

Aadila Jasmin 2019A7PS0801U  
Anukriti Jaiswal 2018A7PS0254U  
Srilatha Bondala 2018A7PS0204U

## ***Ez* LANGUAGE - SPECIFICATION FILE**

### **(EVERYTHING IN RED TEXT TALKS ABOUT PHASE 2 PROGRESS)**

A Language Translator from *Ez* to Python

We have designed a user-friendly language *Ez* (from “Easy”) based on the existing Python language. Python is one of the most widely used languages due to its ease of understanding and readability for beginners. Ez, too is dynamic and strongly-typed. We wish to include different aspects such as high performance, readability and convenience in our programming language *Ez*.

Basic features of *Ez*

- Python is a dynamically typed language and one of its biggest drawbacks is the high chances of errors at run time and difficulty in debugging and error handling. *Ez* will be a **statically typed language** to make error-handling easier and less memory consumption.
- Ez is **case sensitive** and hence any change in case will result in error.

**Hello World Program in *EZ***

```
int helloworld = 5.  
write(helloworld).  
write($helloworld).  
o/p:  
helloworld  
5
```

// \$ means the value of the variable will be taken.

**1.1 INDENTATION - To differentiate different blocks in the program**

In python, the different blocks of the program are separated by using indentation. This becomes difficult for the user to identify when the program size is large.

In Ez, we will be using curly braces ( { } ) to differentiate between different blocks of the program (for example: body of the loop)

As discussed we have used { } for indentation in Ez

## 1.2 COMMENTS

Comments are defined using the symbol # as in Python

Comments in Ez is also using #

## 1.3 VARIABLES

- In EZ, we will have the following **data types**:  
Integer - int (integer values)  
Float - float (floating point values)  
Character - char (characters)  
String - string (Sequence of characters)  
Array - box (described in the section 1.8)

All the data types as discussed above are Implemented.

```
%token<data_type>INT
%token<data_type>CHAR
%token<data_type>FLOAT
%token<data_type>DOUBLE
%token<data_type>STRING
```

- **Declaration**

In *Ez*, we will have the following declaration

Datatype VariableName = Value.

Eg:

integer x= 10.

String name= Shanaya.

There is **no variable declaration in Python**, you just assign a value to a variable. This often creates a confusion and may lead to overwriting

For example:

```
a=10 //lost data
```

```
.
```

```
.
```

```
. //few statements later
```

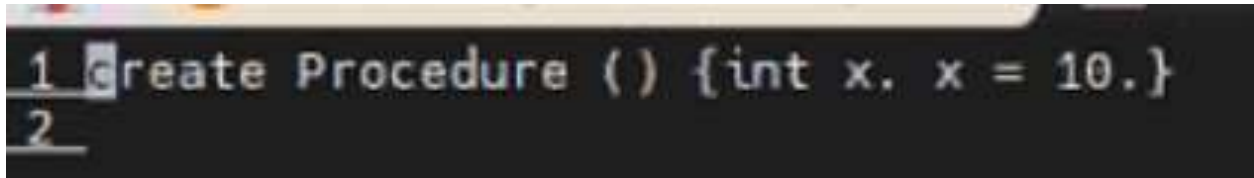
```
a=15
```

```
print(a).
```

```
o/p: 15
```

Hence, to avoid loss of data, **Ez** will include declaration and will not allow reuse of variables.

We have shown initialization and declaration however no access specifiers.



```
1 create Procedure () {int x. x = 10.}
2
```

#### 1.4. End of Statement

In **Ez**, . (fullstop) is used to indicate end of statement (It will help users to easily distinguish between statements)

In Python, there is no separator used to indicate separation between statements or end of statement.

We have used full stop as our end of statement differentiator. Without full stop it will show error.

#### 1.5. OPERATORS

- a. = is the assignment operator as in Python
- b. ? is used for comparison

Example:

```
1?1.
```

```
o/p= True.
```

- c. +, -, \*, /, (, ), % have the same operation as in Python.

- d. The @ symbol will behave like a **reverse operator** with strings and numbers

Example:

```
>write(@letshavefun)
o/p nufevahstel
>write(@614)
o/p 416
```

- e. ^ is for exponential use

Example:

```
>2^3.
o/p 8
```

In Python, it would be 2\*\*3

- f. \\ indicates a new line (In python would be \n)  
g. \$ means the value of the variable will be taken and the string won't be taken as a literal.

#### Operations on binary numbers-

// Initially, we will only consider binary inputs and will try to incorporate a decimal to binary converter.

- h. (x) denotes xor operation (a (x) b. In python it would be a^b;)

For Example:

```
1010 (x) 1111.
o/p 0101
```

- i. Shift left operation <-

Example:

```
write(<- 0111).
o/p: 1110
```

- j. Shift right operation ->

Example:

```
write(-> 0110)
o/p: 0011
```

All other bitwise operations remain the same as in Python.

- k. Logical and is denoted by "&&"

- l. Logical or is denoted by "||"

All the operators as mentioned above have been executed including the built-in operators.

```
create Procedure() {
```

```
int a.  
a=10.  
int b.  
b=7.  
Write($a^b).  
Write($a(x)b).  
Write($a->1).  
Write($b<-1).  
Write($a||b).  
Write($a&&b).  
}
```

```
A_EXPN:      A_EXPN LAND {printf("&");} A_EXPN  
              | A_EXPN LOR {printf("|");} A_EXPN  
              | A_EXPN LEQ {printf("<=");} A_EXPN  
              | A_EXPN GT {printf(">");} A_EXPN  
              | A_EXPN LT {printf("<");} A_EXPN  
              | A_EXPN NEQ {printf("!=");} A_EXPN  
              | A_EXPN DEQ {printf("==");} A_EXPN  
              | NOT {printf("!");} A_EXPN  
              | A_EXPN PLUS {printf("+");} A_EXPN  
              | A_EXPN MINUS {printf("-");} A_EXPN  
              | A_EXPN MUL {printf("*");} A_EXPN  
              | A_EXPN DIV {printf("/");} A_EXPN  
              | A_EXPN MOD {printf("%");} A_EXPN  
              | A_EXPN EXPC {printf("**");} A_EXPN  
              | A_EXPN XOR {printf("^");} A_EXPN  
  
              | A_EXPN SL {printf("<<");} ONE  
              | A_EXPN SR {printf(">>");} ONE  
              | TERMINALS
```

```
TERMINALS:      VAR {  
                  if((temp=lookup_in_table(yyval.var_name))!=-1) {  
                    printf("%s", yyval.var_name);  
                    //open temp file  
                  }
```

## 1.6. Conditional statements

- a. if (condition) do action statements

**if (condition)**

$$\{$$

action statements

$$\}$$
**orelse** action
$$\{$$

## Statements

$$\}$$
**orelse** action
$$\{$$

## Statements

}

## Other

{

## Statements

```
} //if all the else statements fail we can use other as a compulsory statement.
```

If – or else – other has also been implemented in our ez language.

[illegible]

## 1.7. Loop Structures

a. while (condition)

$$\left\{ \begin{array}{l} \text{Statements} \end{array} \right\}$$

While loop has been implemented in our program as well.

```
A_EXPN RB {printf("");\n");}
| WHILE LB {tab_count++; printf("while ");}
A_EXPN RB LCB {printf(": \n {\n");}
STATEMENTS RCB {tab_count--;print_tabs();printf("}\n");}
| VAR COLON {printf("\b\b\b\b\b\b\b\b\b\b%s:\n", yylval.var_name);}
| COMMENT {printf("# %s \n", yylval.var_name);}

<: IF LB { printf(" if ");}
```

The screenshot shows a terminal window with three tabs. The first tab is titled "2. homeinobastem". The second tab is titled "3. 172.18.22.5 (2019A7PS0001U)". The third tab is titled "5. homeinobastem". The terminal displays the following code snippet:

```
1 Create Procedure () {int x, x=1. While { x < 10 } { Write($x). x = x + 1. }}
```

The output of the program is visible as a series of numbers from 1 to 10, each followed by a newline character, resulting in ten lines of output.

## 1.8 Arrays

We will use the concept of boxes to define arrays in *Ez* language. (Often boxes are used to explain the concept of arrays, why not use boxes itself in the language?) `box` will be the keyword for creating array.

Syntax: ("box")[0|1|2|3|4|5|6|7|8|9]\* = { items }

Example:

```
string box1={ Pizza, Burger, Juice}.
```

integer box2= {14, 2, 6, 10}.

We have implemented arrays/ box in our language Ez however it is limited to only string type.

```

{tab_count--;print_tabs();printf("\n } \n");}

BOX_LIST: TYPE {if(current_data_type == 0)
    printf("#int internally declared \n ");
    else if(current_data_type == 1)
    printf("#char internally declared \n ");
    else if(current_data_type == 2)
    printf("#float internally declared \n ");
    else if(current_data_type == 3)
    printf("#double internally declared \n ");
    else if(current_data_type == 4)
    printf("#string internally declared \n ");
    for(int i = 0; i < idx - 1; i++){
        insert_to_table(var_list[i], current_data_type);
        printf("%s ", var_list[i]);
    }
    insert_to_table(var_list[idx - 1], current_data_type);
    printf("%s \n ", var_list[idx - 1]);
    idx = 0;
    BOXNUMBER {printf(" %s ", yylval.var_name);} EQ {printf("=");} LCB { printf("[");} VAR_LIST RCB EOS { printf(")");}

VAR_LIST:
    VAR {
        strcpy(var_list[idx], $1);
        idx++;
    }

```

## 1.9 Input and Output

type(datatype name) takes an input from the user and stores it in the variable name and datatype in the bracket.

print(variable|value) prints the value of the variable or directly the value given in the bracket.

type(String name). //takes user input and stores it in variable name

Example:

>type(Integer ID). //expects user to type and ID

>write(ID).

>write(hello!).

User i/p> 2019A70123U

o/p

2019A7PS0801U

hello!

We have changed our printing keyword from “type” to “typ” in order to avoid confusion as type can also be confused with data type . We have used Write for printing.





```
1 create Procedure () {int x. typ(x). char c. typ(c). float y. typ(y).}
```

## 1.10 ERROR HANDLING

Our language returns errors in different parts of the program based on the errors that it comes across.

## 1.11 FUNCTIONS

A function is a piece of code which runs and displays output when it is referenced. *Ez* has many inbuilt functions like `write()`, `type()`, etc.

In *Ez*, the format for creating functions is as follows,

```
Create functionName()  
{  
write(This is a test).  
}
```

As mentioned we use “Create” for creating functions in our language.

And this function can be called simply by using the function name.

```
functionName(). //how the function is called  
o/p: This is a test
```

In Python,

```
Def func1():
```

```
    print(“This is a test”)
```

//if the function becomes larger it becomes difficult to identify the block, hence *Ez* uses curly braces and full stop at the end of sentences.

```
func1()
```

```
o/p: This is a test
```

## 1.11 SPECIAL CRYPTOGRAPHY FUNCTION

A special function of *Ez* is an inclusive cryptographic function that converts Plain text to Cipher text and Cipher Text to plain text. This can be used if a user wants to display a number that can be understood only by a person with the key or only a person with the program can crack the code. It uses basic cryptographic algorithms.

```
encrypt(int number, int key) //encryption function
{
int number1 = number +key. //adds key
Int number2=@number1. //reverses the number
write(The ciphertext is ,number2). //outputs cipher text number
}
```

```
decrypt(int number, int key) //decryption function
{
int number1= @number. //reverses number
int number2= number1-key. //subtracts key
write(The plaintext is ,number2). //outputs plain text number
}
```

Example:

```
encrypt(213,30)
```

o/p

342

```
decrypt(342,30)
```

o/p: 213

This doesn't exist in python. We aim to first implement this using numbers and then further extend it to strings so secret messages can be sent.

We implemented both the cryptographic functions in our language however the key has to be fixed in the lexer final before implementing. We have used "1" as our key.



## Ez PROGRAM

### 2.0 The main procedure

Similar to python our language does not have any main function.

### 2.1 Ez Sample program

#### EZ

```
create Procedure(){
  int r;
  int s;
  s=0;
  int t;
  ty0(t);
  while(t>0)
  {
    r=t%10;
    s=s*r;
    t=t/10;
  }
  write($s);

  if (s < 102)
  {
    write($s);
  }
  orwline (s > 102)
  {
    write($t);
  }
  other
  {
    write($r);
  }
  @00111331221;
}
=
```

#### PYTHON OUTPUT

```
~$ ./mini-compiler < testinput.txt
def main(): {
    " r " #int internally declared r
    " s " #int internally declared s
    s=0
    " t " #int internally declared t
    t = int(input())
    while t>0:
    {
        r=t%10
        s=s*r
        t=t/10
    }

    print( s )
    if s<102:
    {

    }

    elif s>102:
    {

    }

    print( t )
    else :
    {

    }

    print( r )

    }

    print( str( 00111331221 ) [::-1]
    #reverse function is built-in in ez, whereas in python it is as given that is it uses slicing
    }
~$ ./mini-compiler < testinput.txt
```

## **FUTURE PROSPECTS**

We want *Ez* to be an extremely easy user-friendly language that can incorporate long functions with one symbol (for example “@” can be used for reversing numbers and strings). It should be a language that is suitable for first-time learners! We want it to incorporate the ease of the python programming language but also increase performance. However, python is dynamically typed which has several disadvantages like error handling.

Python is extremely slow and not feasible when it comes to memory intensive tasks and our language, *Ez* is statically typed language which helps to overcome these limitations. Moreover, the WORA (Write once run anywhere) concept makes it platform independent like Python. We implemented cryptographic functions which can be used to crack cipher numbers into plain text and can be extended for strings also in the coming future .All in all, Ez may be preferred over its counterparts for its readability, easy-to-use syntax and memory space efficiency and our team wishes to publish it after further improvement.

## **POINTS TO NOTE (in the learning process.)**

- Traversing through our arrays (box) for implement for loop as in python.
- Since Ez is static and Python is dynamic there was difficulty in translation due to certain factors like how python doesn't pre declare data types.
- The language is case sensitive.
- Simple string print has not been incorporated in our language
- “x” and “1” are used as keywords for xor operation and shift left respectively and hence can't be used as variables.
- Our encryption and decryption function takes a fixed key which is initialized in the lexer file, (we took 1), however the number to be encrypted can keep changing.