> [bool].

% <variable> ::= <identifier> | <assignment>
variable --> identifier.
variable --> assignment.

% <assignment> ::= <identifier> "=" <expression> | <identifier> "="
<ternary>
assignment --> identifier, [=], expression.
assignment --> identifier, [=], string.
assignment --> identifier, [=], ternary.
assignment --> identifier, [++].
assignment --> identifier, [--].

% <identifier> ::= ^[_a-zA-Z][_a-zA-Z0-9]*
% <string> ::= "'" [a-zA-Z0-9_@!.,\s]* "'"
% <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
% <integer> ::= <digit> | <digit> <integer>
% <float> ::= <integer> "." <integer>
% <bool> ::= True | False | <condition>
identifier --> [I], {atom(I), \+ member(I, [program, for, if, else, for,
while, range, print, int, float, char, string, bool, in])}.
string --> [S], {atom(S)}.
integer --> [N], {integer(N)}.
boolean --> [true] | [false].

% <ternary> ::= <condition> "?" <expression> ":" <expression>
ternary --> condition, [?], expression, [:], expression.
```

Grammar snippet

```
% <expression> ::= <expression> + <expression> | <expression> -
<expression>
% | <expression> * <expression> | <expression> / <expression>
% | (<expression>) | <integer> | <float> | <identifier> |
<identifier> "++"
% | <identifier> "--" | <string>

expression --> expression, [+], term.
expression --> expression, [-], term.
expression --> term.

term --> term, [*], factor.
term --> term, [/], factor.
term --> factor.

factor --> integer.
factor --> string.

factor --> identifier.

factor --> ['('], expression, [')'].

% <if_statement> ::= "if" "(" <condition> ")" "{" <block> "}" |
<if_statement1>
% <if_statement1> ::= ""
% <if_statement1> ::= "else" "{" <block> "}"
% <if_statement1> ::= "else" <if_statement>

if_statement --> [if], ['('], condition, [')'], ['{'], block, ['}'].
if_statement --> if_statement1.
if_statement1 --> [].
if_statement1 --> [else], ['{'], block, ['}'].
if_statement1 --> [else], if_statement.


% <condition> ::= <expression> <relation_op> <expression>
% <condition> ::= <expression> <logical_op> <expression>
```

```
condition --> expression, relation_op, expression.
condition --> condition, logical_op, condition.
condition --> boolean.
condition --> [!], condition.


% <relation_op> ::= "<" | "<=" | ">" | ">=" | "==" | "!="
% <logical_op> ::= "&&" | "||"

relation_op --> [<].
relation_op --> [<=].
relation_op --> [>].
relation_op --> [>=].
relation_op --> [==].
relation_op --> ['!='].

logical_op --> ['&&'].
logical_op --> ['||'].


% <while_loop> ::= "while" "(" <condition> ")" "{" <block> "}"
% <for_loop> ::= "for" "(" <identifier> "=" <for_integer> ";" <condition> ";"
<expression> ")" "{" <block> "}"
% <for_range> ::= "for" <identifier> "in" "range" "(" <for_integer> ","
<for_integer> ")" "{" <block> "}"
% <for_integer> ::= <integer> | <identifier>

while_loop --> [while], ['('], condition, [')'], ['{'], block, ['}'].
for_loop --> [for], ['('], identifier, [=], for_integer, [;], condition, [;],
assignment, [')'], ['{'], block, ['}'].
for_range --> [for], identifier, [in], [range], ['('], for_integer, [';'],
for_integer, [')'], ['{'], block, ['}'].
for_integer --> integer.
for_integer --> identifier.


% <output> ::= "print" "(" <expression1> ")"
% <expression1> ::= <expression> "," <expression1> | <expression

print_statement --> [print], ['('], print_values, [')'].
print_values --> string, [','], print_values.
print_values --> identifier, [','], print_values.
print_values --> integer, [','], print_values.
print_values --> integer.
print_values --> string.
print_values --> identifier.
```

Tokenization

- Converting the stream in program into tokens.
- This is implemented in Python
- The output will be given to the parser.

Input:

```
program {
  int a = 10;

  print("Value of a: ", a);
}
```

Output:

```
program
{
int
a
=
10
;
print
(
"
Value of a:
"
,
a
)
;
}
```

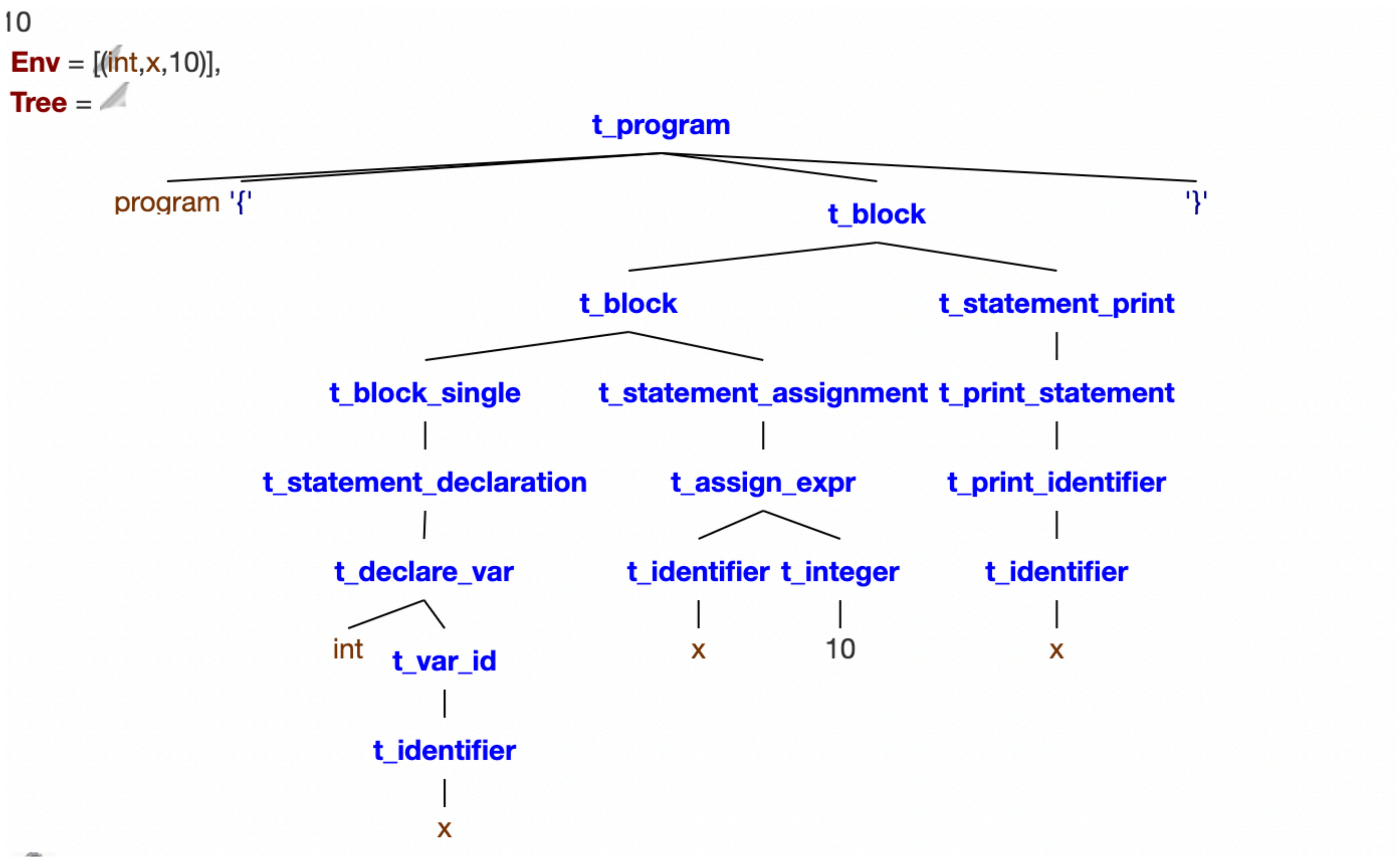
Parsing in Prolog

- Checks the grammar of the language from the input tokens.
- The tokens are converted into the parse tree.
- Parser is written in Prolog and DCG is used to check the grammar.

Input:

```
program
{
int
a
=
10
;
print
(
"
Value of a:
"
,
a
)
;
}
```

Output:



```
10
Env = [(int,x,10)],
Tree =
```

Evaluating expressions in Prolog

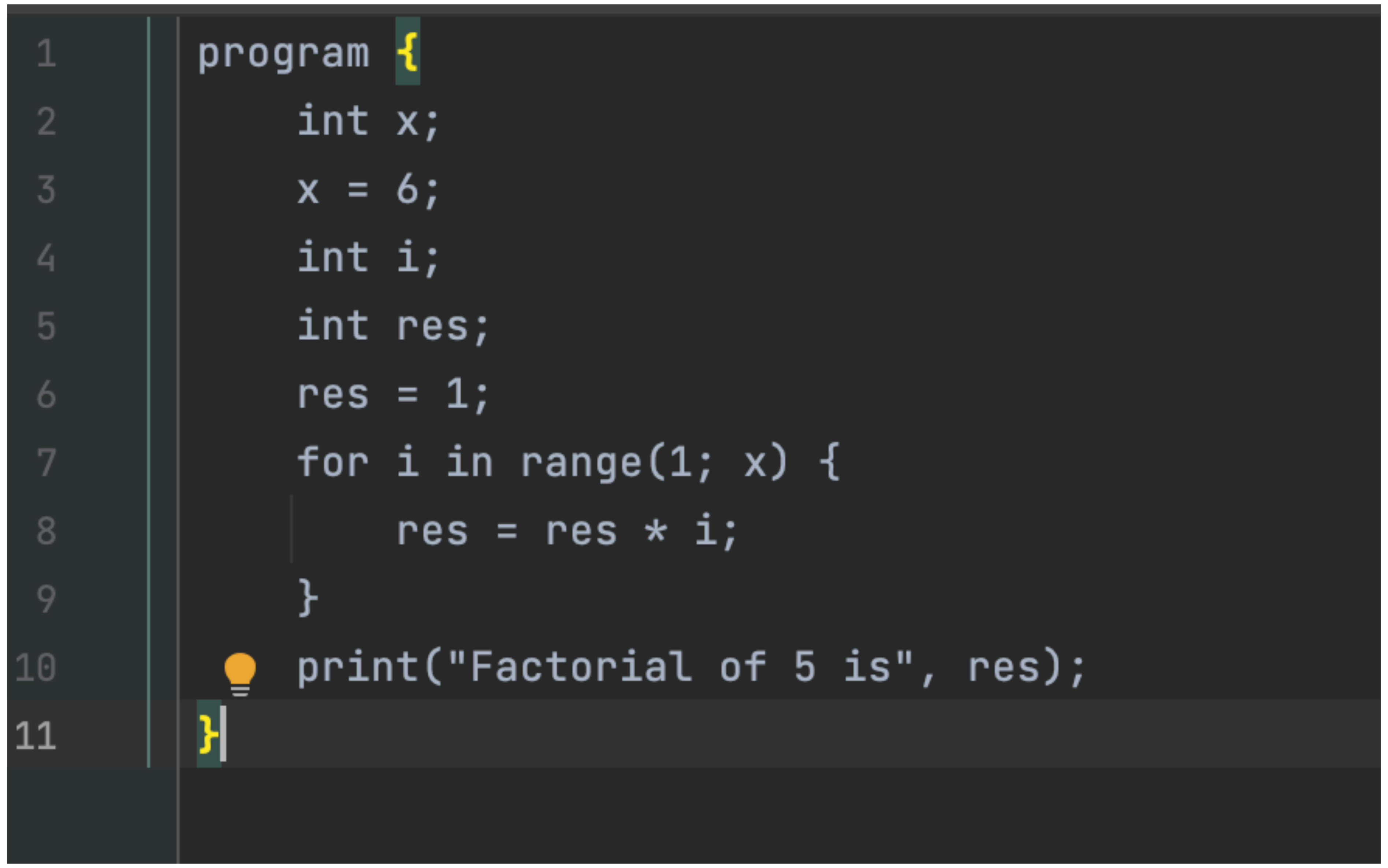
- Evaluators work using pattern matching.
- Take input as expressions and produce resulting values and set of values.
- This is implemented in DCG and prolog.

- Key concepts:

1. Lookup table
2. Update table

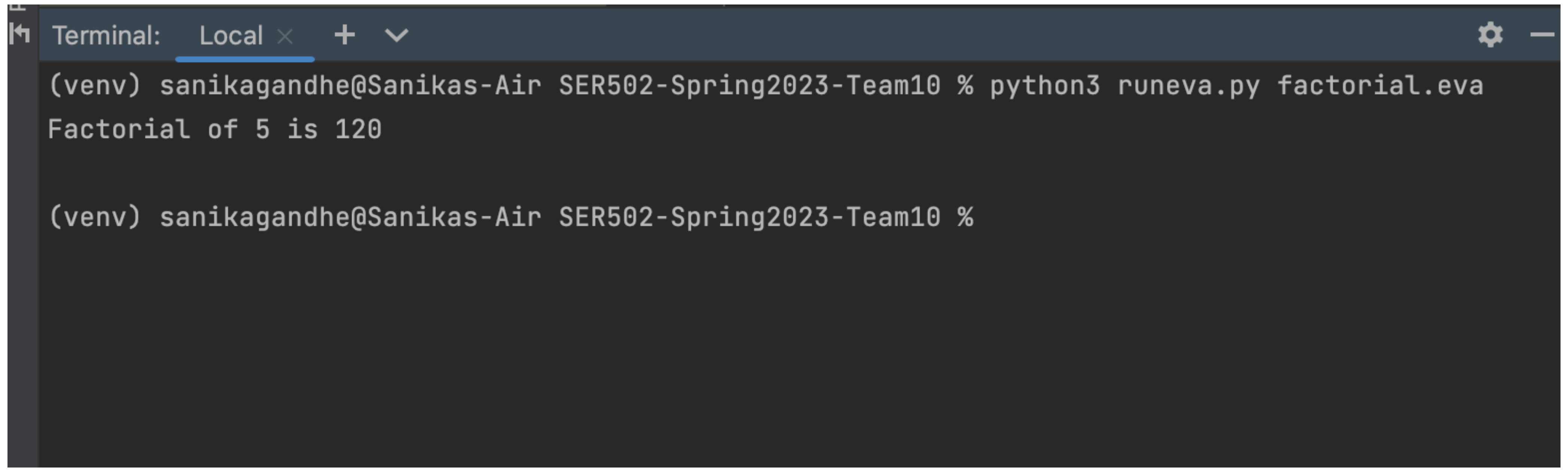
Execution of the program

- Eva program



```
program {
    int x;
    x = 6;
    int i;
    int res;
    res = 1;
    for i in range(1; x) {
        res = res * i;
    }
    print("Factorial of 5 is", res);
}
```

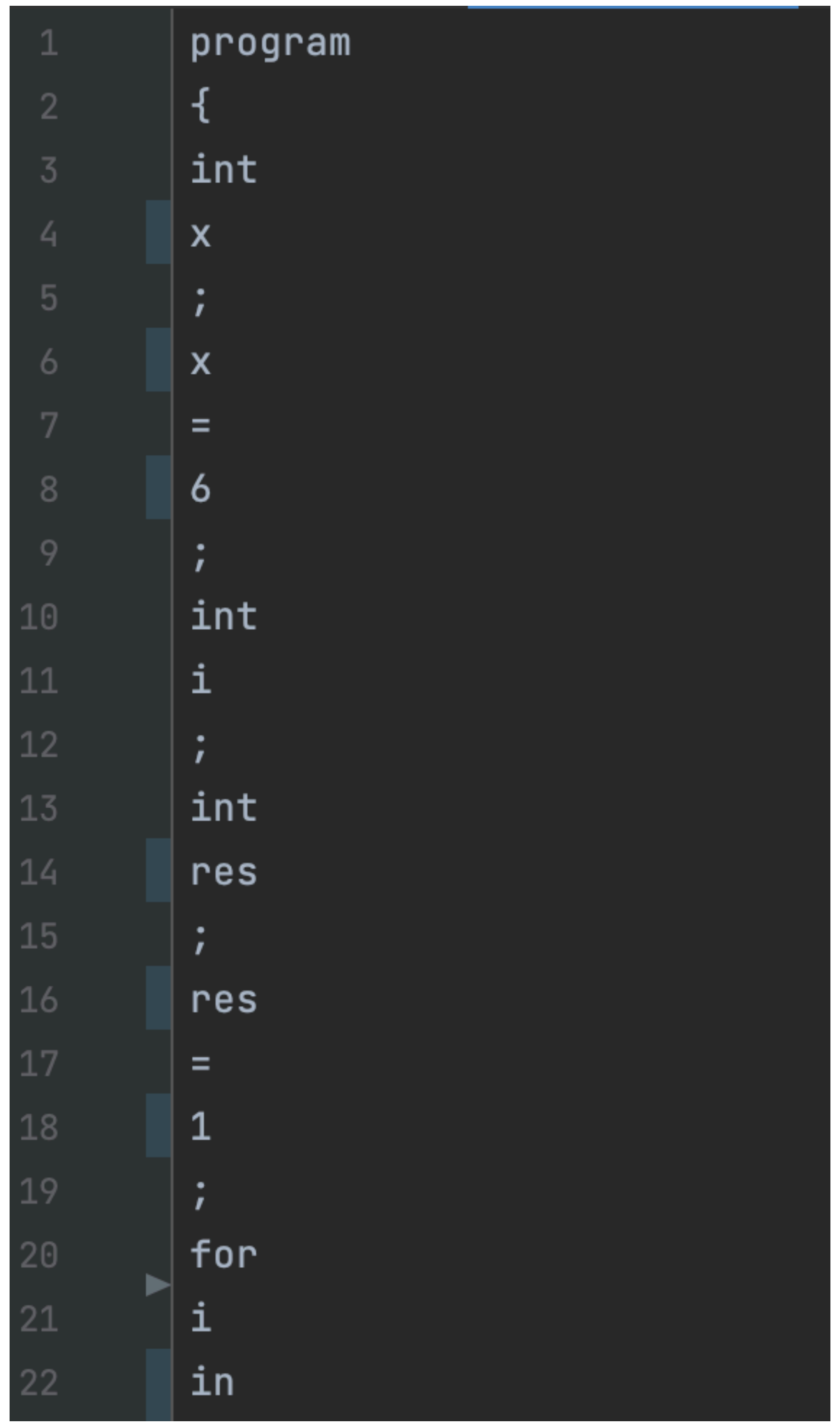
- Terminal commands



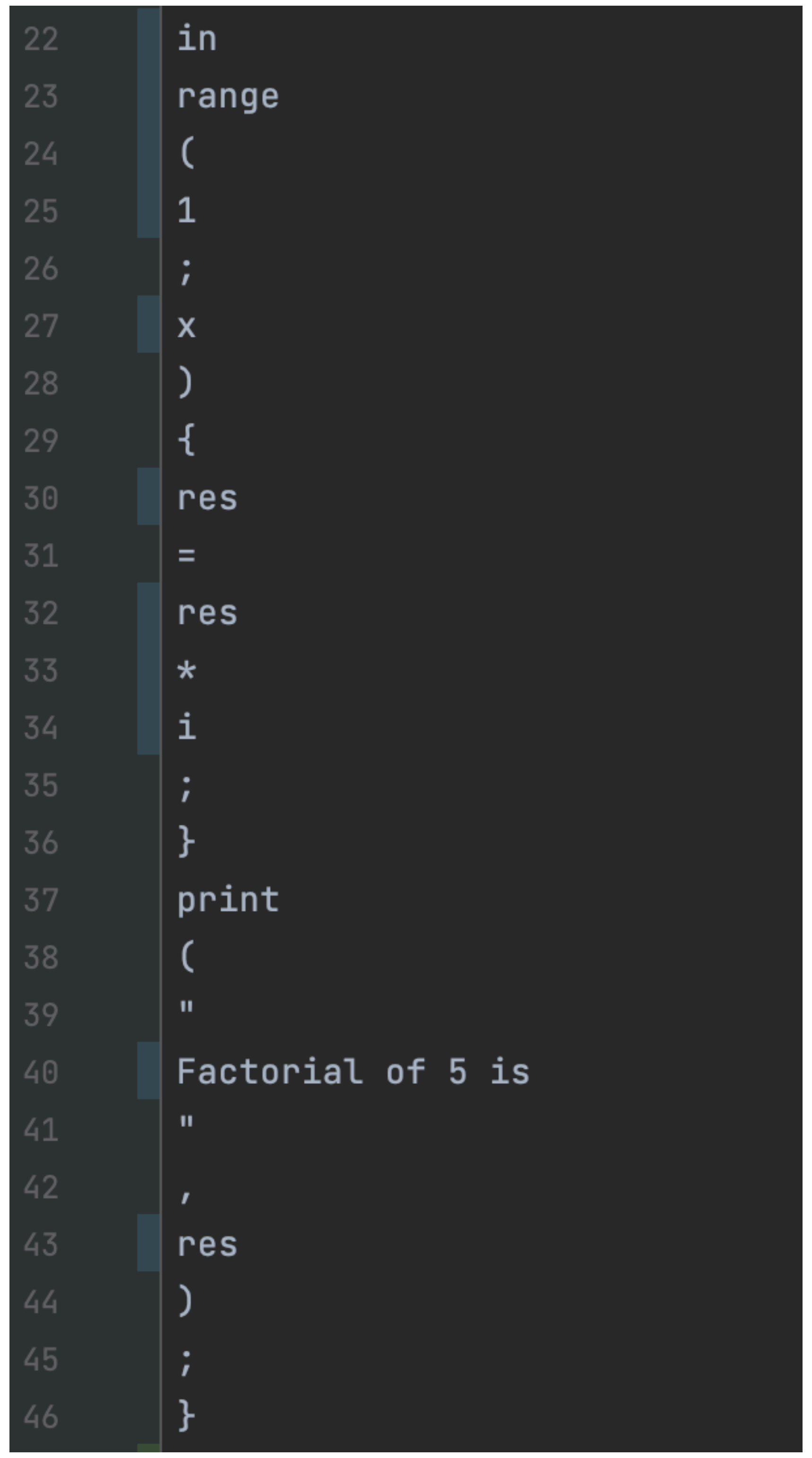
```
(venv) sanikagandhe@Sanikas-Air SER502-Spring2023-Team10 % python3 runeva.py factorial.eva
Factorial of 5 is 120

(venv) sanikagandhe@Sanikas-Air SER502-Spring2023-Team10 %
```

Execution of the program

- Intermediate files (tokens files)

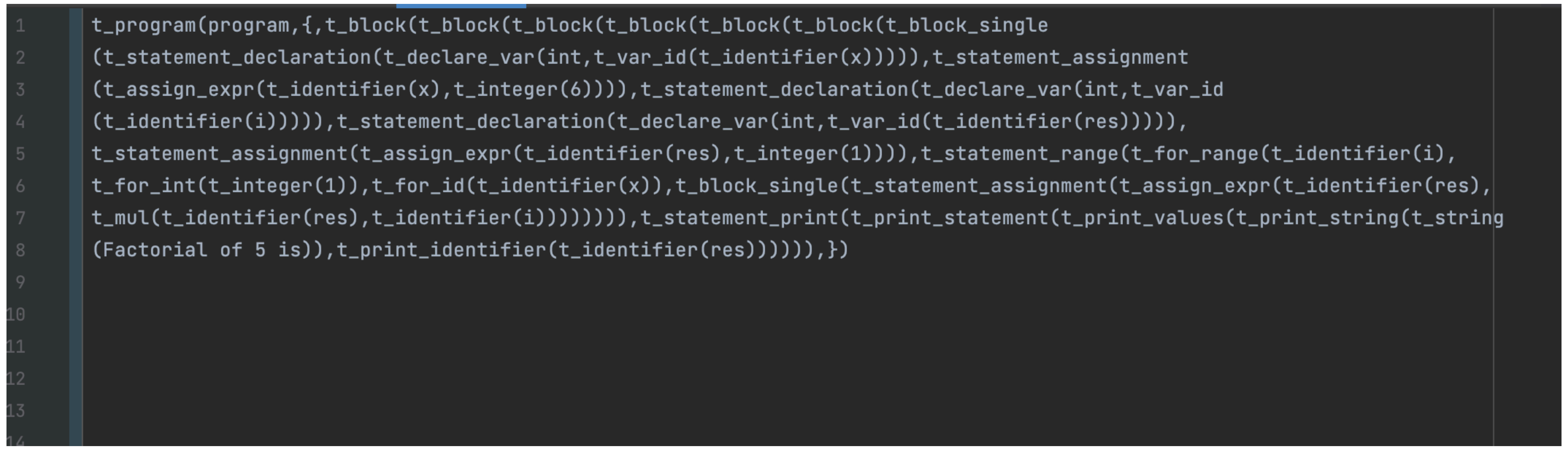


```
program
{
int
x
;
x
=
6
;
int
i
;
int
res
;
res
=
1
;
for
i
in
```



```
in
range
(
1
;
x
)
{
res
=
res
*
i
;
}
print
(
"
Factorial of 5 is
"
,
res
)
;
}
```

- Intermediate files (parse tree files)



```
t_program(program,{,t_block(t_block(t_block(t_block(t_block(t_block(t_block_single
(t_statement_declaration(t_declare_var(int,t_var_id(t_identifier(x))))),t_statement_assignment
(t_assign_expr(t_identifier(x),t_integer(6)))),t_statement_declaration(t_declare_var(int,t_var_id
(t_identifier(i))))),t_statement_declaration(t_declare_var(int,t_var_id(t_identifier(res))))),
t_statement_assignment(t_assign_expr(t_identifier(res),t_integer(1)))),t_statement_range(t_for_range(t_identifier(i),
t_for_int(t_integer(1)),t_for_id(t_identifier(x)),t_block_single(t_statement_assignment(t_assign_expr(t_identifier(res),
t_mul(t_identifier(res),t_identifier(i)))))))),t_statement_print(t_print_statement(t_print_values(t_print_string(t_string
(Factorial of 5 is)),t_print_identifier(t_identifier(res)))))),})
```

Future Scope

- Reading data from the user
- Introducing data structures
- Functions
- More datatypes (char, float)