Final Report: Use-ful Interpreter

By David Johnwen Hung
UID: 604191130
For CS C137B Programming Language Design
CodeBase: https://github.com/david1hung/useful-interpreter

Introduction

The #use-ful interpreter is my exploration of how the REPL environment can be simplified to be more intuitive to use. The need to manually import the file needed was an annoying process so I wanted to add an auto-import feature upon save. Sometimes we type things into the interpreter to test our code, but don't end up saving these "test-cases" so I wanted to add a way to simply copy and paste these code into a testCase file and have them automatically become test cases. Overall, I just found the annoyance, or time-wasters, in the REPL environment and wanted to add some scripts to fix it up.

Main Files

Name	Description
code.ml	The file where the programmer would type code into

testCases.ml	The file where the programmer would copy the test cases into. The test cases would just be the input and output line in OCaml interpreter
	e.g. One test case
	`# rev [1;2;3];;`
	`- : [3;2;1]`
	or Multi-line case
	`# let rec rev n =`
	`match n with`
	`[] -> []`
	`| first::rest -> rev(rest)@[first] ;;`
	`val rev : 'a list -> 'a list = <fun>`
	`# rev [1;2;3];;`
	`- : int list = [3; 2; 1]`

checkSave.sh	The listener script that checks if code.ml is updated and automatically opens an OCaml interpreter if it does.
openTerminal.sh	Uses AppleScript to open new terminal, and run the interpretor. Called by checkSave.sh
openInterp.sh	Opens the OCaml interpretor with the file passed
checkTest.sh	Listens on whether or not testCase.ml or code.ml is updated. Calls runTest.sh if it they are
runTest.sh	Runs generateTest.py to generate tests, runs the tests in an OCaml environment, calls displayResults to format results, and opens the final HTML output
generateTest.py	Reads testCases.ml to generate an OCaml file that can be run and check the success or failure of the tests
displayResults.py	Reads the output of the tests run and generates an HTML output that shows green for tests that passed and red for failed.
init.sh	Automatically setup the environment. Calls the open command on mac. And opens "code.ml" "testCases.ml" "testResults.html" and the 2 terminals that are listening to updates

Generated Files

Name	Description
tests.ml	test file generated by generateTests.py. contains the setup code required by each tests and runs an assert on the final line. Writes the results to testOutput.txt e.g. `if (rev [1;2;3] = [3;2;1]) then printToFile "Success" else printToFile "Failed";;`
testOutput.txt	Displays the Success or fail of the tests. Test 1: Success Test 2: Failed
testResults.html	Displays the tests results in HTML, with the original testCases input marked as green for success or red for fail.

Setup

Auto: Run init.sh from the Example directory.

Requires Mac command open. Will open 2 ml files in the default text editor, 2 terminal windows, and the default browser.

Manual

- Open code.ml (main code base to edit)
- Open testCases.ml (test cases code)
- Run ./checkSave code.ml
  - This listens to whether or not code.ml is updated
- Run ./checkTest testCases.ml code.ml
  - This listens to whether or not testCases.ml or code.ml is updated, in order to automatically run or re-run the testCases.
- Done!
- Type some code in code.ml and click Save

Sample Workflow

1. Code.ml is open, programmer writes a new function. Clicks Save
2. An new OCaml interpreter is automatically opened with Code.ml preloaded. It will provide the usual error reports if there are syntax errors.
3. The programmer proceeds to type code to test the functions in Code.ml. E.g. rev[1;2;3], the results is - : [3;2;1];
4. The programmer copies the input/output line from the interpreter into testCase.ml and clicks save. (i.e. he copies # rev[1;2;3] \n - : [3;2;1] into testCase.ml.
5. Upon save, a test-runner task is generated for this new test case, the test is run, and the results are displayed in an HTML format with the original input, green as success and red as failed.

Final Report Questions

Implementation:

*** What aspects of the design are currently implemented?**

My second rendition of my design was much simpler than the first in that I removed the GUI interface drag-and-drop and made it copy-paste text. So I was able to implemented basically all the features I described. My two main features was auto-import and test generation, and I was able to get solid, although not robust, implementation of both of the features.

For the auto-import feature, as I demo-ed before, I used a kqwait script that listens in on file updates, and calls commands to open terminal functions and run the interpreter. It also utilized Applescript so it appears gimmicky sometimes. But I like features as it is now, and I'll probably use it in the future when working different projects where I can things a lot and need to re-import. Parts of it is hard-coded, although not to hard to configure. So this would be the next development step, handling different file types (e.g. ml, py, c).

For the test generation features, I thought that the most intuitive thing for people to do is to copy the exact input and output of the interpreter and make that into a test case. So I used Python to parse test cases, forming a list, and then generated a OCaml file that runs the test case lines and outputs the results. Then I used another python script to beautify it into an HTML that marks success cases as green and failed cases as red. You can see this in the demo video.

Basically I have the implementation all the way through, but lack in robustness, cleanliness of interface, and have some hard-coded parts.

*** What platforms / languages / etc. did you use for your implementation?**

So the main listener script I used was kqwait, by Sven Schober (https://github.com/sschober/kqwait). It was Mac only so I developed only on my Mac computer.

Then I have a mix of scripting languages such as Bash, Applescript, and Python. Bash and AppleScript were used to call the terminal and run commands, while Python was used to parse files, read test cases, generate test cases that can be run in the original environment, and output an HTML file with the results.

So all in all, I used OCaml, Python, Bash, AppleScript, and a kqwait script.

* Describe your implementation strategy. Did you write an interpreter? A source-to-source translator? How does it work?

My implementation strategy was to get the minimal viable product that improves each step of the REPL environment. I wrote 2 listen and execute scripts. And wrote the scripts that takes interpreter input/output and converts them into test cases that can actually be run in the target environment and display results.

Here's the first listener

CheckSave.sh

```
#!/bin/bash
# kqwait detects saves on $1 and $2,
# calls openTerminal.sh to open $1 in a terminal
echo "Checking $1"
while ./kqwait $1 $2; do
    echo "Save Detected"
    ./openTerminal.sh $1
done
```

It calls openTerminal.sh

```
#!/bin/bash
DIR=$(pwd)
echo "Opening New Terminal Window"

osascript -e 'tell application "terminal"' -e "do script \"cd '$DIR'; ./openInterp.sh $1 \" " -e 'end tell'
```

Which opens a new terminal and launches the OCaml interpreter using openInterp.sh

```
#!/bin/bash
ocaml -init $1 #code.ml
```

The second listener was also similar

checkTest.sh

```
#!/bin/bash
# kqwait detects saves on $1 and $2,
# calls op.sh to open $1 in a terminal
echo "Checking $1"
while ./kqwait $1 $2; do
    echo "Save Detected on $1"
    ./runTests.sh
done
```

But it calls a different script that runs the python code and open the final HTML output

runTest.sh

```
#!/bin/bash
echo "Generating Tests"
python generateTests.py
echo "Running Tests"
ocaml -init 'tests.ml'
echo "Formatting Results"
python displayResults.py
echo "Displaying Results"
open testResults.html
```

Run test basically calls each of the python scrip and finally opens the testResults.html file to display the results visually.

*** What parts of the implementation were harder/easier than expected? Why?**

I originally thought the save-listener was going to be the hardest part, requiring me to write low-level code. However, I was able to find the kqwait script and that simplified things a lot so I just needed to write the scripts around it. I probably would have had some sort of infinite loop listening using bash and file name checks otherwise.

The part that was harder to implement was test case generation. Building the parser was not that easy because I really wanted the testCase file to be able to direct copies from the interpreter. I didn't use an AST tree but just used a testCase class that takes in setup lines and result lines. Then I had to figure out how to make clean test cases runs that I can display nicely. I tried out OUnit, an OCaml unit test program, but it was a little to complex for my purposes. So I ended up building a naïve implementation that uses =’s and if else. And then I ran short on time trying to do the visualization, so it's just simple coloring. It would have been really nice to use something that displays like CS C137A unit tests.

Design:

*** What aspects of the design worked well / did not work well, and why?**

I guess this is more of implementation but the using Python for the parsing and generating output worked really well. The string libraries provided a way to parse the lines fluidly and generating output was not too hard once I decided to use simple HTML. I haven't really used Python in a while and using it in this project made me realize just how clean and powerful it is. It kept things really simple and allowed me to explore more on how I wanted to parse instead of how to parse. AppleScript on the other hand was kind of annoying to use because it wasn't really intuitive.

So the design of auto-import and auto-test generation was the core components of my design, and just testing it out felt like it I could have used this to save time in many of my projects. Something like this, in a cleaner and organized way, should definitely be around. Auto-import felt pretty fluid, as the terminal just opens by itself upon save. However, I didn't get it to totally work because it opens multiple terminals as you save. I think this is fixable by 2 more lines of AppleScript. The test case copying and generation process was also pretty good. Copying the input and output lines directly from the interpreter is a reasonably intuitive process and could very well save people time.

So initially I designed to have a nice visualization of the test results, but I ended not having enough time to explore in this area and just used HTML. It would have been more intuitive if it was a CS 137A type of unit test display. And maybe allowing editing in the test cases to apply to the test case file would be useful too. I didn't think too much about the visualization of the design and this would be another area to explore. Also the grouping of test cases right now is really naïve, just numbering based on their position. This was also an area I didn't fully think through in my design.

*** What design aspects did you try but then discard, and why?**

I originally intended to have use a unit-test program such as OUnit so I can have a good environment to run the tests on, and have additional information, but it turned out to be a bit hard to use for my prototyping. Using a unit-tester would have provided me with tools to show how test cases goes from success to fail or how many of them succeed or not. I could probably just implement this part on my own, but using a unit-tester would have provided a lot more error report. Additionally, I only have one copy of the test results going on, so another thing I wanted to try out was to show how some test cases went from success to fail after you made changes in the code. This would have been done probably by a diff of test logs, but I didn't get to it because of time.

I guess the most obvious thing I discarded was the original GUI interface when I did my One-Pager 2, the #use-less interpreter. I imagined a drag and drop interface to generate test cases, but figured that it is not that big of a set up from copy and paste. So I decided to do that instead in order to get a good product working. However, I still don't have a good way to solve the dependency problems, like importing all the lines that the current line is dependent on, and just have a manual control, allowing copying multiple setup line before a test case as a I described in my video demo.

*** What challenges remain to make this design solve the intended problem?**

- Solving the test-case dependency issue, that is when generating test cases, automatically figure out the dependencies required and importing them. This would probably require a stack/log read and more inferences.
- Better test running system. Particularly test file failing when a function is unbound. This is a problem I didn't realize until I tried recording a video demo. My test-case running is really naïve right now in that if uses if [input == output] success else fail. So if the input is not valid code, then an error would be generated instead of failing gracefully.
- Better test visualization, I would need to improve from using HTML and think about the different information I want to display in test-cases. Maybe even allowing editing from the page.
- Better organization, right now there's 2 listeners, 2 files, 1 interpreter terminal, and one test visualization open, so that can get messy really fast. Ideally there will be a central system that organizes the positions of all these files.

*** What lessons did you learn more generally from the project?**

- I learned that a simple script like auto-import is actually very convenient and can save quite some time in the span of a project. I would definitely consider writing short scripts if I develop in a similar environment in the future.
- Test-generation is not easy to do, but should always be done. By making an easy-test generation system, new programmers can think about having test cases, without diving too deep into how to make them actually work. Also a lot of times we just spend time testing in the interpreter, and end up losing a lot of valuable test cases and results that we should try out later to make sure nothing breaks.
- Test-visualization is not easy. There is more information to show besides success and fail, and visualizing change over time is also important to understanding the whole system better.
- The name I chose is actually not that great because it's not really an interpreter! Maybe I can call it auto-bot for programmers!