Assignment 5: Monads in Haskell and Logic Programming in Prolog

CS 442/642

Due March 25^{th} in class

For this assignment, you will use GHC to complete part A, and SWI Prolog to complete the rest.

Part A—Monads

1. Write a Haskell program to play a simple guessing game. You will think of a number. Your program will ask for a guessing range, and then repeatedly guess numbers in that range until it either guesses the number, or can prove that you are cheating. Each time your program guesses a number, you must either answer `yes`, `higher` or `lower`. A sample interaction follows (your responses shown after the colon(:)):

   ```
   Enter guessing range: 1 10
   Is it 5?: higher
   Is it 8?: lower
   Is it 6?: yes
   Got it!
   ```

   If the guessing range is down to one possibility, and you don't answer `yes`, then your program should output `Cheating!` instead of `Got it!`.

   Your program should employ binary search (with halves rounded down) on the guessing range to output an answer or accuse you of cheating in $O(\log n)$ time, where n is the size of the guessing range.

   Your guessing game should be called `guess`, and its type should be `IO ()`.

2. The following imperative ML program produces the first n elements of the Fibonacci sequence:

   ```
   val current = ref 0
   val next = ref 1

   fun fib 0 = []
   |   fib n =
     let val answer = !current
     in
        (current := !next;
         next := !current + answer;
         answer :: fib (n - 1))
     end
   ```

   Rewrite this function in Haskell, using the State monad from class.

   Hints: The state you maintain should be the counters `current` and `next`, so your state type should be `(Int,Int)`. Use the examples from class (especially the one that counts nodes in a tree) as inspiration.

Your `fib` function will have type `Int -> [Int]`, but it will likely call a helper function whose type will be `Int -> State (Int, Int) [Int]`. Your solution should start with the definition of the State monad, including the instance declaration, as given in class. You should not need to modify this part, and you can paste it in from the tree-labelling example that was posted on Piazza.

The Unlambda Programming Language

A popular recreational activity among hackers is the design of programming languages that are as difficult to use as possible, while (hopefully) remaining Turing-complete. Such languages have come be be known as *obfuscated programming languages*. Well known examples include Intercal, Befunge, and Malbolge; a web search would yield many others.

In this assignment, we focus our attention on a particular obfuscated language that enjoys the distinction of also being a *functional* language. This language is called *Unlambda*. We consider only a (Turing-complete) subset of Unlambda here; for information on the full language, see the creator's web site: `http://www.madore.org/~david/programs/unlambda/`

At its core, an Unlambda program consists of a string made entirely of the characters `s`, `k`, and `‘`. `s` and `k` are functions (or *combinators*). The symbol `‘` is an "apply" operator. It is a prefix operator, and since all functions in Unlambda are curried, it eliminates the need for parentheses. For example, we represent `(ab)c` as `‘‘abc`, and `a(bc)` as `‘a‘bc`. `s` is a ternary combinator with the property that, for any `a`, `b`, and `c`, `‘‘‘sabc` evaluates to `‘‘ac‘bc` (that is, `(ac)(bc)`). `k` is a binary combinator with the property that, for any `a` and `b`, `‘‘kab` evaluates to `a`. For example, consider the evaluation of the following Unlambda program:

```
‘‘‘skkk  →  ‘‘kk‘kk
         →  k
```

Unlambda is an applicative-order language—to reduce `‘ab`, we reduce `a` first, then `b`, and finally, we apply the reduced `a` to the reduced `b`.

The core language, consisting of `s`, `k`, and `‘`, are enough to guarantee Turing-completeness. However, Unlambda supports a number of other interesting primitives, outlined below:

- `i`—the identity function. `i` is a unary function, and `‘ix` returns `x` for all `x`.
- `v`—the "voracious" combinator. `v` is a unary function, and `‘vx` returns `v` for all `x`.
- `.x`—display function. This is actually a family of functions. `.x` is literally the function that prints the character `x` to the screen, but `x` may be replaced with any character. Note that `.` on its own is not a function. Also note that `.x` is a unary function, and does nothing until an argument is supplied. At that point, the function prints `x` and returns its argument. For example, `‘.*i` prints the character `*` and returns `i`.
- `r`—a special case of `.x`. `r` is a unary function that prints a newline and then returns its argument.
- `d`—"delay". `d` is a unary function; the application `‘dF`, where `F` is an unlambda expression representing the argument to `d`, produces a "promise" to evaluate `F`. Note that, unlike with normal function application, `F` is not evaluated before it is passed to `d`. Hence, for example, `‘d‘.xi` produces a promise to return `‘.xi`, but nothing is printed; if the promise is applied, as in `‘‘d‘.xii` (this reduces to the same promise as before, applied to `i`), Unlambda evaluates the argument to the promise *first*, the contents of the promise (`‘.xi`) *second*, and finally applies the reduced promise to the argument. For example, `‘‘d‘.xi‘ii` reduces as follows: `‘‘d‘.xi‘ii` → `‘(promise)‘ii` → `‘(promise)i` → `‘‘.xii` → (prints `x`) `‘ii` → `i`.

Unlambda also supports the primitive `c` (call-with-current-continuation), but we shall ignore this.

Part B—An "Unlambda Machine"

For part A, you are to construct an interpreter for the subset of Unlambda discussed above. To simplify the task, the input to your interpreter will be a list consisting of a sequence of Unlambda tokens. We will represent the `s`, `k`, `i`, `v`, and `r` operators in our input lists as the symbols `s`, `k`, `i`, `v`, and `r`, respectively. To avoid potential compatibility issues across Prolog implementations, we will represent ` as the symbol `a`. Finally, we will represent `.x` by `dot(x)`, where `x` represents the character following the dot in the input string.

The interpreter will work as follows: if the head of the input is anything other than `a`, then the interpreter terminates. Otherwise, it recursively reduces the tail of the list until it is no longer reducible (i.e. it begins with anything other than `a`), and then does the same with the tail of the tail of the list. At this point, the original list consists of `a`, followed by two characters other than `a`, and we can then perform the application requested by the initial `a`. Applications of the primitive operators proceed as follows:

- `[a, k, x, ...]` should be replaced with `[k(x'), ...]` after reduction of `x` to `x'`[1];
- `[a, k(x), y, ...]` should be replaced with `[x, ...]` after reduction of `y`;
- `[a, s, x, ...]` should be replaced with `[s(x'), ...]` after reduction of `x` to `x'`;
- `[a, s(x), y, ...]` should be replaced with `[s(x,y'), ...]` after reduction of `y` to `y'`;
- `[a, s(x,y), z, ...]` should be replaced with `[a, a, x, z', a, y, z', ...]` after reduction of `z` to `z'`;
- `[a, i, x, ...]` should be replaced with `[x', ...]` after reduction of `x` to `x'`;
- `[a, v, x, ...]` should be replaced with `[v, ...]` after reduction of `x`;
- `[a, dot(x), y, ...]` should be replaced with `[y', ...]` after *first* reducing `y` to `y'`, and *then* printing `x` to the screen;
- `[a, r, y, ...]` should be replaced with `[y', ...]` after *first* reducing `y` to `y'`, and *then* printing a newline to the screen.
- The reduction for `d` is for you to determine.

The predicate that runs your interpreter should be called `interpret`. Its first argument should be an Unlambda expression represented as a list; its second argument will be used for output of the reduced expression. For example, invoking `interpret([a,a,a,s,k,k,k], Y)` should result in a binding of `Y` to `[k]`.

Part C—Interpreting from Text

Write a predicate `interpretFromText` that takes two parameters. The first will be an Unlambda expression represented as a (double-quoted) string. The second will be the reduced Unlambda expression, represented in the format given in part A. `interpretFromText` will work by translating its input into the format required by `interpret` and invoking `interpret`. For example, invoking `interpretFromText("```skkk", Y)` should result in a binding of `Y` to `[k]`.

For a small bonus, you can arrange to have `interpretFromText` return the reduced term as a string, by translating the result of the interpreter back to a string before returning it. But note the following: if an Unlambda expression is an incomplete application (and therefore irreducible), like `` `ki `` or `` ``skk ``, your interpreter will return a closure—in this case `k(i)` and `s(k,k)`, respectively. As the correct answers are

[1] Note that the wording here is a bit imprecise, because the precise version is awkward. But in essence, when we say "after reduction of `x` to `x'`," we mean after reduction of (some prefix of the remainder of the list that starts with `x` and represents a complete subexpression) to `x'`. Similar comments apply to the other bullets.

really the unreduced `‘ki` and `‘‘skk`, you will need to look out for closures and translate them back into unreduced Unlambda expressions. Keep in mind that closures may be nested, and so a recursive expansion may be necessary. Similar comments hold for the treatment of `dot(x)`. If the result is (or contains) a promise, you can represent the promise in text as `<promise>`, or some other suitable representation.

Notes

- The “apply” character in Unlambda is a *back*quote (ASCII 96).

Part D—“Unlambdafication”

The Unlambda operators `s` and `k` were not chosen arbitrarily; they are actually Moses Shönfinkel’s “Verschmelzungsfunktion”, or S combinator, and “Konstanzfunktion”, or K combinator, both of which may be represented in the λ-calculus. The S combinator is $\lambda x.\lambda y.\lambda z.(x\,z)(y\,z)$ and the K combinator is $\lambda x.\lambda y.x$.

Schönfinkel showed that every closed term in the λ-calculus may be converted to an equivalent expression consisting only of combinations of S and K. In this way, we may actually remove all of the (bound) variables from an expression. This transformation, which we call “unlambdafication”, is presented below:

$$\begin{aligned}
\text{unlambda}[\![x]\!] &= x \\
\text{unlambda}[\![f]\!] &= f \\
\text{unlambda}[\![M\,N]\!] &= \text{unlambda}[\![M]\!]\,\text{unlambda}[\![N]\!] \\
\text{unlambda}[\![\lambda x.E]\!] &= [x](\text{unlambda}[\![E]\!])
\end{aligned}$$

The notation $[x]E$ denotes the *bracket abstraction*, and roughly means “remove the variable x from the expression E”. The bracket abstraction is presented below:

$$\begin{aligned}
[x]y &= \begin{cases} I & \text{if } y = x \\ K\,y & \text{otherwise} \end{cases} \\
[x]f &= K\,f \\
[x]I &= K\,I \\
[x]K &= K\,K \\
[x]S &= K\,S \\
[x](M\,N) &= S([x]M)([x]N)
\end{aligned}$$

In the definitions above, we use f to denote function symbols. These represent primitive functions, with which we augment the bare λ-calculus.

For Part C of your assignment, you will implement, in Prolog, a translator from the untyped λ-calculus to the Unlambda programming language. To do this, you will need a Prolog representation for λ-terms. The representation you will use is as follows:

- The variable x will be represented by the structure `var(x)`;
- the abstraction $\lambda x.E$ will be represented by the structure `abs(var(x), E)`, where `E` is our Prolog representation for the expression E;
- the application $M\,N$ will be represented by the structure `app(M,N)`, where `M` and `N` are our Prolog representations for the expressions M and N, respectively;
- the function symbol f will be represented by the structure `func(f)`.

For example, we would represent the λ-term $\lambda y.\lambda x.y(y\,x)$ in Prolog as

```
abs(var(y), abs(var(x), app(var(y), app(var(y), var(x)))))
```

You will support the following function symbols:

- `r`—prints a newline and returns its argument. For example, the expression `app(func(r),var(x))` prints a newline (after first reducing the expression `var(x)`—in this case there is nothing to do) and then reduces to `var(x)`;
- `dot(x)`, where `x` is any character—prints the character `x` and returns its argument. For example, the expression `app(func(dot("*")),var(y))` prints the character `*` (after first reducing the expression `var(y)`—in this case there is nothing to do) and then returns `var(y)`;
- `v` consumes its argument and returns `v`. For example, the expression `app(func(v), var(x))` reduces, after reduction of `var(x)`, to `func(v)`.
- `d`—"delay". See the description in part A of this assignment. For example, `app(func(d), app(func(r), func(i)))` returns a promise to evaluate `app(func(r), func(i))` (and prints nothing); `app(app(func(d), app(func(r), func(i))), func(v))` prints a newline and returns `func(v)`.

Your translator will be called `unlambdafy/2`. Its first parameter will be λ-expression to be reduced. Its second will be used for output, and will contain the translation of the the λ-expression to Unlambda. The output parameter should have format matching the input format for your solution to part A, so that the two predicates could be chained together, if desired.

Notes

- Take note of the spelling of `unlambdafy`.

Submission

Submit electronically:

- Part A solution (`a5a1.hs`, `a5a2.hs`).
- Solution for remaining parts as a single file (`a5.pl`).

Submit on paper:

- All source code.

For tips on Unlambda programming, see the creator's web page:
`http://www.madore.org/~david/programs/unlambda/`.
The relative weighting of the components of the assignment is 25% for part A, 35% for part B, and 15% for part C and 25% for part D. This assignment is due at the beginning of class on March 25^{th}.