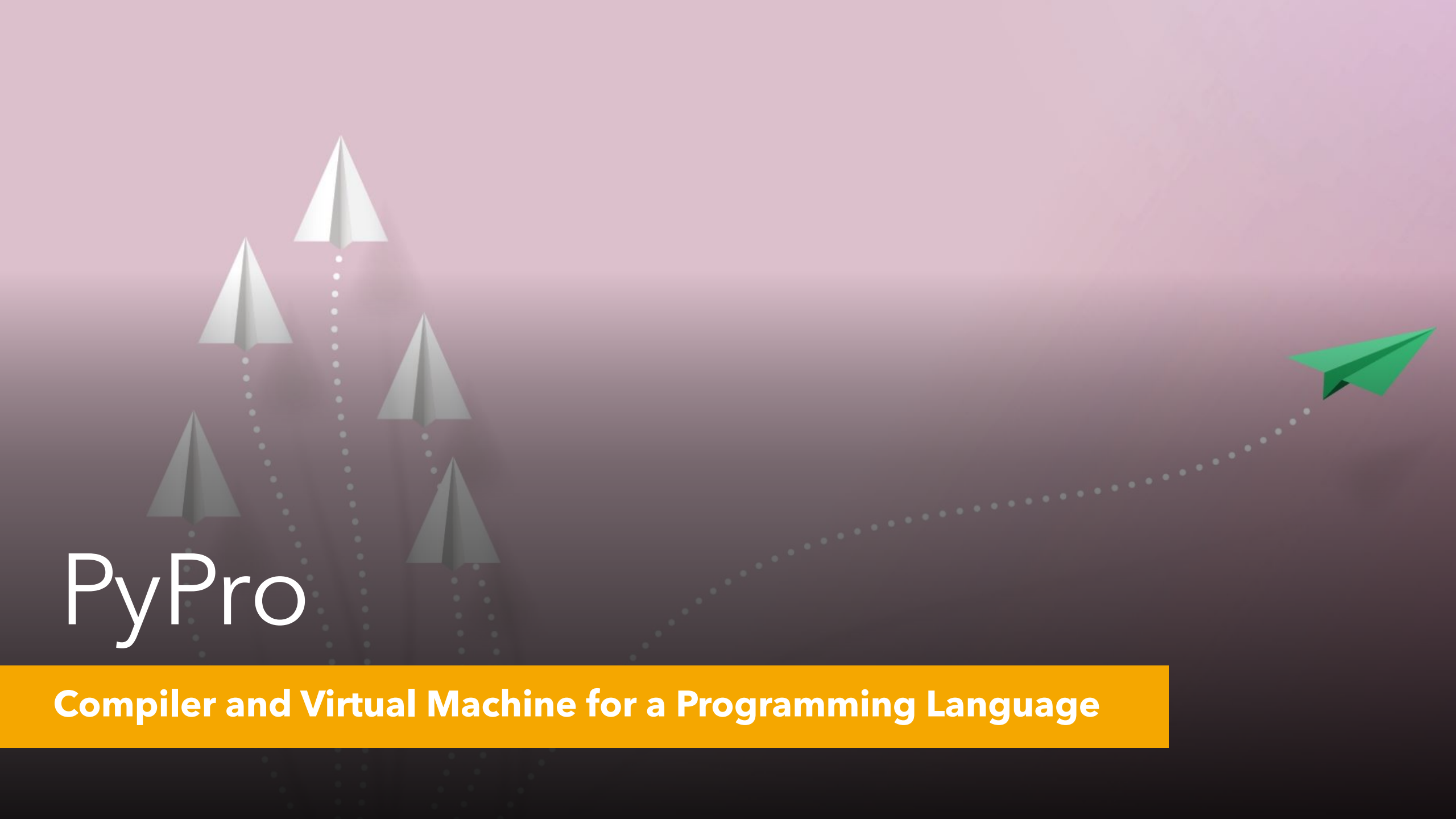

PyPro
Compiler and Virtual Machine for a Programming Language

Language Design

This language is named “PyPro” because we are using python and prolog to build this language. Python is used in lexical analysis phase, Prolog is used in Parsing or to generate Intermediate code and semantics phase.

Pypro Design:

- Program contains Commands
- Commands can be a single command or group of commands.

Commands

- An operation followed by semicolon (;), This operation can be simple assignment. User need not specify the data type of a variable while declaration. The data type of a variable and supported operations on that variable will be handled in semantics phase. Additionally, this operation will support
    - Boolean assignment
    - Multiple assignments
    - Increment operation (++)
    - Decrement operation (--)
    - Increment with assignment (+=)
    - Decrement with assignment (-=)
    - Multiplication with assignment (*=)
    - Division with assignment (/=)
    - Modulus with assignment (%=)
    - Integer Division with assignment (//=)
    - an Expression, an expression can be
        - expression addition (expr + expr)
        - expression subtraction (expr - expr)
        - expression multiplication (expr * expr)
        - expression division (expr / expr)
        - expression integer division (expr // expr)
        - expression modulus (expr % expr)
        - expression power (expr ^ expr)
        - number, string, combination of number and letters
        - parenthesis ( )

Commands

- While loop
  - Conditional Statement supports, refer next slide
  - body of while loop contains commands
- ternary operator
  - Conditional Statement of ternary operator supports, refer next slide
  - body of ternary operator can only contain expressions
- if else conditional statement
  - Conditional Statement supports refer point 8
  - body contains commands
- if else conditional statement and command to support nested if else
  - Conditional Statement of " if ", " elif " supports: refer next slide
  - body contains commands
- for i in range (2,5) { }
  - Conditional Statement of for loop supports refer next slide
  - for loop considers both min and max vales for computation. In above example it take 2 and 5 along with 3,4 for computation as well.
  - body of for loop contains commands
- traditional for loop, for (i=0; i<5; i++) { }
  - Conditional Statement supports refer next slide
  - body of for loop contains commands

Conditional statement in if, ternary operator, while and for loops

- Conditional statements of for loops, while loops, if else, nested if else, include Boolean statement to check if condition satisfies.
    - Boolean includes not, and, or, comparisons like
        - expression equals expression (expr == expr)
        - number
        - String
        - number and Boolean (ex: 1 and true)
        - expr! = Boolean
        - Boolean! = expr
        - expression not equals (expr! = expr)
        - true == true
        - false == false
        - less than or equal to (expr <= expr)
        - greater than or equal to (expr >= expr)
        - less than (expr < expr)
        - greater than (expr > expr)

Commands

- Print statement followed by semicolon (;)
  - Print can display combination of numbers, strings, Booleans and variables
- Function declaration, contains
  - keyword “function”
  - function name
  - Parameter list refer point next slide
  - Body of function is group of commands
- Function declaration with return type, contains
  - keyword “function”
  - keyword “return”
  - function name
  - Parameter list refer point next slide
  - Body of function is group of commands
  - return statement, can only have one return statement, which is an expression
- Function declaration with return type but no body, contains
  - keyword “function”
  - keyword “return”
  - function name
  - Parameter list refer next slide
  - return statement, can only have one return statement, which is an expression

Commands

- Parameter list of a function declaration can be
  - a parameter, which is an identifier or a group of parameters

- Function call with parameters, contains
  - function name
  - parameter list can be a parameter which is a number or an identifier; or group of parameters.
  - arguments cannot contain expressions
  - followed by semicolon

Language Design (cont)

- **The language supports operators and primitive types for:**

a) Boolean values (support and, or and not operators)

b) int, float (supports addition, subtraction, multiplication and division, integer division, modulus, power)

c) numbers can be negative as well, supporting unary minus

d) supports string value assignments to variables and operations as concatenation, multiplication of a string with a number

e) String can be “String” or ‘Sting’ or empty as well “ ” .

f) String can also be combinations of letters and integers

- **Variable naming should follow**
- Lowercase letter followed by integer or uppercase letter
- Lowercase letter followed by various combinations of a integer and uppercase letters

Note: Some restrictions in language design

- If function body contains loops and if-else statements then loops and if-else cannot have return statement inside function. Return statement can only be at the end of the function.

- Strings can't have " inside single quoted string
- Ex: s = ‘Hello “PyPro’ and s = 'hello "world';

- Ternary operator must be used in an assignment statement

- if, then, for, while must to have at least one command.

Grammar

- Program → Commands
- Commands → Command
- Commands → Command Commands
- Command → Assign ;
- Command → while BooleanBool { Commands }
- Command → while (BooleanBool){ Commands }
- Command →Word = BooleanBool ? Expr : Expr ;
- Command → Word = (BooleanBool) ? Expr : Expr ;
- Command → if BooleanBool { Commands }
- Command → if BooleanBool {Commands } Command_el
- Command → if (BooleanBool) { Commands }
- Command → if (BooleanBool) {Commands } Command_el

Grammar (cont)

- Command → for Word in range (Expr, Expr) { Commands }
- Command → for (Assign ; BooleanBool ; Assign) { Commands }
- Command → for (Assign ; (BooleanBool) ; Assign) { Commands }
- Command → function Word (ParameterList) { Commands }
- Command → function return Word (ParameterList) { Commands return Expr ;}
- Command → function return Word (ParameterList) { return Expr; }
- Command → Word (ParameterList_call) ;
- Command → print (Printseq);
- Command_el → elif BooleanBool { Commands }
- Command_el → elif BooleanBool { Commands } Command_el
- Command_el → else { Commands }
- Command_el → elif (BooleanBool) { Commands }
- Command_el → elif (BooleanBool) { Commands } Command_el
- Printseq → (Expr) + Printseq
- Printseq → String + Printseq
- Printseq → String | Expr

Grammar (cont)

- ParameterList → Word , ParameterList
- ParameterList → Word
- ParameterList_call → Parameter_call , ParameterList_call
- ParameterList_call → Parameter_call
- Parameter_call → Number | Word
- Boolean → true | false | not Boolean | Expr == Expr | Expr == Boolean | Boolean == Expr | Expr != Boolean | Boolean != Expr | true == true | false == false | true == false | false == true | Expr != Expr | Boolean and Boolean | Boolean or Boolean | Expr < Expr | Expr > Expr | Expr <= Expr | Expr >= Expr
- BooleanTerm →Number | Word | String
- BooleanTerm → Boolean
- BooleanBool →not BooleanTerm

Grammar (cont)

- BooleanBool →Boolean and BooleanTerm
- BooleanBool →BooleanTerm and Boolean
- BooleanBool → Boolean or BooleanTerm
- BooleanBool →BooleanTerm or Boolean
- BooleanBool →BooleanTerm and BooleanTerm
- BooleanBool →BooleanTerm or BooleanTerm
- Assign →Word = Boolean
- Assign →Word = Assign
- Assign →Word + + | Word - -
- Assign →Word += Assign | Word -= Assign | Word *= Assign | Word /= Assign | Word %= Assign | Word //= Assign | Expr
- Expr →Expr + Expr | Expr – Expr | Expr * Expr | Expr / Expr | Expr // Expr | Expr % Expr | Expr ^ Expr | (Assign)

Grammar (cont)

- Expr → evaluate Word (ParameterList_call)
- Expr → Number | String | Word
- Number → N, Number | N
- N → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- Word → Small Word | Small
- Word → Small Capital Word | Small Capital
- Word → Small Number Word | Small Capital Number Word | Small Number | Small Capital Number
- Small → 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
- Capital → 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 → "Seq" | 'Seq'
- Seq → Word ' ' | Word | ' '

Lexical analyzer

- We used python library called “Sly” to generate tokens. These tokens are passed as a list to Prolog DCG (Definite Clause Grammar) file.
- In this phase we are eliminating single and multiline comments, white spaces and newlines
- Any characters other than characters specified in the grammar will be treated as illegal, lexical analyzer will throw a syntax error with line number and aborts the program.
- Tokens are generated for only necessary code.
- For the below factorial program, the token list generated will be

For the below factorial program, the token list generated will be

```
       n = 5;

# this function computes factorial of an integer
function return factorial(i){
    if(i > 0){
        d = i - 1;
        y = i * evaluate factorial(d);
    }
    else{
        y = 1;
    }
    return y;
}
z = evaluate factorial(n);
print("factorial of " + (n) + " is " + (z));
```

[n,=,5,;,function,return,factorial,'(',i,')','{',if,'(',i,>,0,')','{',d,=,i,-
,1,;,y,=,i,*,evaluate,factorial,'(',d,')',;,'}',else,'{',y,=,1,;,'}',return,y,;,'}',z,=,evaluate,factorial,'(',n,')',;,pr
int,'(',"factorial of ",+,'(',n,')',+," is ",+,'(',z,')',')',;]

Parsing or Intermediate Code Generation phase:

- The generated tokens “list” from Lexical analyzer is consumed by our Prolog DCG file. This phase of will generate Abstract Syntax Tree (AST) if the syntax of the program is correct as per the rules defined by PyPro language.
- The generated AST will be passed to our PyPro evaluator file for assigning semantics.
- If the program doesn’t follow the syntax rules, AST will not be generated, and program execution will be terminated with an error message.
- For the above program, AST will be

For the above program, AST will be

P = *t_command*(*t_command_assign*(*t_aAssign*(*t_word*(n), *t_num*(5))), *t_command*(*t_method_decl_ret*(*t_word*(factorial), *t_word*(i), *t_command*(*t_command_ifel*(*t_b_g*(*t_word*(i), *t_num*(0)), *t_command*(*t_command_assign*(*t_aAssign*(*t_word*(d), *t_sub*(*t_word*(i), *t_num*(1)))), *t_command*(*t_command_assign*(*t_aAssign*(*t_word*(y), *t_mult*(*t_word*(i), *t_method_call_ret*(*t_word*(factorial), *t_parameter_call*(*t_word*(d))))))), *t_command_else*(*t_command*(*t_command_assign*(*t_aAssign*(*t_word*(y), *t_num*(1)))))), *t_word*(y)), *t_command*(*t_command_assign*(*t_aAssign*(*t_word*(z), *t_method_call_ret*(*t_word*(factorial), *t_parameter_call*(*t_word*(n))))), *t_command*(*t_print*(*t_expr_print_sp*("factorial of
", *t_expr_print_ep*(*t_word*(n), *t_expr_print_sp*(" is
", *t_expr_print_e*(*t_paren*(*t_word*(z))))))))))

Assigning Semantics to Abstract Syntax Tree:

- The generated AST will be used to assign semantics by our Prolog evaluator file which also handles run time environment as a “list”.
- We only have global scope throughout the program. Local scopes are not supported. Wherever a variable value is changed it gets updated in global environment list.
- Our language doesn’t need to explicitly specify data type before variable declarations. We are using predicate “number” to identify if it’s an integer or float and “string” predicate to identify strings.
- The type of operations on integer and float are same. We support multiplication, division, subtraction, modulus, integer division, increment, decrement, power. Before these operations are performed on a variable’s value, its type is checked.
- The same way with strings, before performing operation we check its type, our language supports string concatenations, multiplication of a string with number (like in python, 4 * “world” gives output worldworldworldworld), string and number concatenations.

Snapshots of sample executions : 1

```
}ravi@archlinux:~/SER502-Spring2020-Team15/data$ pypro sample_4.pr
i and j are true
2 SER502
string x is empty, so (not x) is true
string123
true and 1 is true
not of (false and 0) is true
```

```
i = 2;
j = "SER502";
if ( i and j){
    print("i and j are true");
    print((i) +" "+ (j));
}

x = "";
if ( not x ){
    print("string x is empty, so (not x) is true");
}

if ("string123"){
    print("string123");
}

a = true;
b = 1;
if (a and b){
    print((a) + " and " + (b) + " is true");
}

a = false;
b = 0;

if (not a and b){
    print("not of (false and 0) is true");
```

Snapshots of sample executions : 2

```
ravi@archlinux:~/SER502-Spring2020-Team15/data$ cat sample_add.pr
x = true;
print("x - " + (x));

y = 2;
print("y - " + (y));

z = "string";
print("z - " + (z));

y++;
print("y++ - " + (y));

add = y + 2;
print("add = y + 2 -->" + (add));

add_2 = add + y;
print("add_2 = add+ y --> " + (add_2));

add_2 += y;
print("add_2 += y --> " + (add_2));

add_3 = z + " second string";
print("z + second string --> " + (add_3));

add_3 += " third string";
print("add_3 += third string --> " + (add_3));
```

```
ravi@archlinux:~/SER502-Spring2020-Team15/data$ pypro sample_add.pr
x - true
y - 2
z - string
y++ - 3
add = y + 2 -->5
add_2 = add+ y --> 8
add_2 += y --> 11
z + second string --> string second string
add_3 += third string --> string second string third string
```

Snapshots of sample executions : 3

```
ravi@archlinux:~/SER502-Spring2020-Team15/data$ cat sample_mod.pr
y = 2;
print("y --> " + (y));

mod = y % 2;
print("mod = y % 2 -->" + (mod));

mod_2 = mod % y;
print("mod_2 = mod % y --> " + (mod_2));

mod_2 %= y;
print("mod_2 %= y --> " + (mod_2));
```

```
ravi@archlinux:~/SER502-Spring2020-Team15/data$ pypro sample_mod.pr
y --> 2
mod = y % 2 -->0
mod_2 = mod % y --> 0
mod_2 %= y --> 0
```

Snapshots of sample executions : 4

```
ravi@archlinux:~/SER502-Spring2020-Team15/data$ cat sample_2_factorial.pr
n = 5;

# this function computes factorial of an integer
function return factorial(i){
    if(i > 0){
        d = i - 1;
        y = i * evaluate factorial(d);
    }
    else{
        y = 1;
    }
    return y;
}
z = evaluate factorial(n);
print("factorial of " + (n) + " is " + (z));
```

```
ravi@archlinux:~/SER502-Spring2020-Team15/data$ pypro sample_2_factorial.pr
factorial of 5 is 120
```

Snapshots of sample executions : 5

```
ravi@archlinux:~/SER502-Spring2020-Team15/data$ cat sample_1.pr
x = 0;
y = "string1";
z = true;
ter = 3>2?(5*3):2/1;
# if else and if
print("if else and if");
if(z == true){
     print("inside if");
    if(x <= 0){
        print("inside nested if");
    }
}
else{
    print("inside else");
}

#if elif else
print("if elif else");
if(y == "string2"){
    print("inside if");
}
elif(y == "string1"){
    print("inside elif");
}
else{
    print("inside else");
}

# for traditional
print("Traditional for loop.");
for(i = 0; i < 10; i++){
    print("i - " + (i));
}

# for variable in range
print("range for loop.");
for i in range(0,10){
    print("i - " + (i));
}
```

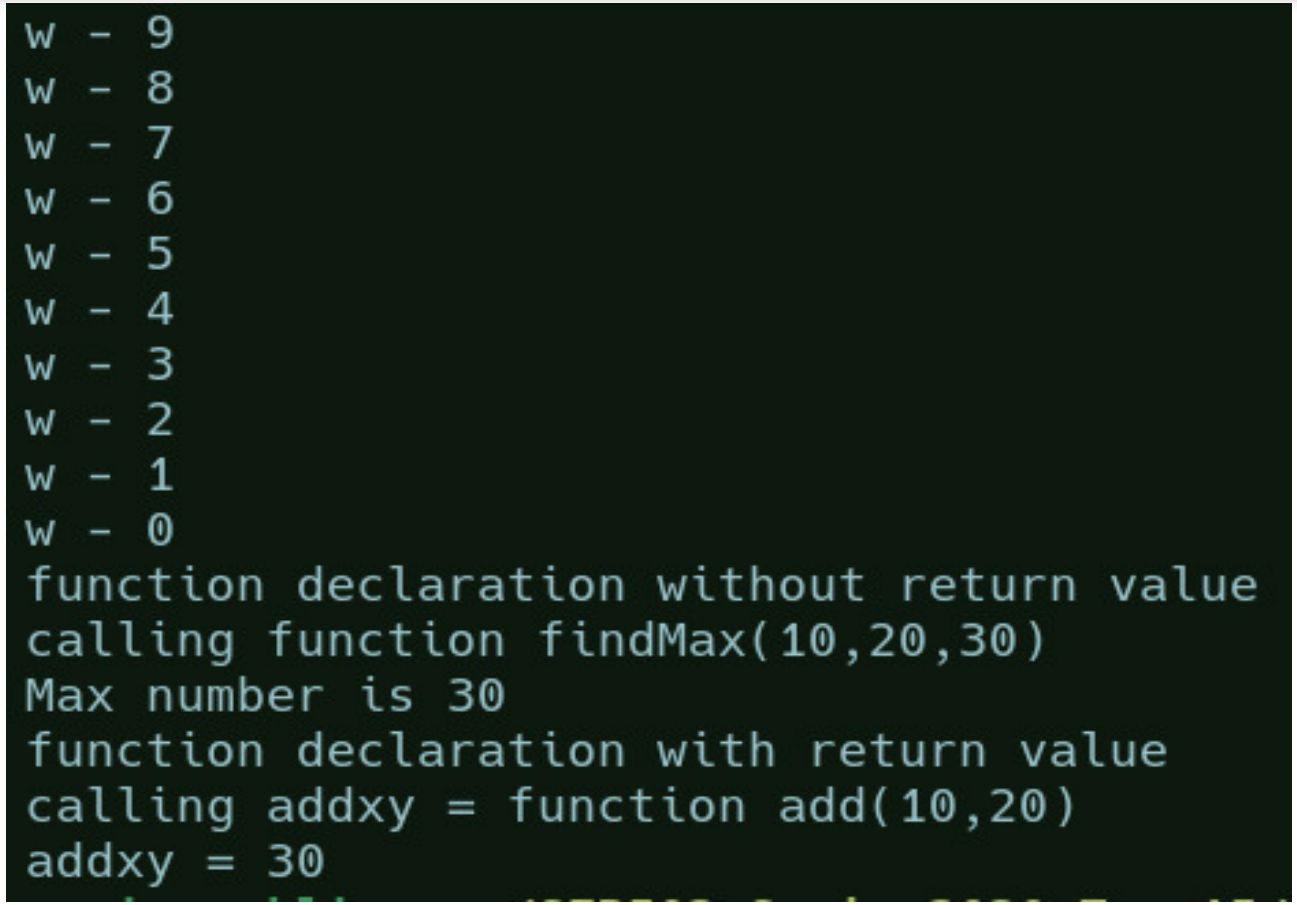
Snapshots of sample executions (continue) : 5

```
# while loop
print("while loop.");
w = 20;
while(w >= 0){
    print("w - " + (w));
    w--;
}

# function declaration without return value
print("function declaration without return value");
function findMax(x1,y1,z1){
    if(x1 > y1){
        if(x1 > z1){
            print("Max number is " + (x1));
        }
        else{
             print("Max number is " + (z1));
        }
    }
    elif(y1 > z1){
         print("Max number is " + (y1));
    }
    else{
        print("Max number is " + (z1));
    }
}

print("calling function findMax(10,20,30)");
findMax(10,20,30);

# function declaration with return value
print("function declaration with return value");
function return add(x,y){
    return x + y;
}

print("calling addxy = function add(10,20)");
addxy = evaluate add(10,20);
print("addxy = " + (addxy));
```

Snapshots of sample executions (continue) : 5

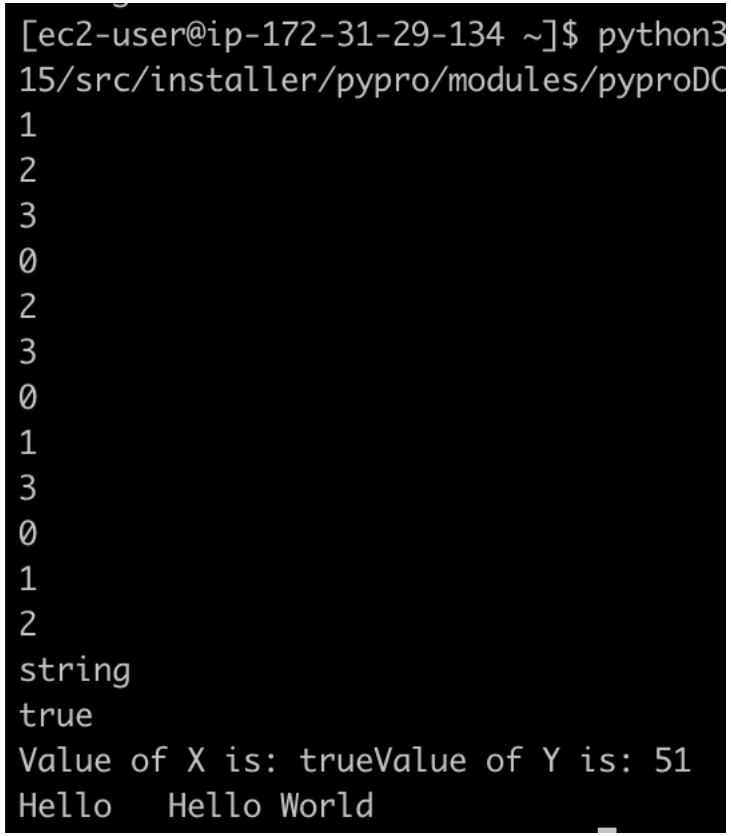
```
ravi@archlinux:~/SER502-Spring2020-Team15/data$ pypro sample_1.pr
if else and if
inside if
inside nested if
if elif else
inside elif
Traditional for loop.
i - 0
i - 1
i - 2
i - 3
i - 4
i - 5
i - 6
i - 7
i - 8
i - 9
range for loop.
i - 10
while loop.
w - 20
w - 19
w - 18
w - 17
w - 16
w - 15
w - 14
w - 13
w - 12
w - 11
w - 10
w - 9
w - 8
w - 7
w - 6
w - 5
w - 4
w - 3
```

Snapshots of sample executions (continue) : 5



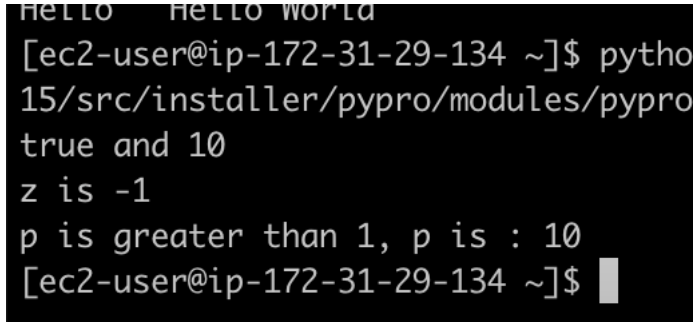
Snapshots of sample executions : 6 (this program is executed on amazon ec2 instance, with Red hat Linux.

```
x = true;
# this is a comment
y = 3;
z = 1;
for(i=0;i<4;i++)
{
for(j=0; j<4; j++)
{
  if(i != j)
  {
  print(j);
  }
  y= y+3;
}
z = z*2;
}

s = "string";
print(s);
print(x);
print("Value of X is: "+(x)+ "Value of Y is: " +(y));
print("Hello "+"  "+ "Hello World");
```



Snapshots of sample executions : 7 (this program is executed on amazon ec2 instance, with Red hat Linux.

```
x = true;
w = false;
y = 3;
z = -1;
p = 10;
if(x and p)
{
print((x)+" "+"and "+(p));
}
else
{
print("outside of if");
}
if(z)
{
 print("z is "+(z));
}
else
{
print("z is not 0");
}

if(p)
{
 print("p is greater than 1, p is : "+(p));
}
else
{
print("z not greater than 0");
}
```



Installation

Note: Currently, you can only install in linux based operating systems. But, you can still run the language in other operating systems as shown in running pypro section. If you are using macOS, you may face issues while installing dependecies, but if you managed to install them correctly then you should be able to install and run the language.

Before installing make sure you have following dependencies installed correctly in your system:

- swi-prolog - version 8.0.3 or higher (make sure it is in your PATH)
- python - 3.6 or higher
- sly - `pip3 install sly`
- Pyswip

Clicking on the links above will take you to the respective installation pages. Note: you may need to use `pip` instead of `pip3` if you are using windows.

After you have installed all the above dependencies follow these instructions:

1.Download the project zip or clone the repo.

2.Open the terminal in the root folder of the project.

3.Execute the following commands:

```
$ cd src/installer/$ sudo ./install.sh
```

To remove/uninstall PyPro, from project root folder:

```
$ cd src/installer/$ sudo ./uninstall.sh
```

Running PyPro

You can run pypro in windows, macOs and linux operating systems without installing or building. First, make sure you have installed all the dependencies mentioned in installation section.

After downloading zip or cloning the repository, From project root navigate to src/ and from command prompt:

```
$ python3 pypro.py path/filename
```

Here, filename is the name of the file with the appropriate code for pypro. It should have `.pr` extension.

Example:

```
$ python3 pypro.py ../data/test.pr
```

Note: you may need to use `python` instead of `python3` if you are using windows.