Programming Languages — Homework 4
Functions

Due: Wednesday, 27 March 2013, 23:55

Feel free to use G+ to ask questions, to discuss corner cases, or to post interesting test cases.

This assignment will be graded out of 10 points.

1 Free and bound variables

[2 pts] Identify the free and bound variables in each of the following expressions:

1. $\lambda x.\ \lambda y.\ \lambda z.\ z\ x\ y$
2. $\lambda x.\ \lambda y.\ (\lambda z.\ z\ y)\ (\lambda x.\ z\ x)$
3. $(\lambda x.\ \lambda y.\ y)\ x$

2 Scoping and evaluation

[2 pts]

- Haskell is a statically scoped language. Write a Haskell function that would return different results if the language semantics were instead dynamically scoped. That is, the function should return one value if evaluation is statically scoped and a different value if dynamically scoped. Obviously, you can only test the former on the real implementation.
- JavaScript is a call-by-value language. Haskell is a call-by-need language. Write a JavaScript function that returns different results if the language semantics were instead call-by-need. Do not use `with` or `eval`.

3 Interpreter

[6 pts] In this assignment you will extend your JS interpreter with first-class functions. For this subset of JavaScript, your interpreter should have the same behavior as V8, the JavaScript engine used in Chrome and in Node.js.

The language

Your interpreter should implement the same features of JavaScript as in Homework 2 (booleans, numbers, operators, etc). In addition, it should implement anonymous functions like the following:

```
function (x) { return x + 1; }
```

For simplicity, all functions take exactly one argument and consist of a single `return` statement. They are really just λ-calculus lambda abstractions with JavaScript syntax. The only variables in the language are function formal parameters.

The language also supports function calls, for instance:

```
(function (x) { return x + 1; })(1)
```

or this implementation of factorial of 10:

```
(function (y) {
  return y(function (f) {
    return function (n) {
      return n == 0 ? 1 : n * f(n-1);
    }
  })(10)
})(function (f) {
  return (function (x) {
    return f(function (v) { return x(x)(v); })
  })(function (x) {
    return f(function (v) { return x(x)(v); })
  })
})
```

The JavaScript (ECMAScript 5.1) specification can be found at:

```
http://www.ecma-international.org/ecma-262/5.1/
```

You should use this document in order to fully understand the semantics of JavaScript expressions.

The output of your interpreter should match the output of V8. Small differences due to rounding of numbers are allowed. Also, equality and relational operations on function values are not implemented correctly—you can ignore these differences. The V8 implemention may, or may not, agree with the JavaScript specification. If there is a difference, the behavior should match V8, not the specification.

You can download Node.js here `http://www.nodejs.org`. You can find the JavaScript Console in Chrome in the "Developer" submenu of the "View" menu, or in the "Tools" submenu of the "wrench" menu in the toolbar.

Template

You should implement your interpreter by modifying the template `hw4.hs` provided on the course web site. The template provides a JavaScript parser an abstract syntax tree type and some utility functions. We have implemented many cases of expression evaluation for you. You need to complete the implementation. Look for `error "todo"` in the template. This is where you must add code. You shouldn't need to change other code, although you are welcome to do so. Your interpreter should implement function calls using substitution.

1. [1 pt] Complete the `fv` function, which returns the free variables of a given expression. This function is used to check that terms are *closed*—i.e., that they have no free variables—before evaluation. It is also used when implementing substitution. You may need to read the documentation on `Data.Set` on Hoogle.

2. [3 pts] Complete the `subst` function. You should only need to implement code for the expression types introduced in this assignment (`Fun`, `App`, and `Var`). There are some utility functions (`freshVar`, as well as `fv` from above) that may be helpful. Be careful that your `subst` function does not accidentally cause free variables in the argument to be captured.

3. [2 pts] Complete the `step` function. You should only need to implement code for the expression types introduced in this assignment.

Note that, as before, the provided parser does not accept the full set of JavaScript expressions. For instance, it does not handle numbers of the form "`1.0e-6`" and it may be missing other features. We will not test your interpreter on expressions the parser cannot already handle.

The interpreter accepts one command-line argument, a file name. The interpreter will read the file and evaluate the JavaScript therein. If no argument is provided, the interpreter will provide a REPL, similar to the Node.js REPL (but obviously supporting only the subset of JavaScript outlined above).

You should be able to run the interpreter as follows:

```
runhaskell hw4.hs file.js
```

To compile and run the template, you need to install the `System.Console.Haskeline` library. You can install packages using `cabal`, the Haskell package manager. To install the above library, you can run the following in your Unix shell:

```
cabal install haskeline
```

Submission

1. Complete the survey linked from the course web page after completing this assignment.
2. Submit your code and solutions on Moodle by 23:55 on Wednesday, 27 March 2013. Include your name in each file you submit.