BYU CS classes

Objective:

This lab is designed to help you begin to understand how to design and implement an interpreter for a simple language. The concepts you will practice here will be foundational to further labs!

Prerequisite:

You will need to download and install Julia for this lab. Note: if you are using Ubuntu 14.04, do **not** use the `apt` version of Julia (it's too old!). **You should use at least Julia 0.4**.

Note that the lexer required for this lab is available on LearningSuite, in the "Content" section, under the "Julia" subsection, on the page titled "Arithmetic expression interpreter and lexer".

You can also find these files in Dr. Wingate's home directory:

```
/users/faculty/wingated/cs330/julia_interpreters
```

You will also need the `Error.jl` file. You may choose to poke around with `Repl.jl`, which provides an interactive lisp-style REPL.

For code development, you may wish to use the IJulia [https://github.com/JuliaLang/IJulia.jl] plugin for Jupyter (which we used in class), or another IDE of your choice.

Deliverable:

For this lab, you will create a simple interpreter for a language we call OWL ("Our Widdle Language"). You need to create a Julia module that implements all requisite functionality.

Your module should export `parse` and `calc` functions, and a single type, `Num`.

Remember that you will use multiple dispatch to implement different "versions" of `parse` and `calc`, based on the input type.

An important difference between the code that you will implement for this lab, and the code we went through in class, is that your code should properly abstract multiple binary AST nodes into a single class we'll call `BinOp`.

Operator Table

Define a data-structure to contain a mapping from operator symbols to the functions that actually implement that symbol. For example:

```
Dict(:+ => +)
```

Parser

Write the following function(s):

```
function parse(expr)
```

`expr` will always be the output of the lexer, and will consist of numbers, symbols, and lists of numbers and symbols.

`parse` parses `expr` into an OWL datastructure according to this grammar:

```
  OWL          =              number
               |              (+ OWL OWL)
               |              (- OWL OWL)
               |              (* OWL OWL)
               |              (/ OWL OWL)
               |              (mod OWL OWL)
               |              (collatz OWL)
               |              (- OWL)
```

where number is a Julia number.

You should have test cases for all legal and illegal types of expressions. For example, the expression `(+ 1 2 3)` will pass the lexer just fine, but it is not accepted by our grammar – our grammar can only handle two arguments, not three! If you are given invalid input, you may throw an error, such as `throw( LispError("Whoa there! Unknown operation!") )`. This is defined in `Error.jl`.

You must parse `expr` into the following types (copy and paste this to the top of your code):

```
abstract OWL

type Num <: OWL
        n::Real
end

type Binop <: OWL
        op::Function
        lhs::OWL
        rhs::OWL
end
```

Plus any other subtypes you deem necessary.

Interpreter

Write the following function:

```
function calc(e)
```

Consumes an OWL representation of an expression and computes the corresponding numerical result.

Implementing mod

This interpreter has another binary operation called `mod`, similar to other binary operations. You should implement it in Julia using Julia's built-in `mod` function (note that this is **not** the same as the `%` in-line operator, at least for negative numbers!).

Implementing collatz

One of the fundamental "primitives" in our language is a function called `collatz`. This function is commonly used as an example of a function that is simple to write, but which cannot be analyzed – illustrating the difficulty of the general problem of program analysis. It comes from the Collatz conjecture [https://en.wikipedia.org/wiki/Collatz_conjecture]. Our collatz function returns the number of times the function recurses. Note that for some numbers, such as `n=28`, the function returns quite quickly – only 18 iterations – but for neighboring numbers, such as `n=27`, the function takes 111 iterations!

The function is defined as:

```
function collatz( n::Real )
  return collatz_helper( n, 0 )
end
```

```
function collatz_helper( n::Real, num_iters::Int )
  if n == 1
    return num_iters
  end
  if mod(n,2)==0
    return collatz_helper( n/2, num_iters+1 )
  else
    return collatz_helper( 3*n+1, num_iters+1 )
  end
end
```

Hints:

Note that the - operator has two distinct usages: as a binary operation (the "minus" operation) and as a unary operation ("negation"). You should create a distinct class to handle unary operations.

If you change a module, it can sometimes be tricky to convince Julia to reload it. I have found that the following two commands work:

```
workspace()
reload("ClassInt")
...
using ClassInt
```

---

cs330_f2016/laby.txt · Last modified: 2016/09/21 12:08 by wingated