FUNLANG

SER502 Spring 2024
Project Team 29

Presented by:-

DARSH MANIAR

SULTAN ALNEIF

JINESH PATEL

SARATH KUMAR DUNGA

WILLIAM HUANG

CONTENTS

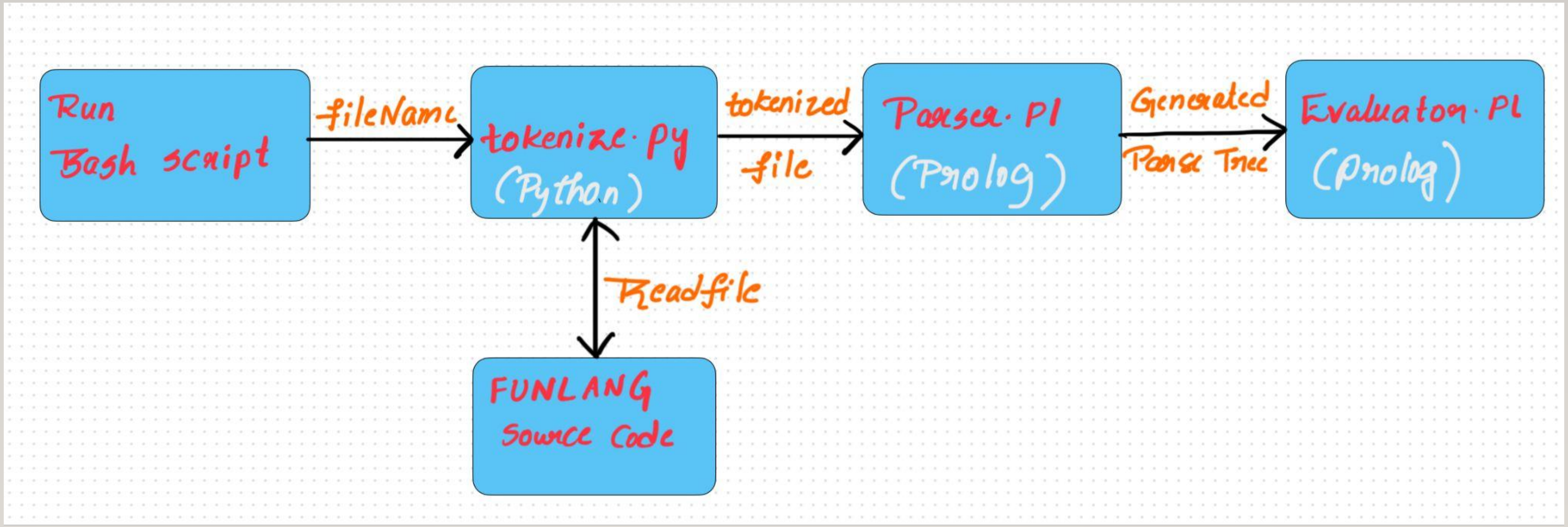
INTRODUCING : FUNLANG

Dive into FUNLANG – a programming language designed to make writing code simpler, intuitive, and, most importantly, fun with every line of code.

Inspired from the class assignments by Dr. Ajay Bansal, we've developed a language that offers intuitive syntax and basic programming features.

While currently in its initial phase, FUNLANG supports all the basic features of the language. Moreover, it's designed to evolve over time, with the capability to add more constructs to support additional features as needed.

Basic Flow of FUNLANG



DESIGN COMPONENTS

- **One line bash script** to run FUNLANG
  ./run_program.sh <file-path>/**<file-name>**
- **File extention:** **.fl**
- **Data folder :** has the sample programs to test out language.
- **Src folder :** This folder has the grammar, parser, evaluator for our language.
- **Tokenizer:** Tokenizer will accept the sample programs from data folder as the input. The input program is broken down into list of tokens with the help of tokenize.py file. The tokenizer was defined with the rules maintaining consistency with the parser.
- **Parser :** This Prolog file defines rules for parsing the parse tree using the rules defined in Grammar.py file. It defines predicates to parse different language constructs recursively, such as declarations, commands, expressions, etc. The file also handles conditionals, loops, function definitions, and other control flow constructs.
- **Evaluator :** This Prolog file contains predicates for evaluating the parse tree generated by the parser. It defines rules for evaluating declarations, commands, expressions, conditions, loops, functions, etc. It interacts with the environment to perform variable assignments, updates, and lookups during evaluation.

LANGUAGE FEATURES:

Commands :

- boolean_assignment
- assignment
- if_statement
- if_else_statement
- ternary_statement
- while_loop
- print_statement
- for_range_statement
- for_list_statement
- func_declaration

Declaration :

- const_declaration
- var_declaration
- bool_declaration
- var_assignment
- bool_assignment
- list_declaration
- func_declaration
- dict_declaration

LANGUAGE FEATURES:

Keywords Reserved:

- const, var, bool, func, true, false, not, and, or, if, else, while, print, for, in, range

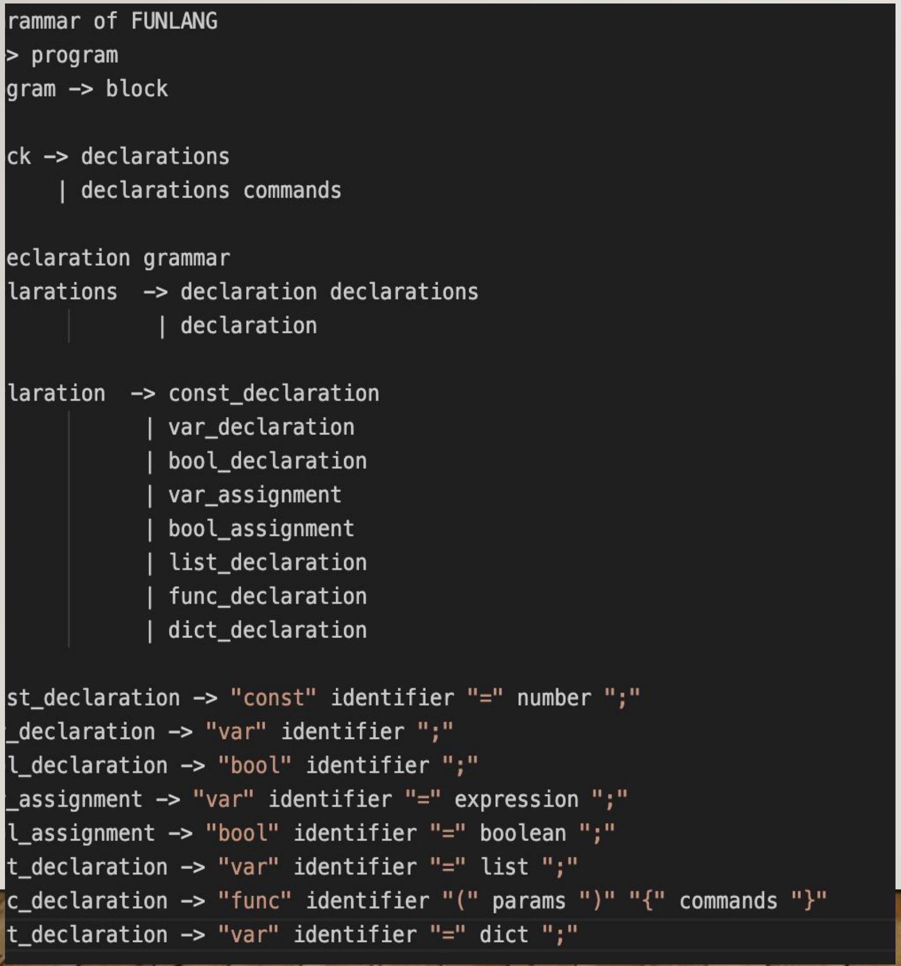
Data types :

- Integer
- Character
- String
- Boolean

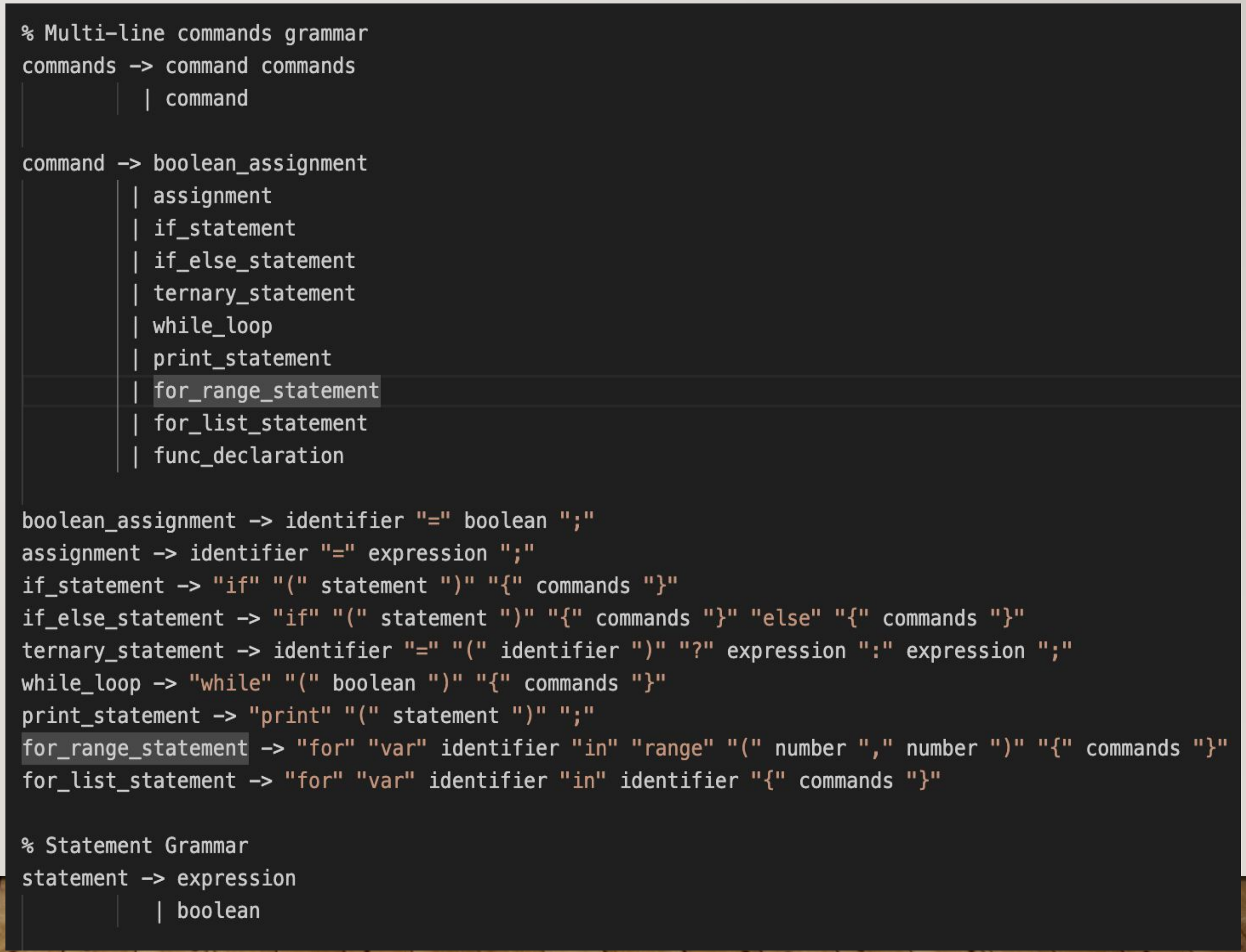
Operations :

- Addition
- Subtraction
- Multiplication
- Division
- Braces

GRAMMAR

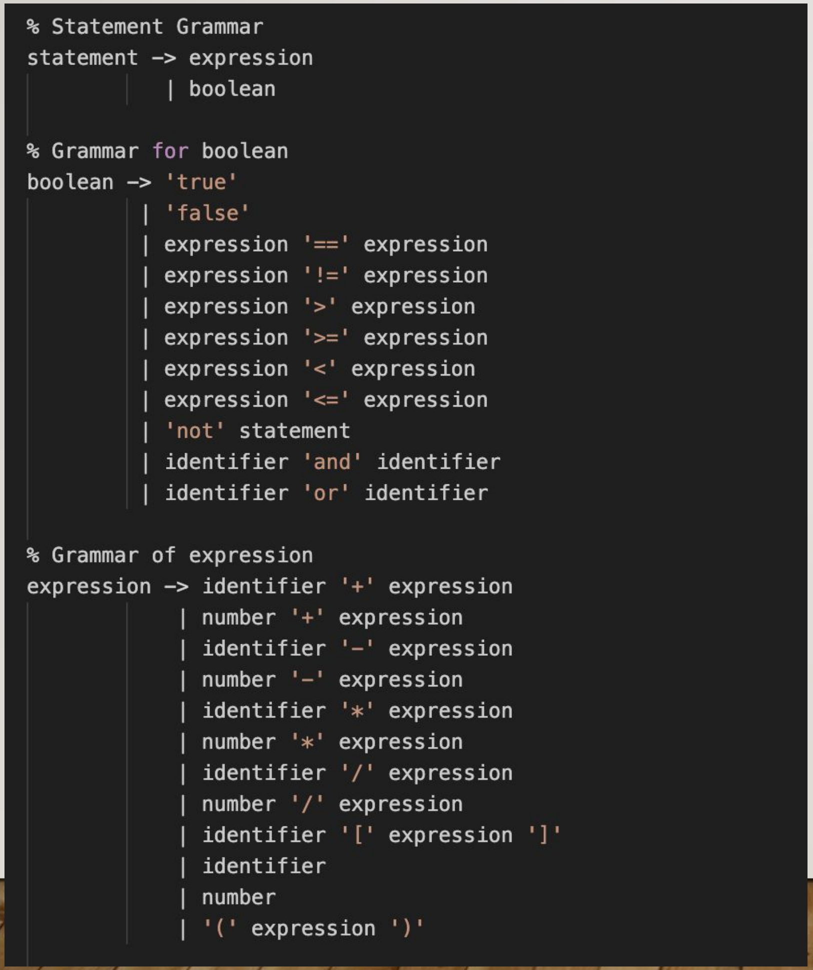


```
rammar of FUNLANG
> program
gram -> block

ck -> declarations
    | declarations commands

eclaration grammar
larations  -> declaration declarations
            | declaration

laration  -> const_declaration
           | var_declaration
           | bool_declaration
           | var_assignment
           | bool_assignment
           | list_declaration
           | func_declaration
           | dict_declaration

st_declaration -> "const" identifier "=" number ";"
_declaration -> "var" identifier ";"
l_declaration -> "bool" identifier ";"
_assignment -> "var" identifier "=" expression ";"
l_assignment -> "bool" identifier "=" boolean ";"
t_declaration -> "var" identifier "=" list ";"
c_declaration -> "func" identifier "(" params ")" "{" commands "}"
t_declaration -> "var" identifier "=" dict ";"
```

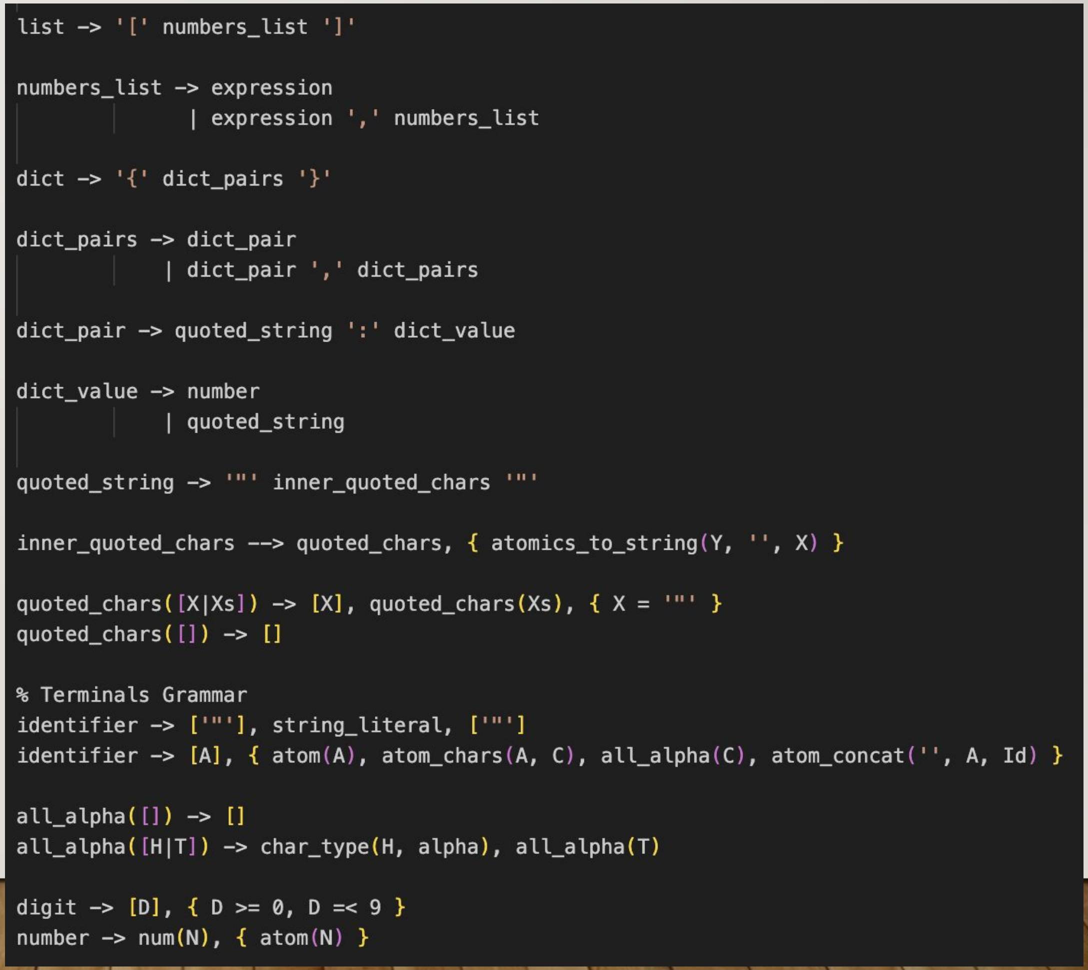


```
% Multi-line commands grammar
commands -> command commands
          | command

command -> boolean_assignment
         | assignment
         | if_statement
         | if_else_statement
         | ternary_statement
         | while_loop
         | print_statement
         | for_range_statement
         | for_list_statement
         | func_declaration

boolean_assignment -> identifier "=" boolean ";"
assignment -> identifier "=" expression ";"
if_statement -> "if" "(" statement ")" "{" commands "}"
if_else_statement -> "if" "(" statement ")" "{" commands "}" "else" "{" commands "}"
ternary_statement -> identifier "=" "(" identifier ")" "?" expression ":" expression ";"
while_loop -> "while" "(" boolean ")" "{" commands "}"
print_statement -> "print" "(" statement ")" ";"
for_range_statement -> "for" "var" identifier "in" "range" "(" number "," number ")" "{" commands "}"
for_list_statement -> "for" "var" identifier "in" identifier "{" commands "}"

% Statement Grammar
statement -> expression
           | boolean
```

GRAMMAR



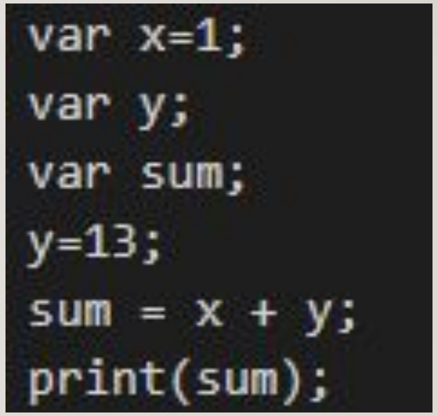
```
% Statement Grammar
statement -> expression
            | boolean

% Grammar for boolean
boolean -> 'true'
          | 'false'
          | expression '==' expression
          | expression '!=' expression
          | expression '>' expression
          | expression '>=' expression
          | expression '<' expression
          | expression '<=' expression
          | 'not' statement
          | identifier 'and' identifier
          | identifier 'or' identifier

% Grammar of expression
expression -> identifier '+' expression
             | number '+' expression
             | identifier '-' expression
             | number '-' expression
             | identifier '*' expression
             | number '*' expression
             | identifier '/' expression
             | number '/' expression
             | identifier '[' expression ']'
             | identifier
             | number
             | '(' expression ')'
```



```
list -> '[' numbers_list ']'

numbers_list -> expression
              | expression ',' numbers_list

dict -> '{' dict_pairs '}'

dict_pairs -> dict_pair
            | dict_pair ',' dict_pairs

dict_pair -> quoted_string ':' dict_value

dict_value -> number
            | quoted_string

quoted_string -> '"' inner_quoted_chars '"'

inner_quoted_chars --> quoted_chars, { atomics_to_string(Y, '', X) }

quoted_chars([X|Xs]) -> [X], quoted_chars(Xs), { X = '"' }
quoted_chars([]) -> []

% Terminals Grammar
identifier -> ['"'], string_literal, ['"']
identifier -> [A], { atom(A), atom_chars(A, C), all_alpha(C), atom_concat('', A, Id) }

all_alpha([]) -> []
all_alpha([H|T]) -> char_type(H, alpha), all_alpha(T)

digit -> [D], { D >= 0, D =< 9 }
number -> num(N), { atom(N) }
```

GENERATE PARSE TREE

- Input: tokenized Funlang source code (tokenized by tokenize.py)
- Sample Input: ['var', 'x', '=', '1', ';', 'var', 'y', ';', 'var', 'sum', ';', 'y', '=', '13', ';', 'sum', '=', 'x', '+', 'y', ';', 'print', '(', 'sum', ')', ';']

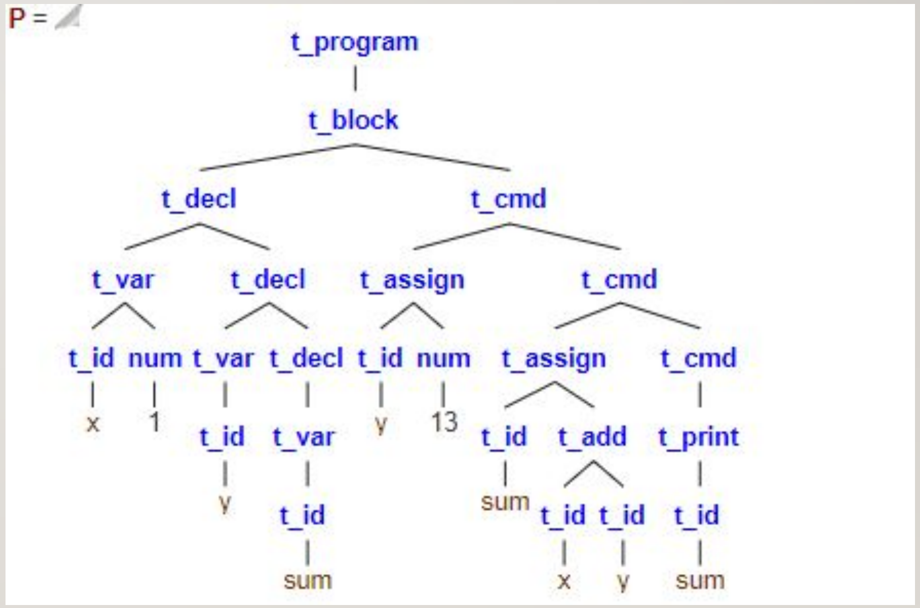


```
var x=1;
var y;
var sum;
y=13;
sum = x + y;
print(sum);
```

Program_1.fl Source Code

GENERATE PARSE TREE

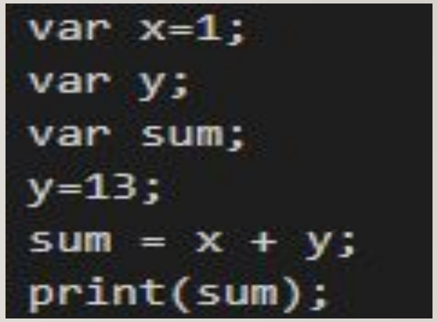
- Output: parsed tree of Funlang source code
- Sample output: t_program(t_block(t_decl(t_var(t_id(x),num(1)),t_decl(t_var(t_id(y)),t_decl(t_var(t_id(sum))))),t_cmd(t_assign(t_id(y),num(13)),t_cmd(t_assign(t_id(sum),t_add(t_id(x),t_id(y))),t_cmd(t_print(t_id(sum)))))))



Output Parsed Tree

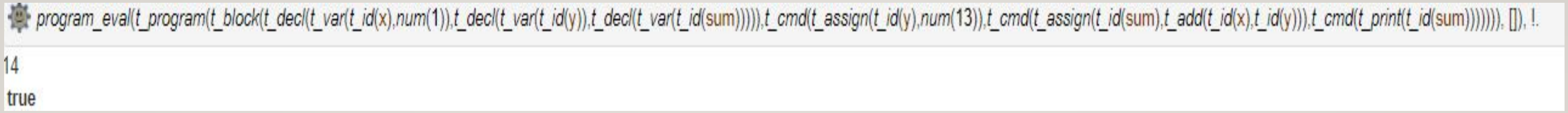
EVALUATOR

- Input: parsed tree of Funlang source code
- Output: evaluation of the input parsed tree
- Inputting the sample parsed tree from the earlier slide evaluates: 14



```
var x=1;
var y;
var sum;
y=13;
sum = x + y;
print(sum);
```

Program_1.fl Source Code

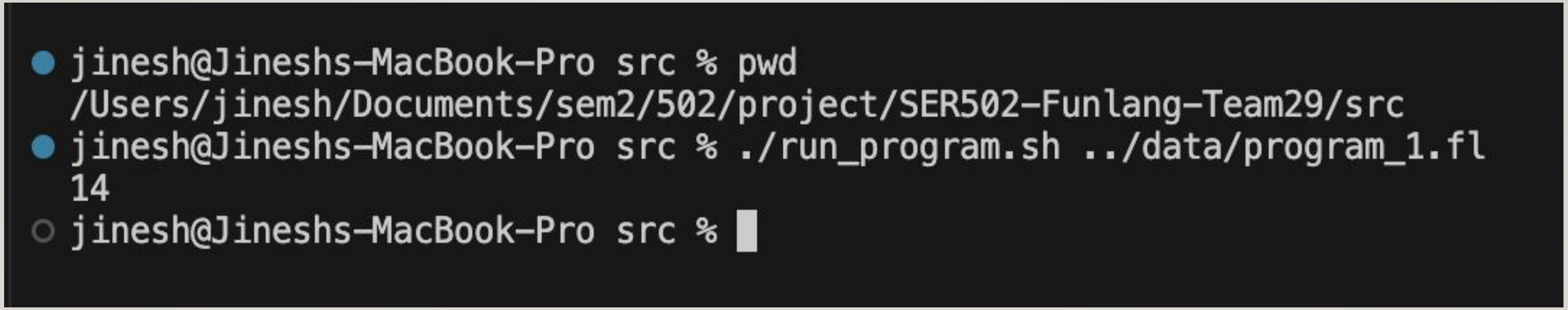


```
program_eval(t_program(t_block(t_decl(t_var(t_id(x),num(1)),t_decl(t_var(t_id(y)),t_decl(t_var(t_id(sum))))),t_cmd(t_assign(t_id(y),num(13)),t_cmd(t_assign(t_id(sum),t_add(t_id(x),t_id(y))),t_cmd(t_print(t_id(sum))))))), []), !.
14
true
```

Output Evaluation Results

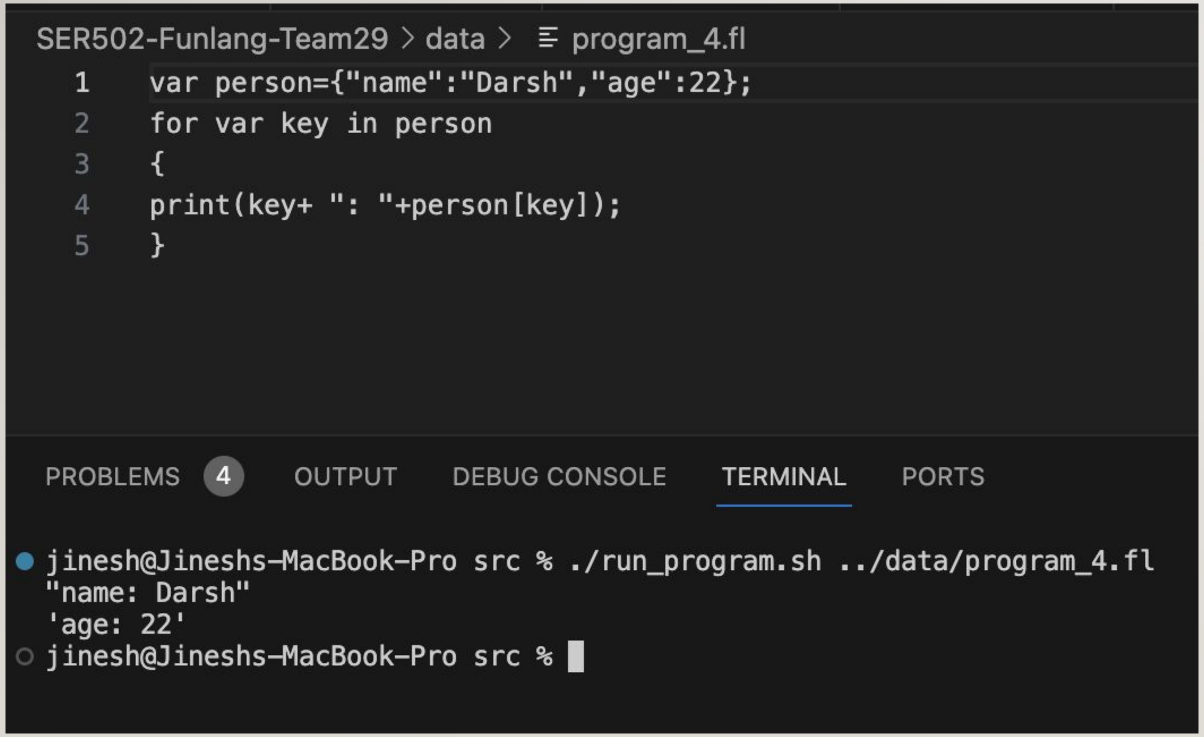
RUNTIME

- Go to **src** folder
- Run: **./run_program.sh <file-path>/<file-name>**



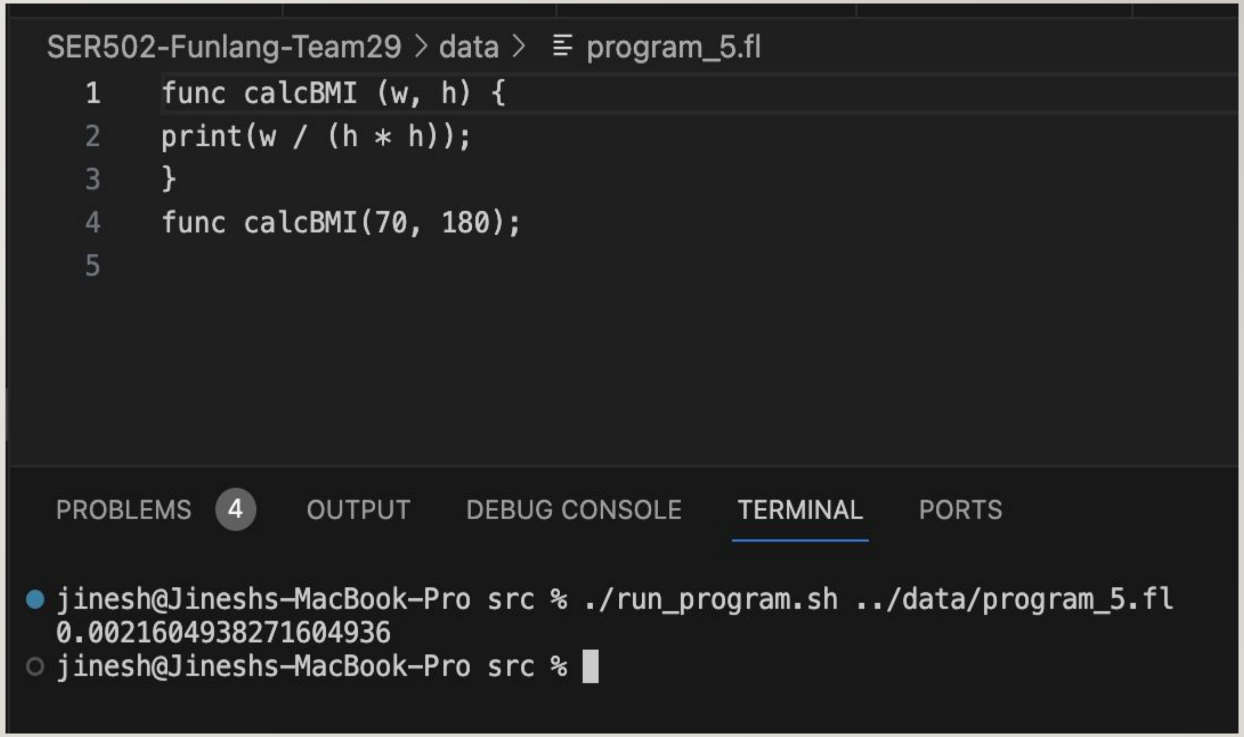
```
● jinesh@Jineshs-MacBook-Pro src % pwd
  /Users/jinesh/Documents/sem2/502/project/SER502-Funlang-Team29/src
● jinesh@Jineshs-MacBook-Pro src % ./run_program.sh ../data/program_1.fl
  14
○ jinesh@Jineshs-MacBook-Pro src % █
```

DEMONSTRATION SNAPSHOTS



```
var person={"name":"Darsh","age":22};
for var key in person
{
print(key+ ": "+person[key]);
}
```

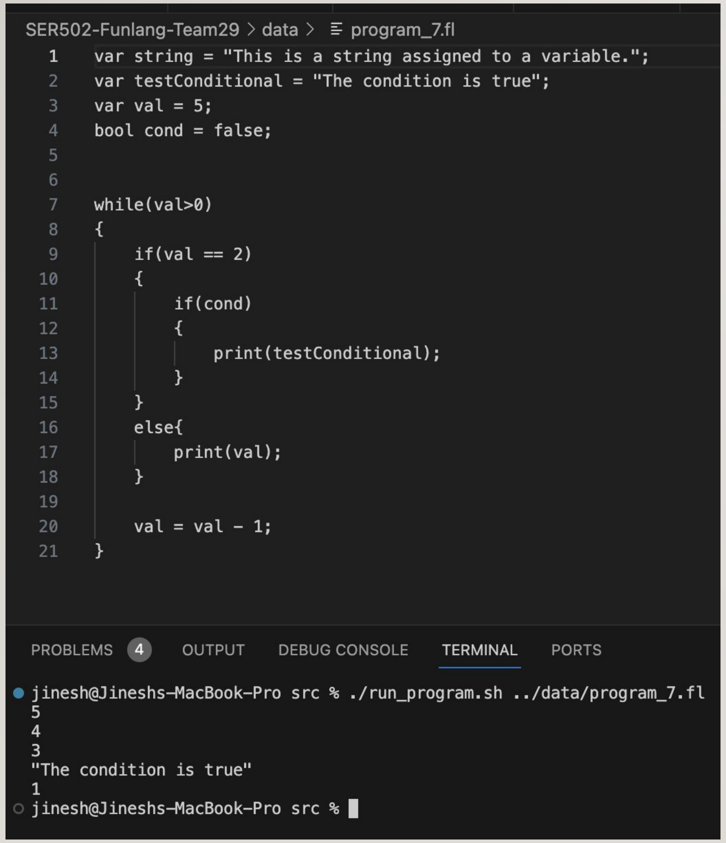
```
jinesh@Jineshs-MacBook-Pro src % ./run_program.sh ../data/program_4.fl
"name: Darsh"
'age: 22'
jinesh@Jineshs-MacBook-Pro src %
```



```
func calcBMI (w, h) {
print(w / (h * h));
}
func calcBMI(70, 180);
```

```
jinesh@Jineshs-MacBook-Pro src % ./run_program.sh ../data/program_5.fl
0.0021604938271604936
jinesh@Jineshs-MacBook-Pro src %
```

DEMONSTRATION SNAPSHOTS

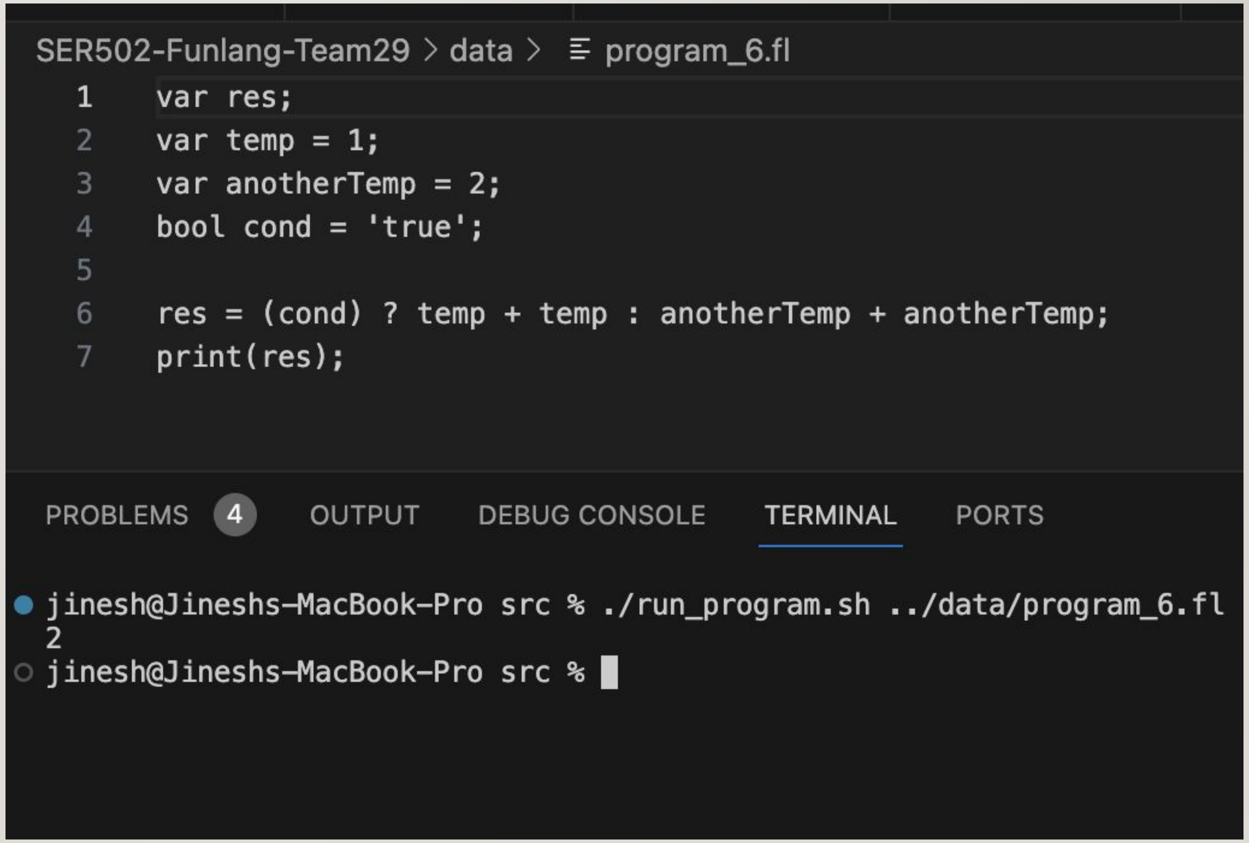


SER502-Funlang-Team29 > data > program_7.fl

```
var string = "This is a string assigned to a variable.";
var testConditional = "The condition is true";
var val = 5;
bool cond = false;


while(val>0)
{
    if(val == 2)
    {
        if(cond)
        {
            print(testConditional);
        }
    }
    else{
        print(val);
    }

    val = val - 1;
}
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
jinesh@Jineshs-MacBook-Pro src % ./run_program.sh ../data/program_7.fl
5
4
3
"The condition is true"
1
jinesh@Jineshs-MacBook-Pro src %
```



SER502-Funlang-Team29 > data > program_6.fl

```
var res;
var temp = 1;
var anotherTemp = 2;
bool cond = 'true';

res = (cond) ? temp + temp : anotherTemp + anotherTemp;
print(res);
```

PROBLEMS 4 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
jinesh@Jineshs-MacBook-Pro src % ./run_program.sh ../data/program_6.fl
2
jinesh@Jineshs-MacBook-Pro src %
```

Conclusion

- Easy to Write:

  FUNLANG is a simple and intuitive language that supports common data types, conditional statements, loop structures, and custom declaration.
- Easy to Run:

  With the fully developed lexer, parser, evaluator, and shell, it is easy to generate the parse tree and program output by using one line bash script.
- Easy to Evolve:

  FUNLANG has a flexible grammar structure so it is always possible to add more constructs and new feature supports.