- A Programming Language Inspired by Scratch.mit.edu

WHY WE MADE PEDESTAL ?

Before this it is good to know about Esoteric Languages.

What are basically Esoteric Languages ?

ESOTERIC LANGUAGES

An esoteric programming language is a programming language designed to test the boundaries of computer programming language design, as a proof of concept, as software art, as a hacking interface to another language, or as a joke.

SOME FAMOUS ESOTERIC LANGUAGES

- Acronym
- Arnold C
- Shakespeare Programming Language
- Whitespace
- LOLCODE

Acronym

Code

```
{{>>{~~~~{-<}~~~~~~~~~{-<-<}}<<}
</(<<<){[<]}:>:{>>{~~~~~~~~{<}~{>}}<<}\
~>{{~{v}}>>>v{~}^<<<}/(<<<){[<<]}:>:{>>{~~~
~~~~~{<<}~{>>}}<<}\
~{>>{vvvvvvvv~~~~~~~~~~~~~~~}<<}~{>>{v
vvv~~~~~~~~~~~~~}<<}~
{>>{^^^^^^^^^^~}<<}~{>>{v~~~~~~~{{<<}~}
v{~}vvvvvvv{~{>>}}^^^^^^~~~
{{<<}~}vvvv{~{>>}}v~~~}<<}~<{{^^^}}~
{>>{vvvvvv~{{<<}~{>>}}^^^^^^^~~~~~~~~~
~{{<<}~{v>}}^^^^^^}}
/{{()}}{[<<<<]}:>:{{~v}}\}
```

PEDESTAL FEATURES

Pedestal is designed as Natural Language, in English. But keeping programming concepts in mind the structured is designed so that the user can understand the programming fundamentals.

Easy to read

Pedestal syntax is easy to read and comprehend. Basically we are more.

Programming Fundamentals

The syntax will help new learners, kids and school students to understand the basic programming fundamentals.

Starter for School kids

Coding in High Level Languages will be easy to understand when user interactive syntax of pedestal.

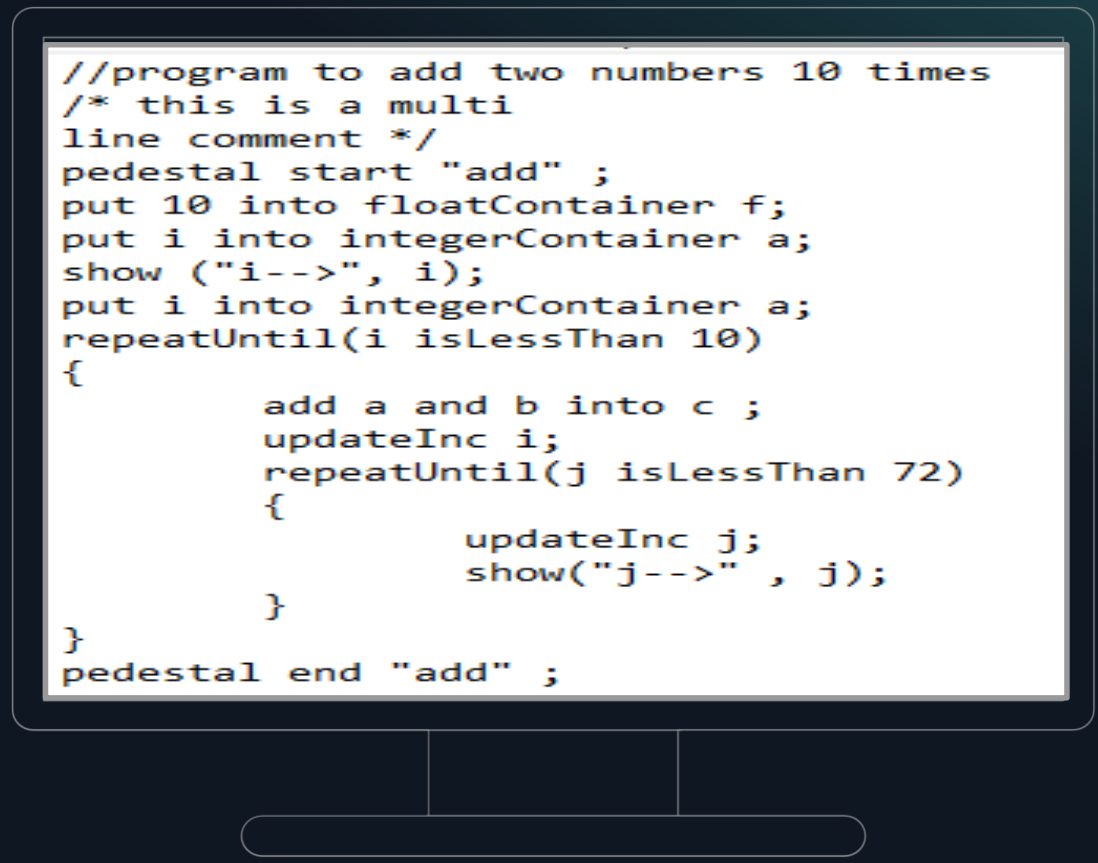
Counter to Esoteric Languages

As, Esoteric Languages are hard to understand and code. Pedestal is a counter to it. It Is easy to code and understand. And it is not implemented as a joke basically.

PEDESTAL PROGRAM

A pedestal program is divided into three sections.

- Header of Program
- Body of Program
- Footer of Program



```
//program to add two numbers 10 times
/* this is a multi
line comment */
pedestal start "add" ;
put 10 into floatContainer f;
put i into integerContainer a;
show ("i-->", i);
put i into integerContainer a;
repeatUntil(i isLessThan 10)
{
        add a and b into c ;
        updateInc i;
        repeatUntil(j isLessThan 72)
        {
                updateInc j;
                show("j-->" , j);
        }
}
pedestal end "add" ;
```

PEDESTAL CFG

How a Pedestal File is Written ?

1

Header

Header is required for the main( ) section of the file to be complete.

2

Body Section

Body section can be empty with no statement or it can have only one statement or can have multiple statements. Statement include loops, nested loops, variable declaration, print command , operations, function declaration, function call and functions definitions.

3

Footer

Footer is also required in pedestal File. It mentions the end of the file and completes the main section of file.

How Pedestal works up to Syntax Analyzer Phase.

There are Three main aspects of pedestal.

1 - First
Input of source file

First the Input file
of a Pedestal programs is given to
The lexical analyzer.

2 - Second
Tokenizing of Source File

This Lexical Analyzer generates
Valid Tokens and store them in a data
Structure.

3 - Third
Mapping of Tokens on grammar

The Tokens generated are mapped on
Grammar of our language. And correct
Syntax is Parsed.

Valid Tokens

Syntax Analyzing

FLEX REGULAR EXPRESSIONS

These are the Flex Regular Expressions that generates Tokens.

KEYWORDS

```
"pedestal start" {yylval.identifier = strdup(yytext); return PEDESTAL_START;}
"pedestal end" {yylval.identifier = strdup(yytext); return PEDESTAL_END;}

"into" {yylval.identifier = strdup(yytext); return INTO;}
"put"  {yylval.identifier = strdup(yytext); return PUT;}
"integerContainer" {yylval.identifier = strdup(yytext); return INT_CON;}
"floatContainer" {yylval.identifier = strdup(yytext); return FLOAT_CON;}
"stringContainer" {yylval.identifier = strdup(yytext); return STRING_CON;}
"repeatUntil" {yylval.identifier = strdup(yytext); return FOR_LOOP_KEYWORD;}
"and" {yylval.identifier = strdup(yytext); return AND;}
"show" {yylval.identifier = strdup(yytext); return PRINT;}
```

FLEX REGULAR EXPRESSIONS

OPERATORS

```
"isLessThan" {yylval.identifier = strdup(yytext); return FOR_LOOP_COND;}
"add" {yylval.identifier = strdup(yytext); return ADD;}
"sub" {yylval.identifier = strdup(yytext); return SUB;}
"updateInc" {yylval.identifier = strdup(yytext); return UPDATE;}
```

FLEX REGULAR EXPRESSIONS

SEPERATORS

```
[;] {yylval.identifier = strdup(yytext); return SEMICOLON;}
[(] {yylval.identifier = strdup(yytext); return OPEN_BRACKET_ROUND;}
[)] {yylval.identifier = strdup(yytext); return CLOSE_BRACKET_ROUND;}
[{] {yylval.identifier = strdup(yytext); return OPEN_BRACKET_CURLY;}
[}] {yylval.identifier = strdup(yytext); return CLOSE_BRACKET_CURLY;}
[,] {yylval.identifier = strdup(yytext); return COMMA;}
```

FLEX REGULAR EXPRESSIONS

INTEGER , FLOAT, STRING LITERALS AND IDENTIFIER

```
[0-9]+ {yylval.integer_num = atoi(yytext); return INT; }
[0-9]*"."[0-9]+ {  yylval.float_num = atof(yytext); return FLOAT; }
\"(\\.|[^"\\])*\" {yylval.string_literal = strdup(yytext);  return STRING;}
[a-zA-Z_][a-zA-Z0-9_]* {yylval.identifier = strdup(yytext); return IDENTIFIER;}
```

FLEX REGULAR EXPRESSIONS

IGNORING WHITESPACES, TABS, NEW LINES, SINGLE LINE COMMENTS AND MULTI-LINE COMMENTS

```
[' '\t\n]+ {  }
"//".*  {  }
[/][*][^*]*[*]+([^*/][^*]*[*]+)*[/]  {  }
. { }
```

CFG OF PEDESTAL IMPLEMENTED IN BISON

```
%%
// grammar rules of pedestal language
pedestal: header body_section footer
{ cout << "End of file!" << endl; };
// the syntax is as " pedestal start string_literal ; "
header: PEDESTAL_START STRING SEMICOLON
{   cout << "reading a pedestal program named as -> " << $1 << $2 << $3 << endl;
    //fprintf(yyout,"reading a pedestal program named as : %s %s %s \n", $1,$2,$3);
};
// body section(prod) can have body statements or nothing
body_section: body_statements | ;
// body statements can be comprises of single body statement or multiple body statements
body_statements: body_statement | body_statements body_statement;
// now body statement can be declerations, loops, operations or print statements
body_statement: declerations | loops | operations | printstatements; | functions
```

CFG OF PEDESTAL IMPLEMENTED IN BISON

```
// these body statement productions are defined below.
// declerations can be of int or float or string
declerations: integer_dec | float_dec | string_dec;
// loops can only be of for_loop
loops : for_loop;
// operations are of add or sub or updation(increment)
operations : addition | subtraction | updation;
// print statement start
// the syntax is as " show("string_literal" , ID) ; "
printstatements : PRINT OPEN_BRACKET_ROUND STRING COMMA IDENTIFIER CLOSE_BRACKET_ROUND SEMICOLON
{
  cout << "bison found an print decleration as : " <<endl;
  cout << "->" << $1 << " " << $2 << " " << $3 << " " << $4 << " " << $5 << " " <<  $6 << " " << " " << $7 <<endl ;
};
// print statement ends
```

CFG OF PEDESTAL IMPLEMENTED IN BISON

```
// declerations starts
// the syntax is as "put integer_num into integerContainer con_name ;"
integer_dec: PUT INT INTO INT_CON IDENTIFIER SEMICOLON
{ cout << "bison found an INTEGER decleration as : " <<endl;
  cout << "->" << $1 << " " << $2 << " " << $3 << " " << $4 << " " << $5 << " " <<  $6 << " " << endl ;
};
// the syntax is as "put float_num into floatContainer con_name ;"
float_dec  : PUT FLOAT INTO FLOAT_CON IDENTIFIER SEMICOLON
{ cout << "bison found an FLOAT decleration as :" <<endl;
  cout << "->" << $1 << " " << $2 << " " << $3 << " " << $4 << " " << $5 << " " <<  $6 << " " << endl ;
};
// the syntax is as "put string into stringContainer con_name ;"
string_dec  : PUT STRING INTO STRING_CON IDENTIFIER SEMICOLON
{ cout << "bison found an STRING decleration as :" <<endl;
  cout << "->" << $1 << " " << $2 << " " << $3 << " " << $4 << " " << $5 << " " <<  $6 << " " << endl ;
};
// declarations ends
```

CFG OF PEDESTAL IMPLEMENTED IN BISON

```
// for loop starts
// the syntax is as "repeatUntil ( ID isLessThan INT ) { body_section } "
// the for loop body can have single or multiple body statements
for_loop : FOR_LOOP_KEYWORD OPEN_BRACKET_ROUND IDENTIFIER FOR_LOOP_COND INT
CLOSE_BRACKET_ROUND OPEN_BRACKET_CURLY body_section CLOSE_BRACKET_CURLY
{
    cout << "bison found a for loop decleration as : " <<endl;
    cout << "->" << $1 << " " << $2 << " " << $3 << " " << $4 << " " << $5 << " " <<  $6 << " " << " " << $7 << " " << " " << $9 << endl ;
}
;
// for loop ends
```

CFG OF PEDESTAL IMPLEMENTED IN BISON

```
// operations starts
// the syntax is as "add ID and ID into ID ; "
addition : ADD IDENTIFIER AND IDENTIFIER INTO IDENTIFIER SEMICOLON
{
  cout << "bison found an addition decleration as : " <<endl;
  cout << "->" << $1 << " " << $2 << " " << $3 << " " << $4 << " " << $5 << " " <<  $6 << " " << " " << $7 << endl ;
}
;
// the syntax is as "sub ID and ID into ID ; "
subtraction : SUB IDENTIFIER AND IDENTIFIER INTO IDENTIFIER SEMICOLON
{
  cout << "bison found an subtraction decleration as :" <<endl;
  cout << "->" << $1 << " " << $2 << " " << $3 << " " << $4 << " " << $5 << " " <<  $6 << " " << " " << $7 <<endl ;
}
;
// the syntax is as "updateINC ID ; "
updation : UPDATE IDENTIFIER SEMICOLON
{
  cout << "bison found an updation decleration as : " <<endl;
  cout << "->" << $1 << " " << $2 << " " << $3 << endl ;
}
;
// operations ends
```

CFG OF PEDESTAL IMPLEMENTED IN BISON

```
//footer starts
// the syntax is as " pedestal end string_literal ; "
footer: PEDESTAL_END STRING SEMICOLON
{
  cout << "terminated a pedestal program named as -> " << $1 << $2 << $3 << endl;
  //fprintf(yyout,"terminated a pedestal program named as : %s %s %s \n", $1, $2,$3);
}
;
//footer ends
%%
```

THANK YOU!