MP4 - Forth

Logistics

- revision: 1.0
- due: March 13, 2017 23:59:59

Objectives

The objective of this MP is to implement a stack-based language. Such languages are often useful for embedded systems. (Postscript, the printer language, is an example.) The language we will implement is called Forth.

Goals

- Simulate a stack using a recursive function calls
- Create a function lookup table
- See a new language paradigm (stack based) and have fun with it.

Further Reading

To get more ideas about Forth, you can look at http://www.forth.org/tutorials.html. This page also seems particularly good. Note that we are implementing a greatly reduced version of the language!

Getting Started

How Forth Works

Forth is a stack-based language. All user-input is split on whitespace into a sequence of *words*. The words are checked and pushed onto a stack one at a time; they are interpreted specially if they are meant to be built-in operators.

- First it checks if the first input word is a special built-in operator. If so, the operator is handled.
- Then it checks if the word is in the dictionary of special symbols. If so, it loads the definition and runs it.
- If it's not in the dictionary it checks if it's an integer, and pushes it onto the integer stack if it is.
- If it's neither an integer or a dictionary word (or one a of a few special operators), it flushes the input stream and outputs an error message.

```
> 2 5 furbitz
Unknown symbol: 'furbitz'
>
```

Relevant Files

In the directory `app` you'll find `Main.hs` with all the relevant code. In this file you will find all of the data definitions, the primitive function maps, some stubbed-out simple parsers for special inputs, stubbed out evaluation functions, and the REPL itself.

Running Code

As usual, you have to `stack init` (you only need to do this once).

To run your code, start GHCi with `stack ghci` (make sure to load the `Main` module if `stack ghci` doesn't automatically). From here, you can test individual functions, or you can run the REPL by calling `main`. Note that the initial `$` and `>` are prompts.

```
$ stack ghci
 ... More Output ...
Prelude> :l Main
Ok, modules loaded: Main.
*Main> main
Welcome to your forth interpreter!
> 10 32 + .
42
> 2 3 + 5 6 + + .
16
> quit
Bye!
```

To run the REPL directly, build the executable with `stack build` and run it with `stack exec main`.

Testing Your Code

You will be able to run the test-suite with `stack test`:

```
$ stack test
```

It will tell you which test-suites you pass, fail, and have exceptions on. To see an individual test-suite (so you can run the tests yourself by hand to see where the failure happens), look in the file `test/Spec.hs`.

You can run individual test-sets by running `stack ghci` and loading the `Spec` module with `:l Spec`. Then you can run the tests (specified in `test/Tests.hs`) just by using the name of the test:

```
*Main> :l Spec
*Main> :l Spec
[1 of 3] Compiling Main             ( /home/ehildenb/work/cs421/su16/assignments/mp4-forth/app/Main.hs,
[2 of 3] Compiling Tests            ( /home/ehildenb/work/cs421/su16/assignments/mp4-forth/test/Tests.hs
)
[3 of 3] Compiling Spec             ( /home/ehildenb/work/cs421/su16/assignments/mp4-forth/test/Spec.hs,
Ok, modules loaded: Main, Tests, Spec.

--- Call an individual test-set
*Spec> tests_drop
[False,False,False]

--- or call `main`
*Spec> main
Exception: `splitIf` parser
Fail: show command `.S`
Fail: stack manip command `dup`
Fail: stack manip command `drop`
Fail: stack manip command `swap`
Fail: stack manip command `rot`
Fail: definition command `: ... ;`
Fail: if command `if ... [else ...] then`
```

```
Fail: loop command `begin ... again`

Score: 0 / 100
*** Exception: ExitFailure 1
*Spec>
```

Look in the file `test/Tests.hs` to see which tests were run.

Given Code

The setup code is concerned with importing the modules we will need and declaring the types we will use. We will be using `HashMap` to store a dictionary of definitions. You may also find the function `intercalate :: [a] -> [[a]] -> [a]` useful for printing out the integer stack.

```
import Data.List (intercalate)

-- for stack underflow errors
underflow :: a
underflow = error "Value stack underflow!"
```

The Types

The Forth machine will need to keep track of its state. It will have an integer stack for intermediate results (`IStack`), a call stack to keep track of subroutine calls (`CStack`), a dictionary for primitive and defined functions (`Dictionary`), and a place to keep track of output (`Output`).

```
type ForthState = (IStack, CStack, Dictionary, Output)
type IStack     = [Integer]
type CStack     = [[String]]
type Dictionary = [(String, [Entry])]
type Output     = [String]
```

The dictionary will store entries (`Entry`), which can be either primitive functions (`Prim :: (IStack -> IStack) -> Entry`) or defined functions (`Def :: [String] -> Entry`).

We also have data-constructors for numbers (`Num :: Integer -> Entry`) and unknowns (`Unknown :: String -> Entry`). This is so that the dictionary lookup function `dlookup` can signal the `eval` function that the lookup failed, and the input wasn't just a number.

We also provide a `Show` instance for `Entry`, which allows us to pretty-print things of type `Entry`.

```
data Entry = Prim (IStack -> IStack)
           | Def [String]
           | Num Integer
           | Unknown String

instance Show Entry where
    show (Prim f)    = "Prim"
    show (Def s)     = show s
    show (Num i)     = show i
    show (Unknown s) = "Unknown: " ++ s
```

Dictionary Access

Lookups

When `eval` uses `dlookup` and the lookup succeeds (the entry is present), the most recent definition of the entry will be used (front of the list of entries). If the list of entries is empty, or the lookup fails, then `dlookup` will try to convert the input to an `Integer` using `reads`. If that fails, it will signal `eval` that nothing worked by using the data-constructor `Unknown`.

```
-- handle input lookups (and integers)
dlookup :: String -> Dictionary -> Entry
dlookup word dict
    = case lookup word dict of
        Just (x:_) -> x
        _          -> case reads word of
                        [(i,"")] -> Num i
                        _        -> Unknown word
```

Insert

On inserting, we will simply add the new definition (value) to the front of the list of entries in the dictionary. In this way, you preserve past definitions in case you ever extend our language with the ability to revert to previous definitions. Our language will not have that extension yet, but you can think about how to add it.

```
-- handle inserting things into the dictionary
dinsert :: String -> Entry -> Dictionary -> Dictionary
dinsert key val dict
    = case lookup key dict of
        Just vals -> (key, val:vals) : dict
        Nothing   -> (key, [val])    : dict
```

Initial State

The initial state will have an empty integer stack (`initialIStack`), an empty call stack (`initialCStack`), a dictionary with primitive operator definitions (`initialDictionary`), and an empty output stack (`initialOutput`).

The `initialDictionary` is defined later in the Problems section because you must complete it as part of the assignment.

```
-- initial integer stack
initialIStack :: IStack
initialIStack = []

-- initial call stack
initialCStack :: CStack
initialCStack = []

-- initial output
initialOutput :: [String]
initialOutput = []

-- initial ForthState
initialForthState :: ForthState
initialForthState = (initialIStack, initialCStack, initialDictionary, initialOutput)
```

The Read-Eval-Print Loop

The read-eval-print loop handles all the action. It puts a prompt on the screen, reads some input, spits the input into words, feeds the words to the evaluator, and keeps track of the updated state. It also outputs anything that `eval` says should be output.

```
repl :: ForthState -> IO ()
repl state
    = do putStr "> "
         input <- getLine
         if input == "quit"
            then do putStrLn "Bye!"
                    return ()
            else let (is, cs, d, output) = eval (words input) state
                 in  do mapM_ putStrLn output
                        repl (is, cs, d, [])

main = do putStrLn "Welcome to your Forth interpreter!"
          repl initialForthState
```

Problems

Lifters

`liftIntOp`

To make our lives easier, we will use a function `liftIntOp` that takes a Haskell function and converts it into one that will work on the integer stack. This will be used to create primitive operators, which will be stored in the `initialDictionary` (see next problem for example).

Note the order that the arguments are given to the operator `op` in. Also note how we generate the `underflow` error if there are not enough entries on the input stack.

```
liftIntOp :: (Integer -> Integer -> Integer) -> IStack -> IStack
liftIntOp op (x:y:xs) = (y `op` x) : xs
liftIntOp _  _        = underflow
```

`liftCompOp`

You need to define `liftCompOp` so that we can have comparison operators between integers in our Forth language. In Forth, `0` is false and anything else is true, so you must return `0` if the result is `False`. You can return anything else otherwise (traditionally `-1` is used for `True`).

```
liftCompOp :: (Integer -> Integer -> Bool) -> IStack -> IStack
liftCompOp = undefined
```

The Dictionary

We have provided the entry for `+` in the `initialDictionary`. You must finish the rest of the definitions. Notice how we use `liftIntOp` to turn the Haskell function `(+)` into one that works on our Forth integer stack. Then we wrap the result of type `IStack -> IStack` in a `Prim :: (IStack -> IStack) -> Entry` so that we can hold it as a value in a `Dictionary`. Notice also that we put the `Prim ...` in a list (as `[Prim ...]`) so that we can keep a history of symbol definitions (as mentioned above).

```
initialDictionary :: Dictionary
initialDictionary = initArith ++ initComp
```

Arithmetic Operators

Provide the definition of the arithmetic operators subtraction (`"-"`), multiplication (`"*"`), and integer division (`"/"`). You will find the function `liftIntOp` useful here. We have provided addition (`"+"`) for you.

Comparison Operators

Provide the definition of the comparison operators less-than (`"<"`), greater-than (`">"`), less-than-or-equal-to (`"<="`), greater-than-or-equal-to (`">="`), equal-to (`"=="`), and not-equal-to (`"!="`). You will find the function `liftCompOp` useful here.

The Parser

Some input forms span many words of input and can be nested (for example the `... if ... else ... then` and `begin ... again` forms). To handle these, we'll need special input handlers which split the input sequence into a well-nested form and the rest of the input.

```
*Main> splitWellNested ("begin", "again") [ "3", "5", ".", "6", "<", "if"
                                          , "10", "else", "20", "then"
                                          , "again", "20", "50", "+"
                                          ]
(["3","5",".","6","<","if","10","else","20","then"],["20","50","+"])

*Main> splitWellNested ("begin", "again") [ "3", "5", "begin", "3", ".", "again"
                                          , ".", "6", "<", "if", "10", "else"
                                          , " 20", "then", "again", "20", "50", "+"
                                          ]
(["3","5","begin","3",".","again",".","6","<","if","10","else","20","then"],["20","50","+"])
```

Notice how the function `splitWellNested` keeps track of how many enclosing `start` and `end` tokens it has seen so that it knows it grabs an entire well-nested form. Also note that this function assumes that the initial start-token has been stripped off (notice there is no beginning `"begin"` in the above input stream), and removes the corresponding output-token from the result.

```
-- get the first well-nested string of tokens and the rest
splitWellNested :: Eq a => (a, a) -> [a] -> ([a], [a])
splitWellNested (start,end) words = splitWN 0 [] words
    where
        splitWN 0 acc (word:rest)
            | word == end    = (reverse acc, rest)
        splitWN n acc (word:rest)
            | word == start  = splitWN (n+1) (word:acc) rest
            | word == end    = splitWN (n-1) (word:acc) rest
            | otherwise      = splitWN n     (word:acc) rest
        splitWN _ acc []     = (reverse acc, [])
```

Input Parser `splitIf`

Implement the function `splitIf :: [String] -> ([String], [String], [String])`. It must also assume that the initial input-token `"if"` has already been stripped away, and it must strip away the trailing `"then"`

from the output. The output must consist of three lists of strings - the true and false branches of the `<cond> if <true branch> [else <false branch>] then` form (both possibly empty), as well as any input that followed the `if` form.

```
*Main> splitIf [ ".", "5", "7", "else", "3", "4", "<", "if"
               , ".", "else", "5", "then", "then", "5", "."
               ]
([".","5","7"],["3","4","<","if",".","else","5","then"],["5","."])

*Main> splitIf [ ".", "5", "7", "3", "4", "<", "if", "."
               , "else", "5", "then", "then", "5", "."
               ]
([".","5","7","3","4","<","if",".","else","5","then"],[],["5","."])
```

You may find it helpful to use `splitWellNested` in defining `splitIf`, at least to get the initial `... if ... then` clause. Then you can do further processing on the result to determine if there is an `else` branch.

The Evaluator

Next is the evaluator. It takes a list of strings as the next tokens of input, a Forth state, and returns a Forth state.

```
eval :: [String] -> ForthState -> ForthState
```

If the input is empty and there is nothing left on the call-stack, we are done processing input and should return the current Forth state. Notice how the output is reversed before returning it because it is built in reverse order when processing the input recursively.

If the input is empty, but the call stack is not, that means we should "return" to the previous entry in the call stack.

```
-- empty input and empty call stack -> return current state
eval [] (istack, [],       dict, out) = (istack, [], dict, reverse out)

-- empty input and non-empty call stack -> pop element off call stack
eval [] (istack, c:cstack, dict, out) = eval c (istack, cstack, dict, out)
```

We also handle one Forth form for you, the `.` operator. This operator consumes one element of the integer stack and outputs it. Notice once again how we handle the underflow case by generating the `underflow` error.

```
eval (".":words) (i:istack, cstack, dict, out)
    = eval words (istack, cstack, dict, show i : out)
eval (".":words) _ = underflow
```

Printing the Stack

Define `.S` which prints the entire stack. It does not consume the stack, however. It should print from bottom of stack to top.

```
> 2 3 4 .S
2 3 4
>
```

Stack Manipulations

Define `dup` (which duplicates the top of stack), `swap` (which swaps the top two elements), `drop` (which pops the top element without printing), and `rot` (which takes the third element and makes it the first element).

Make sure you handle the cases where the stack is empty by emitting the `underflow` error.

```
> 2 3 4 .S
2 3 4
> dup .S
2 3 4 4
> drop .S
2 3 4
> swap .S
2 4 3
> rot .S
4 3 2
> 3 dup rot .S
4 3 3 3 2
> 1 2 3 4 rot .S
4 3 3 3 2 1 3 4 2
```

User definitions

Now you are ready to add something interesting. Add the ability to define new words. The syntax is

```
: <word> <definition...> ;
```

The definition will be added to the dictionary of the Forth state. In the future when the symbol `<word>` is encountered, the definition will be looked up in the dictionary and loaded as the input to the `eval` function. The current input to the function will be saved onto the call-stack so that it can be restored after the defined function is run. See below for how this is handled.

So, to make the square function:

```
> : square dup * ;
> 4 square .
16
```

So `4 square .` becomes `square .` (because `4` is pushed onto the integer stack), which becomes `dup *` when `square` is looked up in the dictionary (`.` is pushed onto the call-stack).

In the case where you get an empty definition (`[":", ";"]`), continue evaluation after the definition with an unmodified state.

Conditionals

Add conditionals. The syntax for conditions is

```
<condition> if <true brach> else <false branch> then
```

The `else` is optional.

The keyword order looks a bit different than in the languages you have been using. In real Forth, conditionals can only occur in definitions, but we will not bother restricting that here.

This one is tricky compared to the rest. When you read an `if`, you will need to modify the input stream to select the correct branch. This is why real Forth restricts this to definitions; real Forth will compile the addresses into memory rather than playing with the input.

Remember that you will be inspecting the top of the integer stack to determine which branch to take. A `0` is interpreted as false, and anything else is interpreted as true. If the stack is empty, make sure to emit `underflow`.

Notice how nested `... if ... else ... then` statements are handled correctly in the examples below.

```
> 3 4 < if 10 else 20 then .
10

> 3 4 > if 10 else 20 then .
20

> 3 4 > if 10 then .S


> 3 4 < if 10 then .S
10

> 3 4 < if 3 4 < if 10 else 20 then else 30 then .
10

> 3 4 < if 3 4 > if 10 else 20 then else 30 then .
20

> 3 4 > if 3 4 < if 10 else 20 then else 30 then .
30
```

Loops

One of the loop structures in Forth is as follows:

```
begin <code> again
```

The part between `begin` and `again` is executed repeatedly until a `exit` word is executed.

```
> 4 begin dup . dup 0 > if 1 - else exit then again
4
3
2
1
0
```

Use the call-stack to handle loops. When you hit a `begin`, push everything after the loop (after the `again`) onto the call-stack, then push the loop (including the tokens `begin` and `again`) onto the call-stack. Recursively call `eval` with the loop body (without the enclosing `begin` and `again`). Once the loop body finishes executing, the recursive call to `eval` will pop an element off the call-stack (which will be the loop again) and begin evaluating it.

If you hit an `exit`, you need to inspect the call-stack and pop elements off until you reach one that starts with a `begin`. That element of the call-stack is the next-enclosing loop, so it should also be popped off the call-stack. Then discard the rest of the current input and recursively call `eval` on the remaining state with an empty input.

Lookup in dictionary

Finally, if the input does not match one of the pre-defined Forth forms, `eval` will use `dlookup` to determine what to do with it. It could be that we get a number. If so, push it onto the stack. It could be that we get a primitive. If so, modify the stack by feeding it to the primitive. Be sure you understand the code that accomplishes this!

If instead `dlookup` says that its an `Unknown`, flush the input stream and call-stack and output that the symbol is unknown.

We have handled this case for you.

```
-- otherwise it should be handled by `dlookup` to see if it's a `Num`, `Prim`,
-- `Def`, or `Unknown`
eval (word:words) (istack, cstack, dict, out)
    = case dlookup word dict of
        Def def   -> eval def   (istack,   words:cstack, dict, out)
        Prim f    -> eval words (f istack, cstack,       dict, out)
        Num i     -> eval words (i:istack, cstack,       dict, out)
        Unknown s -> eval [] (istack, [], dict, ("Unknown symbol: " ++ s) : out)
```