User Manual

This is called the lz (lazy) language. See instead of designing the logo I just highlighted the text to a certain blue.
It wants to be lazy and the keywords all are 2 letters each.
Then most symbols used are the ones that you do not need to press shift. It really tries to avoid typing at all except for a few end keywords which are a little troublesome at times.
Yup that pretty much explains all there is to it.

language design and first compiler by Jose Carlos Rodrigo Azcarraga

system requirements

-python 2.7.15rc1 should be installed
-has only been tested on ubuntu 18.04, so this version of ubuntu would be advised
-install PLY for python 2.7.15rc1 to be sure it will work
- to run, make sure all files are in the same folder, and open the folder in terminal, then type:
-the author requests files end with .lz but any file ending will do

```
python cidcompiler.py nameOfFileName.lz
```

basics

-basic keywords for print, if, else, while

pr (rp)

-for printing strings and expressions
-printing_stuff can be a string or expression or multiple strings, expressions separated by space
-strings are denoted by an apostrophe, ‘STRING’
-to print a new line an **nl** without a surrounding pr rp, on its own line is needed to produce a new line in the output

Usage

<printing keyword> printing_stuff <end printing keyword>

pr *printing_stuff* ***rp***

sample code:

```
pr ‘this is so cool kaya’ 1+1 ‘9+9+9’ rp
nl
pr ‘eyy’ rp
```

output:

```
this is so cool kaya 2 9+9+9
eyy
```

ip

-for inputting strings and expressions
-it will open up system input and ask for the input

Usage

<input keyword> <name>

ip *name_of_variable*

sample code:

```
ageNow = in 0

pr ‘what is your age right now’ rp
nl
ip ageNow
pr ’your age in 10 years is’ ageNow+10 rp
```

output:

```
what is your age right now
user_input>>21
your age in 10 years is 31
```

if (fi)

-for conditional statements
-can be multiline

Usage

<if keyword> <left bracket> conditional_statement <right bracket> statements <end if keyword>

```
if [ conditional ] statements fi
```

```
if [ conditional ]
      statement1
      statement2
fi
```

sample code:

```
if [1 < 2]
      pr 1 'is greater than' 2 rp
fi
```

output:

```
 1 is greater than 2
```

el (le)

-modified if statement to have an else to it
-can be multiline as well

Usage

<if keyword> <left bracket> conditional_statement <right bracket> statements <end if keyword> <else keyword> statements <else end keyword>

```
if [ conditional ] statements fi el statements le
```

```
if [ conditional ]
      statement1
      statement2
fi
el
      statement3
le
```

sample code:

```
if [1 < -3]
      pr 1 'is greater than' -3 rp
fi
el
      pr 1 'is less than' -3 rp
le
```

output:

```
1 is less than -3
```

wh (hw)

-while loop statement
-will run the statements inside as long as the condition still holds

Usage

<while keyword> <left bracket> conditional_statement <right bracket>
statements <while end keyword>

wh [*conditional* ***]*** *statements* ***hw***

```
wh [ conditional ]
      statement1
      statement2
hw
```

sample code:

```
a = in 1
wh [a <= 3]
      pr a rp
      nl
      a = a+1
hw
```

output:

```
1
2
3
```

\comments

-for comments
-has to be on a new line

Usage

<comment keyword> comments

**** *your comment*

sample code:

```
\this is a comment the program can’t catch me
pr 'there are hidden comments here' rp\im a ninja the program can’t see me
nl
pr 'dangit it evaded me' rp
\pr 'this wont run since it is after a comment keyword' rp
```

output:

```
there are hidden comments here

dangit it evaded me
```

declarations

-declaring strings, float, integers, arrays

st

-string declaration
-strings are denoted by ‘apostrophes’

Usage

<name> <equal sign> <string keyword> <apostrophe> string <apostrophe>

*name = **st** ‘stringtext’*

sample code:

```
a = ‘string ko si ’
pr a ‘a’ rp
```

output:

```
string ko si a
```

in

-integer declaration
-can be an expression but has to equate to an integer
-you need to do a declaration whether int/string/float when creating a variable

Usage

<name> <equal sign> ***<in keyword>****<expression that becomes an int>*

*name = **in** number*

sample code:

```
a = in 1
b = in 2

pr ‘a + b is ’ a+b rp
```

output:

```
a + b is 3
```

fl

-float declaration
-can be an expression but has to equate to an float
-you need to do a declaration whether int/string/float when creating a variable

Usage

<name> <equal sign> ***<float keyword>****<expression that becomes a float>*

```
name = fl number
```

sample code:

```
a = fl 1
b = fl 2

pr 'a / b is ' a/b rp
pr 'a // b is ' a//b rp
```

output:

```
a / b is 0.5
a // b is 0
```

array

-array declaration
-no special keyword to denote it is an array (the brackets are enough)
-can declare an empty array as well
-can hold multiple types
-can also use this to replace an array with a whole new list

Usage

<name> <equal sign> <left bracket> array list <right bracket>

*name = **[** element1,element2…**]***

sample code:

```
a = [1,2,3]
pa a
b = [4,5,6]
a = b
pa a
a = []
pa a
```

output:

```
[1, 2, 3]
[4, 5, 6]
[]
```

array operations

pa

-prints the entire array

Usage

<print array keyword> <name>

pa *name*

sample code:

```
a = [1,2,3]
pa a
a = [4,5,6]
pa a
a = []
pa a
```

output:

```
[1, 2, 3]
[4, 5, 6]
[]
```

ln

-returns the length of the array

Usage

<length keyword> <name>

*name = **[** element1,element2…**]***

sample code:

```
a = [1,2,3]
pu a 4 up
pa a
pr 'array a has length ' (ln a) rp
```

output:

```
[1, 2, 3, 4]
array a has length  4
```

pu(up)

-pushes a value to the end of the array

Usage

<push keyword> <name> <expression> <push end keyword>

pu *name expression* ***up***

sample code:

```
a = []
num = in 0
wh [(ln a) < 5]
      pu a num up
      num = num + 1
      pa a
hw
```

output:

```
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
```

po

-pops the last element of the array and returns a value
-if you want to simply pop you still need to assign it to a variable

Usage

<pop keyword> <name>

po *name*

sample code:

```
a = [1,2,3,4,5]
num = in 0
wh [(ln a) > 0]
      pr 'will pop the value' po a rp
      pr 'the array still has the values ' rp
      pa a
hw
```

output:

```
will pop the value 5 the array still has the values
[1, 2, 3, 4]
will pop the value 4 the array still has the values
[1, 2, 3]
will pop the value 3 the array still has the values
[1, 2]
will pop the value 2 the array still has the values
[1]
will pop the value 1 the array still has the values
[]
```

to

-returns the last value of an array
-treated as an expression

Usage

<top keyword> <name>

to *name*

sample code:

```
a = [1,2,3,4,5]
num = in 0
wh [(ln a) > 0]
      pr 'top is' po a rp
      pr 'the array still has the values ' rp
      pa a
      num = po a
hw
```

output:

```
top is 5 the array still has the values
[1, 2, 3, 4, 5]
top is 4 the array still has the values
[1, 2, 3, 4]
top is 3 the array still has the values
[1, 2, 3]
top is 2 the array still has the values
[1, 2]
top is 1 the array still has the values
[1]
```

em

-returns true or false depending if an array is empty or not

Usage

<emptyArray keyword> <name>

em *name*

sample code:

```
a = [1,2,3,4,5]
pr 'array has the values ' rp
pa a
pr 'is the array empty?' em a rp
nl
a = []
pr 'array has the values ' rp
pa a
pr 'is the array empty?' em a rp
```

output:

```
array has the values
[1, 2, 3, 4, 5]
is the array empty? False
array has the values
[]
is the array empty? True
```

operations

```
an <- and
or <- or
no <- not
```

as is

from other languages C/Java/Python

+	addition (can add arrays and strings)
-	subtraction
*	multiplication
/	division
//	floor division
^	exponent
%	modulo
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	is equal
!=	is not equal

logical expressions can be wrapped around brackets [ ]
arithmetic and other expressions can be wrapped around parenthesis ( )

errors

assigning a variable value without declaring it

code:

```
a = 1
```

error:

```
Undeclared variable a at line no  1
```

assigning a float (doesn't end with .0) to an int declared variable

code:

```
a = in 1.111
```

error:

```
The input at line no 1 is not an int
```

assigning a string to an int/float declared variable (it requires an expression and string isn't an expression)

code:

```
a = fl 'this is not a float'
```

error:

```
Syntax Error in input!
```

using a nondeclared variable

code:

```
b = fl 1.111
b = b + a
```

error:

```
Undeclared variable a at line no  2
```

declaring with a declared variable

code:

```
b = fl 1.111
b = fl 1.112
```

error:

```
Variable name b at line 2 already in use, please use another name
```

accessing an array element of a non-array
(also same error for pushing,popping, checking if array is empty, checking top of array, array updating)

code:

```
b = fl 3.1415926
b = b[1]
```

error:

```
Variable b is not an array. Error at line 2
```

accessing beyond the array's capacity

code:

```
b = [1,2,3,4,5,6]
a = in b[7]
```

error:

```
Array index out of bounds exception at line 2 accessing value 7 but array only
until 5
```

wrong grammar usage (lacked an rp)

code:

```
b = fl 0.00001
pr 'my chances of failing in this MP are ' b
```

error:

```
Syntax error in input!
```

Accessing an array with a non-integer

code:

```
a = [1,2,3,4]
pr a[2.1] rp
```

error:

```
Wrong array index type at line1
```

Arithmetic error, dividing by zero

code:

```
pr 1/0 rp
```

error:

```
Arithmetic Error, cannot divide by 0
```

special nums

```
wa   0
oh   1
to   2
ti   3
fo   4
fi   5
si   6
se   7
ei   8
ni   9
```

This was the inspiration for the language. that is why all keywords are two letters. These were implemented as well to the language

`tiohfoohfi

-returns the integer version of the special num
-can be used as an integer

Usage

<grave accent> <special number word form>

` *wordformofnumber*

sample code:

```
pr `tiohfoohfi rp
pr `tiohfoohfi + `oh rp
```

output:

```
31415
31416
```

`tidoohfoohfi

-returns the float version of the special num
-can be used as a float

Usage

*<grave accent> <special number word form with a **do** inside>*

```
` wordformofnumberwithdo
```

sample code:

```
pr `tidoohfoohfi rp
pr `tidoohfoohfi + `oh rp
```

output:

```
3.1415
4.1415
```

`3.1415

-returns the word version of the number
-can only function as a string

Usage

<grave accent> <special number form in digits>

` *numberform*

sample code:

```
pr `31415926 rp
pr `3.1415926 rp
```

output:

```
tiohfoohfinitosi
tidoohfoohfinitosi
```

thank you and so far that is the end of my user manual
(im serious I want to make this become more than a class requirement)

if you wish to contribute to this language or express support in developing this language or if you know other lazy programmers that will want this

please contact
jjazcarraga@up.edu.ph
or
+63 998 549 7697