MEG Programming Language

By: Merle Crutchfield and Avram Parra
CSC 372

Design Choices:

Wanted to make a language that was easy to use and understand

- Follows similar conventions
- Can accomplish many of the same tasks
- Maybe even express those tasks simpler?

Wanted to borrow from other languages which incorporate concepts that make code easier to read and write.

- Such as Type Casting in Java
- Branching statements for ease of understanding

Design Choices: Starting MEG

- After brainstorming we decided to create MEG
    - Includes a Type Casting system similar to Java
    - Has simpler implementations for basic concepts. Like Loops
    - Has necessities such as nesting for running loops and ifs

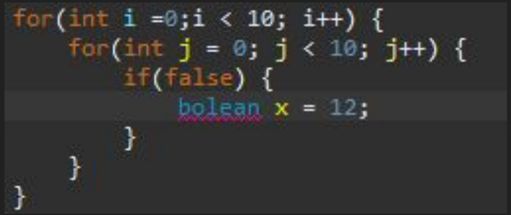
- Wanted to make MEG using Python as the base language
    - Flexibility of Python would allow us to more diverse language
        - For instance, Python’s handling of types allow us to easily add Type Casting
    - Various modules and foundation to take advantage of
        - While modules like RE (Regular Expressions) were not used. Python’s use of modules like SYS and OS allow for our language to take in Command Arguments at the terminal

Design Choice Challenges: Error Handling

Error Handling: Error Handling was a more difficult challenge than we originally thought.

If code is translated into our source code, Python, then how will un-ran translation code be flagged for errors like in other languages? ex. Java

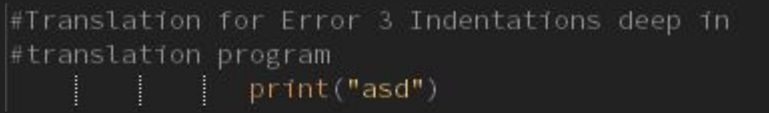
Example Picture: Although not reached, still error →



```java
for(int i =0;i < 10; i++) {
    for(int j = 0; j < 10; j++) {
        if(false) {
            bolean x = 12;
        }
    }
}
```

While we found that keeping track of indentation helped with our nesting of loops and conditionals. We also thought that it would help catch errors as they should be printed whenever Python comes across them.

However, this would create an Indentation Error. See picture →



```python
#Translation for Error 3 Indentations deep in
#translation program
            print("asd")
```

So how can we print the user’s error when in N indentation spots so that we catch it instead of Python?

Solution: If any error is found within the program, we just put the error message at the very start start of the program and wipe all previous data.

Design Choice and Creation Challenges:

One of the largest problems in creating our translator and one that we knew we would face was the problem of Parsing and translating our program to Python without the use of the Regular Expressions module.

With the Regular Expressions module this would still be hard.

Realized that through type casting, the locations of the types relative to their names could be used.

This along with the use of an indentation system along with unique keywords such as “_int_” made us better able to parse and translate written code in MEG.

A challenge we faced more on the design side was the structure of our language.

We wanted to create an easy to understand language but that still was different

Had to think of intuitive ways or ways in which we could combine other pieces of languages.

One example of this combination, was the use of “.” instead of “;” to end lines inspired by Prolog.

Language: Variables

MEG has three different types:

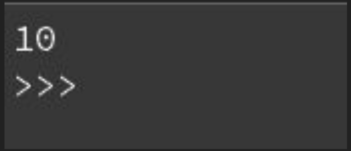
In the form of int , boolean , and string.

When defining a variable types must be Prefixed and Suffixed with a “__” (double underscore) tag.

Example: __int__ <var_name> -> <integer_val>.

__int__ test -> 10.

After running this line and running show(test). →



```
10
>>>
```

Can be done with booleans

__bool__ boolvar -> false.

This can also be done with strings

__string__ stringvar -> (abcd).

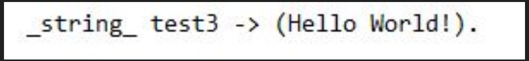
White text color text are examples of MEG Code.

Language: Keyword and Restrictions

In MEG there are several restrictions that must be followed in order to define variables and creating a working program in general.

Terms where things are defined must end with a ”.”

Instead of “=” MEG uses the sign “->”



```
_string_ test3 -> (Hello World!).
```

In MEG the keyword show() is used with either nothing or a parameter passed into it in order to print to the console

Example: __int__ x -> 10.

show x.

Language: Generalities

As stated when defining integer variables, it can be done through many ways:

Example One: _int_ x -> 10.

Example Two: _int_ y -> 10+ 10. This Case shown in photo

This Is also the case for _string_ variables :

Example One: _string_ x -> (abc).

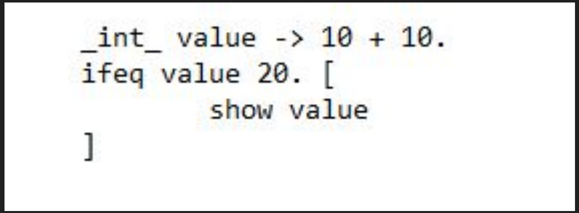
Example Two: _string_ x -> y + a.

And this also goes for _boolean_ variables:

Example One: _bool_ test -> true.

Example Two: _bool_ test5 -> ! true | test4 & true.

Where in Example Two the logic expressions are done before assignment.



```
_int_ value -> 10 + 10.
ifeq value 20. [
        show value
]
```

Language: Loops

Looping in MEG is done through the use of For loops.

For loops in MEG are defined through:

```
for <var_name> -> <val1><val2><val3>. [
    *stuff*
]
```

Where <val1> is checked to be less than <val2> and then incremented by <val3> after running *stuff*.

<var_name> must start with a character

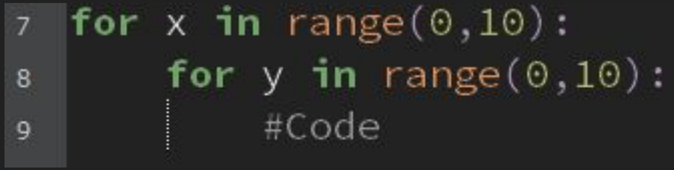
<val> , <val2>,<val3> must be an integer or an already initialized int

While for loops can be used on their own, MEG also allows for nesting.

Nested for Loop Example:

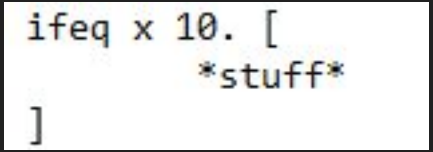
```
for varx -> 0 10 1. [
    for y -> 0 10 1. [
        *stuff
    ]
]
```

← Same loop expressed in Python shown in Picture below



```
for x in range(0,10):
    for y in range(0,10):
        #Code
```

Language: Conditionals


```
ifeq x 10. [
        *stuff*
]
```

Conditionals in MEG can be done through 6 different ways

Each for a certain logical operator similar to branching

The 6 types are: ifneq, ifgt, iflt, ifgteq, iflteq, ifeq

A general conditional statement is stated through:

General Example:

```
<if> <var_name> <integer_val>. [
    *stuff
]
```

Real Example:

```
ifneq y 27. [
    show(x).
]
```

<if> here denotes any type of the 6 ifs stated above.

Similar to for loops these can also be nested

Language: Creating and Running Your File

Prior to translating your program Python must be installed on your machine.

To translate and compile your program run this command in your Terminal:

```
python3 meg.py yourfile.txt
```

Where “yourfile.txt” would be the name of your file

Then in order to run your program run this command in your Terminal:

```
python3 output.py
```

Where “output.py” is the name of the newly created file after running the first line

Additionally, there are several command line arguments which can be included

Shown on the next slide, but can be used without them

Language: Command Line Arguments

With MEG Command line arguments can be used to run code

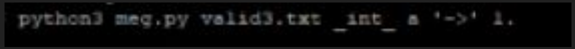
MEG arguments can be written through using the command:

python3 meg.py yourfile.txt *YourCommandArguments

Where *yourCommandArguments can be a statement like: _int_ x ’->’ 10. _int_ y ‘->’ 10.

NOTE: The single quotes are the markers in the statement shown. This is needed the program to work correctly.

Here is a live picture of using the command arguments with a program.



```
python3 meg.py valid3.txt _int_ a '->' 1.
```

I am not sure why this comes out blurry.

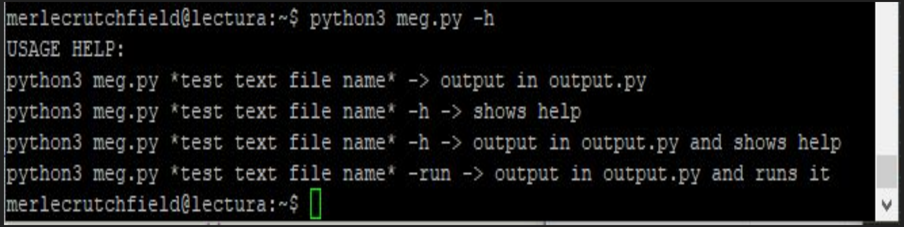
Language: Command Line Arguments Continued

Our language includes the use of Command Line Arguments

Exist in the forms of *None,”-h,”-run”

An example command shown on line 1 →

*None represents No command passed.



```
merlecrutchfield@lectura:~$ python3 meg.py -h
USAGE HELP:
python3 meg.py *test text file name* -> output in output.py
python3 meg.py *test text file name* -h -> shows help
python3 meg.py *test text file name* -h -> output in output.py and shows help
python3 meg.py *test text file name* -run -> output in output.py and runs it
merlecrutchfield@lectura:~$
```

*None uses the program as normal

“-h” tag is used to print the Help Message for the program

Can be used with or without a translation file, ex. “Python3 asdf.py -h”

“-run” is used to the run the output program immediately after creation

Post-Thoughts

After finishing our language we noticed there was vast potential for expanding the language and including various concepts and elements present in other languages.

One of these things we would have liked to expanded on was classes and along with these, private and public fields, a common concept in many languages such as Java. For example, in the process of making our program we found that many elements of a language we thought were very hard to implement were not that difficult to implement. If we were able to further expand on our language and make this design choice of adding classes and accessor attributes, we possibly could have made our language more flexible and capable similar to other languages.

MEG Programming Language

By: Merle Crutchfield and Avram Parra
CSC 372