Motivation

The idea behind our project was to experiment with creating a web language that would be easier to use than PHP and more suited for web development. All members of our group grew up learning PHP as one of our first languages. Unfortunatly this meant we thought we hated programming for a long period of time. Eventually we discovered other languages and much of the pain dissapated as we discovered things such as consistentency,

Today, however, PHP is ranked 6th in terms of popularity amongst programming languages\(^1\). It is impossible to avoid PHP when working as a freelance web developer, since maintaining other peoples code is almost invariably involves mucking around in some terrible PHP codebase. Thusly our idea of writing our own webserver and DSL for web development was spawned. Considering the disaster that is PHP, how hard could it be to make a web language that was less clunky and still well fitted to web development?

Project Outline

The goal was to create a web server in Haskell, and then build a simple DSL based on Common Lisp that would give us all the necessary tools to build basic web sites easily. We argue that anything based on Lisp, and implemented in Haskell would have a very hard time being worse than PHP. We kept our design as simple as possible, implementing only a small core subset of LISP required to accomplish tasks in a functional way. We also implemented a few easy to use functions that interact with basic web architecture, including a lightweight way to manage sessions, cookies, POST and GET.

We decided to call our DSL Sane Parmesan Badger, appreviated SPB from here on out, and using a file extension of spb. Below are a list of features SPB enjoys and some example code.

Features

**Basic language constructs**

Basic lisp constructs

- strings are lists (print “Hello”) => (H e l l o)
- List operations (first, rest, length, cons, append)
- quote [(quote (1 2 3)) or ’(1 2 3) ]
- defun (defun myFun (a b) (+ a b))
- lambda [(lambda (a b) (+ a b)) or ((a b) (+ a b)) ]
- do (like progn): (defun myFun (a b) (do (printString “Hello!”) (+ a b)))
- let with “improved” syntax: (let a 5 b 10 c 20 (+ c (+ a b))) => 20 * (5 + 10)
- if that also functions as cond (like in the Ark language):

\(^1\) [http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html](http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html)

Code located here: [www.github.com/saneparmesanbadger/SPB](http://www.github.com/saneparmesanbadger/SPB)
- `(if (> a b) 1 (> b 1) 2 50)` => If `a > b` then 1 else if `b > 1` then 2 else 50

- **read/eval/evalSafe/load**
- **print/printString**
- **show** (converts an s-expression to a string)
- **hashtables**
  - `(make-hashtable symbolName)`
  - `(hashtable-set symbolName sexpKey sexpValue)`
    - eg) `(hashtable-set myHashtable '(1 2 3) “One two three”)`
  - `(hashtable-get symbolName sexpKey)`
- **Arithmetic** (+ - * /)
- **Logical** (and or < > >= <= == not)
- **Random** (random 5)

**Curried functions, partial application**
All user defined functions are implemented as nested lambdas, allowing us to have partial application which is something that we have found sorely lacking in most lisps. For example:
```
(defun fireSomething (what howMany)
  (printString
    (append (append (append "Firing " (show howMany)) " ") what)))

(defun fireMissiles () (fireSomething “Missiles”))
(fireMissiles 500)
```

**In-Memory store, Sessions, Cookies**
The SPB.SPBWebLib module provides the ability to store and retrieve S-expressions directly in cookies and managed sessions, as well as the ability to store data in the server’s in-memory store. None of this functionality is available inside evalSafe (and thus in the web interpreter).

**Compile to Javascript**
Compiling to javascript is fairly straight forward: there is a base library written in javascript called SPBLib.js (incomplete at the moment, though functional) to handle base library calls, and the only real snags are renaming and functions are curried and support partial function application.

To give us partial application and currying, SPBLib.js has a javascript function `_$_apply which will apply a list of arguments to a function and either return a result or a partially-applied function implemented as a javascript closure.

Currently there is no dependency analysis done when compiling to javascript, so all user defined values in the symbolMap are presumed necessary and compiled automatically.
The online demo supports compile to javascript, so if we enter this program into the online interpreter:

```
(defun x (a b) (+ a b))
(printString (compile-to-javascript '(x 1 2)))
```

We will get back (reformatted and comments added):

```
//(defun x (a b) (+ a b))
var a$1 = function(a$2){
    return function(a$3){
        return _$add(a$2, a$3);
    }
}

var a$4 = true;
var a$5 = null;
//(x 1 2)
_$apply(a$1, [1, 2])
```

(Note that if you click “Run JS” in the online demo, be sure to open your browser’s javascript console to view the output.)

Variables a$4 and a$5 are unnecessary and eventually we will strip them from output.

Renaming is another tricky aspect, since sometimes a function is defined after it is needed. We use a state monad to keep track of a RenameMap in order to do the renaming:

```
type RenameMap = (Int, HM.HashMap T.Text [(Bool, Int)], HM.HashMap T.Text T.Text)
```

The Int element of the tuple is a variable name counter for producing new names. The second element of the tuple is a map of symbols to stacks of previous renames coupled with whether that renamed variable is used but not yet defined (the Bool) and the third element of the tuple is the map of renamings for library functions.

Renamings allow us to write code like this:

```
(defun x (a b) (z a b))
(defun z (x y)
    (let z 500
        (+ z (* x y)))
(printString (compile-to-javascript '(x 1 2)))
```

Where the ambiguity (x calling z before it’s defined, z having an argument named x, and z having the symbol z be re-bound inside its let to 500) will be automatically resolved with lexical scoping:

```
//(defun x (a b) (z a b))
```
var a$1 = function(a$2){
    return function(a$3){
        return _$apply(a$4, [a$2, a$3]);
    };
};

// (defun z (x y) (let z 500 (+ z (* x y))))
var a$4 = function(a$6){
    return function(a$7){
        return function(){
            var a$8 = 500;
            return _$add(a$8, _$mul(a$6, a$7))
        }();
    }; };);
 _$apply(a$1, [1, 2])

Note that the (let ...) is implemented by creating an anonymous function and then calling it immediately to get correct scoping.

Project Architecture

Lines of code breakdown:
ParserTest.hs  50
JavaScript.hs  225
BaseLib.hs     270
SPBWebLib.hs  30
Parse.hs       125
Interpreter.hs 250
Base.hs        75
Server.hs      150
Env.hs         75
HTTP.hs        200
SPB.hs         5
Test.hs        30
SPBWebLib.hs   150

----------------------------------
Total        1860

Server Architecture
The server (defined in SPB.Server.Server) first creates an Env record (defined in SPB.Server.Server) that contains STM references (TVars) to a hashmap for in-memory program storage (persistent between requests), session information, and a route map that caches programs (not yet implemented).

The server listens on a socket and whenever it receives a connection, it forks a haskell thread (ideally more scalable than spawning an OS-thread) that is passed the Env record.
The forked function then parses the HTTP request, and attempts to match the request URL with filenames found in spbMap.map. If it finds a match, it interprets the program found in the given file, giving it access to corresponding cookies, the in-memory store, and session information. Any haskell exceptions, parsing errors, or errors during program operation are propagated out to the browser.

**Interpreter Architecture** (See SPB/Interpreter.hs)

*Stack for lexical scoping*

```haskell
type SymbolMap m = HM.HashMap T.Text (SymbolValue m)
```

Lexical scoping is implemented using a list of hashtables from symbols to values ([SymbolMap m]). A hashtable gets pushed or popped from the list as we enter and leave a scope. This design unfortunately means that symbol lookups have worst-case \(O(n)\) performance in the depth of the stack. This could be mitigated by instead using a hashtable with symbols as keys and stacks of values as values.

**Functions and base library functions**

A SymbolValue can either be a built in function, a built in “macro” (same as a function but without arguments being evaluated), or a value from user code.

```haskell
data SymbolValue m =
    Function Int ([CExp] -> InterpM m CExp) -- ^ builtins
  | Macro Int ([CExp] -> InterpM m CExp) -- ^ quotes lambdas
  | SymbolValue CExp
```

The base functions (car, cdr, append, quote, lambda etc) are defined in SPB/BaseLib.hs, SPB/ Javascript.hs, and SPB/SPBWebLib.hs. All functions from BaseLib.hs and Javascript.hs have the type ([CExp] -> InterpM m CExp), able to work both in evalSafe and normal eval (see Interpreter Typeclass).

All user-defined functions are ALambda atoms (ALambda T.Text CExp (HM.HashMap T.Text CExp) (defined in SPB/Base.hs), that are closures (with the environment being the third argument to ALambda) allowing us to represent multi-parameter functions as nested lambdas.

**Interpreter Typeclass**

Much of the complexity of the interpreter is due to the fact that we wanted to have an “evalSafe” function in the language, quarantining the code so that it wouldn’t have any access to IO.

In order to do this and not write multiple versions of eval or base functions, we made the InterpM monad polymorphic in its base monad and created the Interpreter typeclass. This lets us create a versatile eval function that will work on code evaluated
safely or unsafely, using the type signature:

\[
\text{evalM} :\ (\text{Monad} \ m, \ \text{Interpreter} \ m) \Rightarrow \ \text{CExp} \rightarrow \ \text{InterpM} \ m \ \text{CExp}
\]

The Interpreter typeclass essentially says that we have to be able to eval things inside of it (\text{evalI}), print output somehow (\text{putStrI}), generate random numbers (\text{randomI}), and read files (\text{readFileI}).

\[
\text{class \ Interpreter \ m \ where}
\text{evalI} :: \ \text{CExp} \rightarrow \ \text{InterpM} \ m \ \text{CExp}
\text{putStrI} :: \ \text{T.Text} \rightarrow \ \text{InterpM} \ m \ ()
\text{randomI} :: \ \text{Int} \rightarrow \ \text{InterpM} \ m \ \text{Int}
\text{readFileI} :: \ \text{String} \rightarrow \ \text{InterpM} \ m \ \text{String}
\]

In all of the instances so far created, \text{evalI} is always set to \text{evalM} though that doesn’t need to be the case. For \text{evalSafe} we use a state monad keeping track of a virtual world-state as the base monad, using a given seed for pure random number generation and just returning NIL for \text{readFileI} calls.

\[\text{InterpM monad (transformer)}\]

The InterpM monad gives us two big pieces of functionality: keeping track of the state of the list of symbolMaps (using the \text{StateT} monad transformer) and being able to cut the computation short when there’s an error (using the \text{EitherT} monad transformer).

\[
\text{newtype \ InterpMm \ a = \ InterpM} \{\ \text{unInterp} :: \ \text{S.StateT} ([\text{SymbolMap} \ m]) (\text{E.EitherT} [\text{Error}] \ m) \ a \}
\]

The use of a newtype for InterpM instead of a type synonym was first due to the copying of the forms seen at https://en.wikibooks.org/wiki/Haskell/Monad_transformers, but we eventually learned that this is so that we have complete control over people using the monad correctly (not exporting the \text{unInterp} constructor for example), and being able to derive all our own instances.

InterpM is a monad transformer in a sense because it will take any base monad, though we don’t derive MonadTrans class (allowing lift for example) because there’s no need to use InterpM at an intermediary stage in any other monad transformer stack.

\[\text{Changes from original plan}\]

Contrary to our expectation of quickly dropping features out of our plan, we added quite a few features to the language. In hindsight, our original project idea seems simple. However, it is hard to judge a project difficulty in a language you have not used before. Haskell was a blessing and a curse because it allowed us to implement many more features than we would
have dreamed possible at the beginning of the quarter. But it also gives you many sleepless
nights trying to mash types together until the compiler stops complaining.

Our original intent, was to have a bare bones web server and an extremly featureless
Lisp implementation with almost no functionality other than printing strings. Haskell had other
plans for us. Getting a working web server took us under an hour. We had alotted much time
to get a webserver up and running. When it 'just worked' we had no idea what to do with all
that extra time. From here the complexity of the project snowballed, more features of Lisp
were added than originally intended.

Also worth note is that nothing from our plan was dropped. We managed to implement
everything and then some. We believe Haskell is soley responsible for this. With all the extra
time we added a compile to javascript feature and lots of minor modifications to the language,
and had much more time to check things over and make sure they worked properly.

Conclusion

SPB gave us a powerful way to play with the parsec library and the functionality of
Haskell itself. Haskell makes many complicated things either simple, or easy to implement by
mashing types together until it works. Interstingly, we could probably implement a similar DSL
and webserver in C, but it would require almost an exponential increase in the amount of time
required to get it all working. And most likely would not include functional features like currying
or lambdas. In fact if we had to take one lesson away from this project, it is that lambdas are 
magical creatures that buy you all sorts of other properties.