Project: Compiler and Virtual Machine for a Programming Language

SER 502 - Team 15
Suyog Halikar
Somesh Siddabasappa
Narmada Ravali Namburi
Ravikanth Reddy Dodda

Name: PyPro

Language Design:

- Program contains Commands
- Commands can be a single command or group of commands.
- Each command can be
    1. 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 ( )

2. While loop
    - Conditional Statement supports, refer point 8
    - body of while loop contains commands
3. ternary operator
    - Conditional Statement of ternary operator supports, refer point 8
    - body of ternary operator can only contain expressions

4. if else conditional statement
    - Conditional Statement supports refer point 8
    - body contains commands
5. if else conditional statement and command to support nested if else
    - Conditional Statement of " if ", " elif " supports: refer point 8
    - body contains commands
6. for i in range (2,5) { }
    - Conditional Statement of for loop supports refer point 8
    - 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
7. traditional for loop, for (i=0; i<5; i++) { }
    - Conditional Statement supports refer point 8
    - body of for loop contains commands
8. 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)

9. Print statement followed by semicolon (;)
    - Print can display combination of numbers, strings, Booleans and variables

10. Function declaration, contains
    - keyword “function”
    - function name
    - Parameter list refer point 13
    - Body of function is group of commands

11. Function declaration with return type, contains
    - keyword “function”
    - keyword “return”
    - function name
    - Parameter list refer point 13
    - Body of function is group of commands
    - return statement, can only have one return statement, which is an expression

12. Function declaration with return type but no body, contains
    - keyword “function”
    - keyword “return”
    - function name
    - Parameter list refer point 13
    - return statement, can only have one return statement, which is an expression

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

14. 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

- The language supports operators and primitive types for:
    1. Boolean values (support and, or and not operators)
    2. int, float (supports addition, subtraction, multiplication and division, integer division, modulus, power)
    3. numbers can be negative as well, supporting unary minus
    4. supports string value assignments to variables and operations as concatenation, multiplication of a string with a number
    5. String can be “String” or ‘Sting’ or empty as well “ ” .
    6. String can also be combinations of letters and integers

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

- **Note: Some restrictions in language design**
    1. 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.

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

    3. Ternary operator must be used in an assignment statement

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

Grammar: Our language PyPro 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

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

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

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)

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

```
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,fa
ctorial,'(',n,')',;,print,'(',"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

  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.

- During runtime, certain type of operations are not allowed, for example:

    - redefining a function, will throw an error "*function name is already defined*"

    - parameters mismatch in function calls, will throw an error "parameters does not match for function call"

    - comparison of a number and Boolean, will throw "Boolean and Number cannot be compared"

    - comparison of a string and Boolean, will throw "Boolean and string cannot be compared"

    - increment of a string or Boolean, will throw "String and Boolean cannot be incremented"

    - decrement of a string or Boolean, will throw "String and Boolean cannot be decremented"

    - multiplication of strings, will throw an error "Type Error: multiplication of string and string is not allowed"

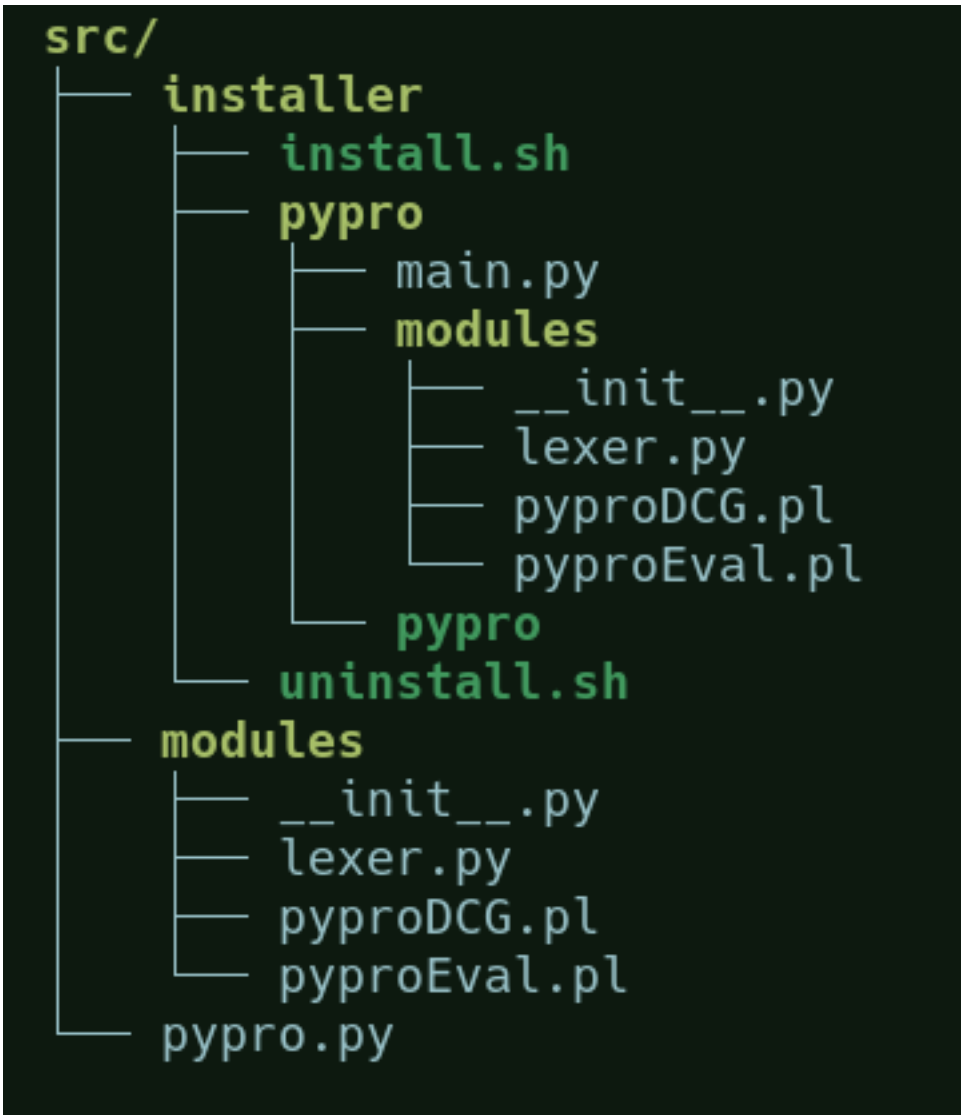
    - negating a string, will throw an error "unary minus cannot be used with string"

    - Power operation of string, Boolean, are rejected

- And many other illegal comparisons are rejected with appropriate error messages.

Folder structure:

We used the following folder structure for src folder:



src folder has an installer folder which contains all required files for installing the language. Additionally it has 1 more folder “modules” which contains files for lexer,parser and runtime evaluation. One python file “pypro.py” is also in the src folder which bridges all the files in modules and can be used to run the language.

Built using

- MacOS
- Linux

Tools used

PyPro uses following languages and open source libraries:

- SWI-Prolog - A declarative language we used for writing parser and runtime.
- Python - A language we used to do handle frontend part and lexical analysis.
- Sly - A python library we used to do lexical analysis.
- Pyswip - A python library we used for interfacing between python and prolog.

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.