

Jeffery Zhu, UTEID: JZ995
Lee McCuller, UTEID: LPM284
CS 315H
10/02/06

Program Assignment 3 - CritterInterpreter

Problem description (CRITTERia):

The problem is to interpret a critter language file in to a computer readable data structure, verify its correctness, and then execute the interpreted code.

Solution Description:

Writing:

For the implementation we had to decide on a proper data structure to hold the parsed code. We were restricted to storing code in a list format by the interfaces provided. We opted to using a polymorphic instruction hierarchy. This allows all instructions to be treated the same during execution.

This was implemented by creating an Action superclass from which all other instruction running objects descend from. We created a list of subclasses, each representing a possible line of instruction. Each Action subclass object stores its arguments, if needed, and has a Validation() method to insure proper arguments, for example bearings and headings must be multiples of 45 and registers must be between 1 and 10. During execution, each Action subclass is referenced from the index of the Critter's List of code and its Do() method is called which performs the action appropriate.

The parsing operates by sorting the instructions by their number of arguments and then using a series of if statements to find the proper Action subclass to construct and add to the code list.

"Tales":

We had to reorganize the parse function several times for readability and to handle each instructions special cases such as go's relative line numbers and the r's prefixing the registers

Testing:

Our first test of the interpreter was to use the Action subclasses toString() methods to output the parsed code. If parsing worked then the outputted code should match the original without its comments at the end of the critter file.

The second test was to create a bare Critter implementation and then execute the code using predicted returns from that Critter. For example, the hop instruction sent "the critter hopped" to standard out and ifenemy was always true.

The third test was to create critters that used all of the instructions in their first line and have them turn on some condition. If the instructions failed so would the condition. We used the previous Critter implementation to run each of these test cases.

The fourth case was to test the corner case for code running off the end. We had to fix this case with a specialized Action Subclass to output a warning and loop to line 1.

We finally ran the given critters in the test GUI. Since then constant test critters for submission have verified robustness.

Pair Programming Experience:

The pair programming experience was an enjoyable one.

Results:

The test GUI is really cool. Figured out that the GUI is terrible for testing because of the large number of critters it runs and the fact that it must be restarted to load new critter code or to link with new interpreter code.

Log:

Thur 21nd

Both drove about the same amount

between 10m - 20m

random take overs

7-9:30

battery life of laptop limited time spent

Tue 26

Lee drove more (3/2 as much perhaps)

same time

fewer random takeovers

code refactoring

15-25 min drives

Thur 28

7:15 - 7:40 Jeff

7:40 - 8:00 Lee

8:00 - 8:15 Jeff

8:15 - 8:30 Lee

Talked a bit

ran test cases for each instruction

both of us created creatures

fixed minor bugs (only had 2)

Monday Oct. 1

started at seven

both looked over code, fixed indentation, added comments where needed

wrote report

Independent;

Lee worked on karma critter compiler for 3-4 hours

Lee Has worked on compiled critter code for about 6 hours Tuesday night

Karma:

The critter code compiler is an incredibly basic macro expansion tool. It also can turn labels into line numbers. It uses multiple passes to extract function names and their code and formats them so that instruction replacement can occur. It then expands function calls iteratively. This is because they are essentially text macros. When function calls are recursed then each pass expands each recursed function call. I (Lee) Have implemented several features into this compiler. Because it extracts function definitions into its internal metadata, It

can include external files to extract functions from. The implementation is pretty straightforward except for the way it handles label names in functions. Because it does text expansion, the compiler must make each of its functions labels unique or there will be conflicts after function expansion. To do this all I did was create a special variable call {unique} that works in any function. This variable is designed to expand to a unique text at every function call. To make the labels unique all I did was append the unique variable to each name and each call so that the labels are made unique.

In doing this appending I used another trick. I had already written the label processor for the function extracted code, so I wanted to re-use it. What I did was make the label replacing functions take a special class. In the normal usage the class passed to the label processor replaces labels with blank lines and label call with the line number. In the function extractor, however, the special class passed replaces labels with unique labels and it replaces label calls with the appropriate unique label call.

To use the code compiler the command is

```
java Critterpreter File.in File.out #recursions
```

the # recursions is optional but probably necessary for any advanced critters. You can also append function libraries as additional object but that feature is obsolete now that #include statements work.

the Critterpreter java files are Critterpreter.java, TextFunc.java, and LabelMod.java so you can distinguish them from the others

Critter Strategy:

The basic strategy is to create a number that represents the current danger level of the critter based on the enemies around it. It can then create a second number for the danger it will be in if it hops. If it can not turn to infect and enemy before the enemy can kill it, then it hops if the danger is decreased. Before critterfest it will do many more things that are hard to do in a reasonable amount of time. The critters will clump to form defensive structures. They will turn away from friends if they know they will be infected next turn (partially implemented). They will also chase down and destroy enemies that run away. And when they run, they will turn so as to not be chased. These are very advanced things to implement currently however so they will have to wait until critterfest.

Also, The critter will be much more efficient once I put in an algorithm so that it can figure out that it is pointing orthogonally or diagonally, its enemy danger tests need that data to give better results. It will also help with structure forming. There is already support for orthogonality knowledge in the code but it just cant figure it out yet. Luckily the orthogonality data can be passed easily via infection with a couple of simple tests