MP 4 — Scheme

CS 421

Revision 1.4.1

Assigned April 1, 2016
Due April 11, 2016

1 Changelog

Version 1.3.1 Added "`!`" to symbols in `identFirst`

Version 1.3.2 Better description of (*f args*) form in requirements

Version 1.3.3 Updated due date

Version 1.3.4 Added information about how to show macros

Version 1.3.5 Added explicit symbols for `whitespace` to handle

Version 1.3.6 Added changelog to pdf

Version 1.3.7 Changelog moved to top, changes boxed

Version 1.4.0 Major version

- Fixed `ConsVal` type in requirements
- Added clause about different types for lifter functions being ok
- Removed "Splicing" from section header

Version 1.4.1 Should check for `SymExp "define"` before evaluating first argument of `SExp`

2 Objectives and Background

The objective for this MP is to build an interpreter for a small Lisp-like language called Scheme. You will use the Parsec parser we went over in class to build a parser, an evaluator to interpret the code, and a printer to show the output. Together these will form your repl.[1].

This language will have the normal things you would expect in a Lisp-like language, such as functions, integers, and lists. You will also write a macro system and explore how to use it. Macros give you the ability to program your programming language, redefining it to be anything you want.

3 Getting Started

Update your repository with `git pull` to find a directory called `mp4-scheme`. Inside are the usual Haskell-stack directories. The file you will be most interested in is `app/Main.hs`.

This file has the types you will need and enough parser to parse a single integer.

To run this, use `stack ghci` or `stack repl` (they are equivalent). If stack says "Compiler version mismatched...", follow the instructions and run `stack setup`.

Immediately you should be able to run the following example:

```
% stack ghci
*Main> repl H.empty
scheme> 435
435
scheme>
```

The `scheme>` string will be the prompt.

[1] read-eval-print loop

Pro Tip! Write the type of each function you are writing *before* you write the function. This assists you two-fold. First, it forces you to think about the function a bit before you start writing it and gives you a clearer overall view of the program. Second, it assists `ghc` during compilation; when you don't write the types explicitly `ghc` tries to infer the most general type possible, which can lead to difficult-to-debug situations (especially when using large types like the `ParsecT` type).

4 Problems

4.1 Basic Functionality

This first section of problems walks you through the process of adding functionality to your language. The second section explains how to add macros to your language.

4.1.1 Identifiers

First you will add identifiers to your parser.

Parsing Identifiers represent symbols and variable names. The first character of an identifier must be one of the set "`-*+/:'?><=!`", plus the upper and lower-case letters. The remaining characters can be from this set combined with the digits.

Define a parser for the first character of the identifiers (defined in the paragraph above), and let's call it `identFirst`. Also define one for the following characters of the identifiers, `identRest`. To parse an entire identifier, we can run the `identFirst` parser followed by `many` of `identRest`. Make a composite parser `identifier` which parses an entire identifier.[2]

It's a good idea to consume any whitespace that occurs after each atom (characters " ", "`\t`", and "`\n`"). Make a `whitespace` parser and call it at the end of `identifier`. Make sure it handles any amount of whitespace (including none!).

Representation Identifiers are stored in the `SymExp` constructor, and we will call them *symbols*. In Scheme, a symbol serves two purposes. As in most languages, it can represent a variable. The evaluator will have an `env` parameter that allows us to look up variables' values. A symbol can also be a value in its own right. We're not doing that just yet, but to cover this we have a `Val` constructor `SymVal`. If you happen to need to print a `SymVal` s, it will just print the s (remember to add this to the `Show` instance for `Val`).

Make a *grammatical* parser `aSym` which uses the *lexical* parser `identifier` to parse a symbol and return it as a `SymExp`. Make sure to extend `anExp` with the option to parse `aSym` (right now `anExp` can only handle `anInt`, which we have provided).

When the evaluator (`eval`) comes across a `SymExp`, it should perform an environment lookup. If you look up a variable that does not have a value, return an `ExnVal` to signal an exception to the repl. Remember to modify the `Show` instance of `Val` to handle the new data-constructor `ExnVal`.

If you've implemented this correctly, you should be able to run the example below. Notice that we pass an explicit initial environment with a single mapping $x \mapsto 5$ to the `repl` command.

```
*Main> repl $ H.fromList [("x", IntVal 5)]
scheme> x
5
scheme> y
*** Scheme-Exception: Symbol y has no value. ***
scheme>
```

4.1.2 Function Calls

In Lisp, a *form* starts with a parenthesis and a name (an identifier), then some arguments, and then a closing parenthesis. Forms are used for everything in this language. If the initial symbol in a form is not a reserved word, it is taken to be a function or primitive operation. For the rest of this MP, when we say "form", we are specifically referring to an s-expression with a specific symbol in the first position.

Parsing To get started, update your parser to handle forms. You will need to read an opening parenthesis, then a list of expressions, then a closing parenthesis. Internally, this structure is called an *s-expression*. We will use an `SExp` data-constructor (of type `Exp`) to store them. Remember to consume any extra whitespace which may occur inside a form!

If you've implemented this correctly, you should be able to run the following code (notice we are testing the parser explicitly; we are *not* inside the repl here).

[2] You do not need to name the parsers the same things we have except for the top-level parser `anExp`.

```
*Main> parse aForm "form test" "(f 10 30 x)"
Right (SExp [SymExp "f",IntExp 10,IntExp 30,SymExp "x"])
*Main>
```

Implement `liftIntOp` **UPDATE** We do not need your `liftIntOp`, `liftBoolOp`, or `liftIntBoolOp` to have the same type/implementation as ours. Please implement them however you see is best - indeed the original way for `liftBoolOp` does not work as well as it could. Make sure that the primitive operators are in your variable `runtime` as `PrimVal`s with the correct type and when called give the correct behaviour. Note that the shown examples of testing `liftIntOp` directly will not work if you change the types.

Your `eval` function will check the first element of an incoming `SExp`. If it turns out to be some kind of function or primitive, the rest of the elements of the `SExp` are considered to be arguments to the function (or operator).

For example, `(+ 10 20)` should evaluate to `30` and `(- 10 5 2)` should evaluate to `3`. Operators in Scheme take a variable number of arguments!

To add support for primitive operators in your language, add a `Val` data-constructor `PrimVal :: ([Val] -> Val) -> Val` to store operators as values. Use `liftIntOp` to convert a Haskell operator into a suitable `PrimVal`. Note: to get subtraction (and other operators) working, use `foldl1` if there are elements in the list, and return the base case if there are no arguments.

We have provided some translators to go between Haskell values and Scheme values. They are `liftbool`, `lowerbool`, `liftint`, and `lowerint`. These can help when defining the various operator lifters.

If you have implemented `liftIntOp` correctly, you should be able to run the following code snippet. Note that we have to create an auxiliary function `testPrimVal` which will actually extract the lifted operator from the `PrimVal` for use over other values.

```
*Main> let testPrimVal = \(PrimVal f) vs -> f vs
*Main> testPrimVal (liftIntOp (+) 0) [IntVal 3, IntVal 5, IntVal 7, IntVal 9]
24
*Main>
```

Update the runtime environment The constant `runtime` is the initial runtime environment for the repl; it is a map from `String` (identifiers) to `Val` (values). This will be used to hold the values of defined constants, operators, and functions. You will call `repl` with this `runtime` when running it.[3]

You need to initialize `runtime` with predefined primitive operators as well. This will make these operators available to users of your language. Start by adding the key-value pair `(+, V)`, where `V` is the `PrimVal` associated with the operator `(+)` (above you created a function which can lift a Haskell operator into a `PrimVal`). We have supplied the tuple you will add to `runtime` for `"+"`.

Update `eval` Our `eval` function will only run into a `PrimVal` upon evaluating the first element of an s-expression, and this will correspond to applying that operator to the rest of the elements of the s-expression. Modify `eval` to handle s-expressions (specifically forms with primitive operators). You need to check if the first element of the s-expression is a primitive operator, and if so you should run the primitive operator over the rest of the elements of the s-expression. The final result should be a `Val` of some kind (as is always the case with `eval`).

Remember that the function that a `PrimVal` stores is of type `[Val] -> Val`, so you cannot just pass the rest of the expressions in the s-expression in unevaluated.

If you did all of this right, you should now be able to do this:

```
*Main> repl runtime
scheme> (+ 3 4 2 10)
19
scheme> (+ 3 3 3)
9
scheme>
```

Go ahead and add subtraction and multiplication. To do this you only need to change `runtime`. Notice how easy it is now to extend our language with any primitive operators we want by injecting them into `runtime`. Since we've defined `eval` recursively, you already have quite a bit of power:

```
*Main> repl runtime
scheme> (+ 2 3)
```

[3] You can call `repl` with any initial environment. Look at Section 4.1.1 to see an example where `repl` is called with a custom environment. `runtime` will just be a *default* initial environment to use.

```
5
scheme> (* 2 3)
6
scheme> (+ 2 (* 3 4))
14
scheme> (- 20 1)
19
scheme> (- 10 5 2)
3
scheme>
```

When primitives are called on their own, print out the string `*primitive*` by adding to the `Show` instance for `PrimVal`.

```
*Main> repl runtime
scheme> +
*primitive*
scheme>
```

The empty s-expression `()` should be evaluated to `SymVal "nil"`.

```
*Main> repl runtime
scheme> ()
nil
scheme>
```

4.1.3 User-defined functions

Now it is time to allow users to define their own functions in your language. In Scheme, a function definition takes the form `(define` *f* `(`*parameters*`)` *exp*`)`. There can be as many or as few parameters as you want.

To implement this, we have two new constructors. The function value will be stored in the `Val` constructor `Closure`. It has the usual three arguments: a list of strings (for the arguments), an expression (for the body of the function), and an environment.

You also need a value constructor called `DefVal`. It will have a string argument for the variable being defined and a value for what it's being defined as. We use `DefVal` to signal the `repl` that a new variable has been defined. If `repl` gets a `DefVal` back from `eval`, it will simply print the name of the variable being defined, and make a recursive call with the updated environment.

Now you must modify `eval` to handle the special case where the first element of the s-expression you are evaluating happens to be `SymExp "define"`. Once you've identified that the s-expression is of the `define` form in `eval`, you must build the correct `DefVal` which holds the name of the defined function and a `Closure` corresponding to the definition.

```
*Main> repl runtime
scheme> (define inc (x) (+ x 1))
inc
```

While you are at it, make another special form `def` which defines a variable that is not meant to be a function (a constant).

```
scheme> (def x (+ 10 20))
x
scheme> x
30
scheme> y
*** Scheme-Exception: Symbol y has no value.
scheme>
```

Of course, we haven't implemented function calls yet. Let's do that now. If the first element of an s-expression turns into a `Closure` upon evaluation, then the rest of the elements become arguments to the closure. These arguments must be evaluated in the current environment. Then call `eval` again, binding (*in the closure's environment*) the closure's parameter names to the evaluated arguments.

If you did it right, you should now be able to do this:

```
*Main> repl runtime
scheme> (def x 1)
x
scheme> (define inc (y) (+ y x))
inc
scheme> (inc 10)
11
scheme> (def x 2)
x
scheme> (inc 10)
11
scheme> (define add (x y) (+ x y))
add
scheme> (add 3 4)
7
scheme>
```

As long as we're thinking about functions, we should add the `lambda` form. It will return a closure, which should not be printed directly by the evaluator but instead should display `*closure*`.

```
*Main> repl runtime
scheme> (lambda (x) (+ x 10))
*closure*
scheme> ( (lambda (x) (+ x 10)) 20)
30
scheme> (define mkInc (x) (lambda (y) (+ x y)))
mkInc
scheme> (def i2 (mkInc 2))
i2
scheme> (i2 10)
12
```

This, by the way, is a major difference between Scheme and Lisp. In Lisp, we would have had to write `(funcall i2 10)` since functions have a separate name-space than variables. If you get bored and want to start a fight, go to a Lisp or Scheme discussion forum and tell them they should be doing it the other way. Log in as someone else first.

4.1.4 Quoting Symbols

The quote operator tells Scheme to convert the next expression to a value, as a symbol or a list. You can quote anything in Scheme, but for now we'll just focus on variables and integers (the only other thing being s-expressions).

There are two ways to quote something in Scheme, and you should supply both. The long way is the special form `quote`. The shortcut way is the `'` operator. It is often used as a shortcut for `(list ...)` (which we will define soon) but it quotes all the arguments before they are evaluated.

Add a parser `aQuote` to handle the `'` operator; any expression `'e` should be parsed as `SExp [SymExp "quote", e]`. Remember that you'll need to change your `anExp` parser to handle `aQuote` as well, and that *the order that you try parsers in `anExp` matters.*

Now modify `eval` to handle the symbol expression. Quoting a `SymExp` returns the corresponding `SymVal` without looking it up. Quoting an integer expression returns the integer value. Don't try double-quoting anything just yet.

Once you have that, you should be able to run your repl like this.

```
*Main> repl runtime
scheme> 'a
a
scheme> '5
5
scheme> (quote a)
a
scheme> 'a
a
scheme> 'asdf
asdf
scheme> '*first-val*
*first-val*
```

4.1.5 Cond

The form for conditions looks like this: `(cond (` c_1 e_1 c_2 e_2 $\cdots$`))`

The cond form will evaluate each of the c_i in turn until it finds one that is true, and then evaluate the corresponding e_i. If none of them turn out to be true, then the form will return the symbol `nil`. Also return `nil` if a particular c_i is true, but is missing a matching e_i.

Of course, we will need to add support for booleans and boolean expressions. In many dialects of Lisp, `false` is indicated by the symbol `nil`, `true` is specifically indicated by the symbol `t`, and anything that is not `nil` is counted as `true` also. That's what we will use in our language.[4]

To prepare for `cond`, write the other two function lifters, `liftIntBoolOp` (takes integers and returns a boolean), and `liftBoolOp` (takes booleans and returns a boolean). Add the primitive integer comparison operations `">"`, `"<"`, `">="`, `"<="`, `"="`, and `"!="`. Also add the primitive boolean operators `"and"`, `"or"`, and `"not"`.

You should also add the predicate `"eq?"` which will tell if a list of expressions all are the same expression. This should handle both the `IntVal` and `SymVal` data-constructors. Remember that the lifted operator `PrimVal` *must* handle lists of arguments. In the case of `"not"`, which is our only unary primitive, you should return an exception if the user does not supply exactly one argument. Note that you will not be able to use your lifting operators to defined `"not"` and `"eq?"` because they don't behave the same as the other operators.

You can test these lifted functions in the same manual way we tested `liftIntOp` in section 4.1.2. Once you are convinced you are lifting operators correctly, and you have added the corresponding primitives to your `runtime`, you should be able to do the following:

```
*Main> repl runtime
scheme> (> 5 3)
t
scheme> (> 6  4 2)
t
scheme> (> 6 4 2 6)
nil
scheme> (and 't 't)
t
scheme> (and 't 'nil)
nil
scheme> (and 't 't 't 't)
t
scheme> (and 't 'nil 6 4)
nil
scheme> (and 't 5)
t
scheme> (and (> 4 2) (> 5 2))
t
scheme> (and (> 4 2) (> 2 5))
nil
scheme> (not (> 5 3))
nil
scheme> (not (< 5 3))
t
scheme> (not 't 't)
*** Scheme-Exception: `not` is a unary operater. ***
scheme> (not 'nil 't 't 't)
*** Scheme-Exception: `not` is a unary operator. ***
scheme> (eq? 'a 'b)
nil
scheme> (eq? 'a 'a)
t
scheme> (eq? 'a 5)
nil
scheme> (eq? 5 5)
t
```

[4]Warning to C/C++ users: this includes 0! You can always tell when a C programmer learns Lisp: they walk around in a daze saying "nothing is true!"

```
scheme> (def x 5)
x
scheme> (eq? x 5)
t
scheme> (eq? x x)
t
scheme> (= 3 2)
nil
scheme> (= 3 3)
t
scheme>
```

Now that we have some proper booleans, you can write the `cond` form. Here are some examples of what it will look like when you are done.

```
*Main> repl runtime
scheme> (cond ((> 4 3) 'a (> 4 2) 'b))
a
scheme> (cond ((< 4 3) 'a (> 4 2) 'b))
b
scheme> (cond ((< 4 3) 'a (< 4 2) 'b))
nil
scheme>
```

You can write functions now that test for things, such as `max` or `min`, but recursion won't work yet.

```
scheme> (define fact (n) (cond ((< n 1) 1 't (* n (fact (- n 1))))))
fact
scheme> (fact 5)
*** Exception: Cannot lower, not an IntVal!
*Main>
```

To fix recursion, you will have to make the closures returned by `define` have the definition for themselves present in the closure environment. Modify your definition of `define` to add this and make recursion work.

```
*Main> repl runtime
scheme> (define fact (n) (cond ((< n 1) 1 't (* n (fact (- n 1))))))
fact
scheme> (fact 5)
120
```

4.1.6 Let

The `let` form is `(let ((`v_1 e_1`)` $\cdots$ `(`v_n e_n`)) `e`)`.

Each v_i is assigned value of e_i evaluated, and the body e evaluated in the augmented environment. The variables' definitions are discarded after the entire `let` has been evaluated.

```
scheme> (let ((x 5) (y 10)) (+ x y))
15
scheme> (def x 20)
x
scheme> (def y 30)
y
scheme> (let ((x 11) (y 4)) (- (* x y) 2))
42
scheme> x
20
scheme> y
30
```

Note that the e_i do *not* have access to the new definitions. This is called "simultaneous assignment". Alternatively, the form `letrec` allows each e_i to access the new definitions (i.e. to mention any v_j also defined in the `let` clause).

4.1.7 Lists

Lisp and Scheme are all about lists. It's time we added them.

A list is stored in a data structure called a *cons cell*. A `cons` cell has two parts. The first part is called the *car*, and the second part is called the *cdr*. The names are historical.

You can use a `cons` cell to store a pair or a list. To store a pair, just use the `cons` function to put them together. A pair is printed with a period between the two elements.

```
scheme> (cons 2 3)
(2 . 3)
```

We will not implement dotted notation in our parser.

A list is a `cons` cell with another `cons` cell in the `cdr` position. In such a case, we do not print the dot separator. The symbol `nil` represents the empty list.

Be careful implementing the `show` instance for `ConsVal`. you may need to make a helper function which the `show` function calls to handle printing the list properly. In addition, you may want to write functions `liftlist` and `lowerlist` which can be used to convert between Haskell lists and Scheme lists (these are useful later too!)

All told, you should get the same output as shown below:

```
scheme> (cons 2 (cons 3 4))
(2 3 . 4)
scheme> (cons 2 (cons 3 (cons 4 'nil)))
(2 3 4 )
```

Once you have that, write the `car` and `cdr` primitives so you can extract the components of a list. You should also write the `list` primitive, which returns a list of the arguments. You may find the helpers `liftlist` and `lowerlist` mentioned above helpful here as well.

Make sure that if `car` or `cdr` is called on something that doesn't evaluate to a `ConsVal`, you raise an `ExnVal`.

```
*Main> repl runtime
scheme> (list (> 3 4) 't 15 'nil (< 5 2 3 5) (cons 3 (cons 4 3)))
(nil t 15 nil nil (3 4 . 3) )
scheme> (car (cons 'a 'b))
a
scheme> (cdr (cons 'a 'b))
b
scheme> (car (list 'a 'b 'c))
a
scheme> (cdr (list 'a 'b 'c))
(b c )
scheme> (cdr (list 'a))
nil
scheme> (cdr 'a)
*** Scheme-Exception: Not a cons cell: a ***
scheme> cdr
*primitive*
scheme>
```

4.1.8 Quoting Lists

Now that we have lists, we can talk about quoting arbitrary expressions. To quote an `SExp`, quote each of the elements of the `SExp`, then combine the results as a `ConsVal` list. Note that you may find it useful here to make an explicit `quote` function to help with this process.

These examples should illustrate the interactions between `quote` and other forms. Remember that a symbol is printed without the initial quote.

```
scheme> 'a
a
scheme> ''a
(quote a )
scheme> (car (quote (a b c)))
a
```

```
scheme> (car '(a b c))
a
scheme> (car ''(a b c))
quote
scheme> '(2 3 4)
(2 3 4 )
scheme> (list (+ 2 3))
(5 )
scheme> '( (+ 2 3))
((+ 2 3 ) )
scheme> '(+ 2 3)
(+ 2 3 )
```

4.1.9 eval

You have enough power now to build Scheme expressions from inside the language itself! Now add a function that takes a quoted Scheme expression, unquotes it (i.e. converts it from a value back into an expression), and evaluates it. Here (and later) you may find it useful to write an `unquote` helper function.

```
scheme> '(+ 1 2)
(+ 1 2 )
scheme> (eval '(+ 1 2))
3
scheme> (eval ''(+ 1 2))
(+ 1 2 )
scheme> (eval (eval ''(+ 1 2)))
3
scheme> (def a '(+ x 1))
a
scheme> (def x 5)
x
scheme> (eval a)
6
```

4.1.10 Quasiquote and Comma

These next three operators will allow you to write Scheme programs that write Scheme programs more efficiently.

The quasi-quote operator ` is like quote. It quotes the argument, but any expression preceded by a comma (or the special form `unquote`) gets evaluated instead of quoted.

Use a helper function `quasiquote` to handle this. (You get to supply its definition.) Note that it will need to be able to call `eval`.

```
scheme> (def a 5)
a
scheme> `(+ a 1)
(+ a 1 )
scheme> `(+ ,a 1)
(+ 5 1 )
```

You may handle multiple levels of quasi-quoting if you want, but handling just one is sufficient for this MP. You need to keep track of how many levels of quasi-quoting you have seen to know whether to output a symbol for unquote or to evaluate the argument. It is an error to have an unquote without an enclosing quasi-quote. Remember that your parser needs to handle quasi-quotes and unquotes now too!

This example has been updated slightly.

```
scheme> (def a 5)
a
scheme> ``(+ ,,a 1)
(quasiquote (+ (unquote 5 ) 1 ) )
scheme> ``(+ ,,a ,a)
(quasiquote (+ (unquote 5 ) (unquote a ) ) )
```

```
scheme> `(+ a ,,a)
(+ a *** Scheme-Exception: Cannot `unquote` more than `quasiquote`. *** )
scheme> ``(+ a ,,a)
(quasiquote (+ a (unquote 5 ) ) )
scheme> (eval ``(+ ,,a 1))
(+ 5 1 )
scheme> (eval (eval ``(+ ,,a 1)))
6
scheme>
```

4.2 Macros

I hope you've had fun so far. Many people, seeing Lisp for the first time, are put off by the parentheses. But they are actually the source of power for this language. Code and data have the same form in this language, a feature called *homoiconicity*[5]. It is easy to write programs in Lisp that write other programs in Lisp.

It is considered bad form to use calls to `eval` directly, but there is a more disciplined way. It's called a *macro*.

It is another special form, the last we will add using Haskell. It looks just like a `define` except it uses the symbol `defmacro` instead.[6]

The form for defining a macro is `(defmacro` *f* `(`*parameters*`)` *body*`)`. Macro definitions are handled exactly the same as function definitions, only we store it in the `Macro` data-constructor instead of the `Closure` one so that we know it's a macro and not a function when we go to use it. When asked to display a macro, show the string `*macro*`.

There are two differences when actually using a macro (calling it on arguments). First, we will *not* evaluate the arguments to the macro, instead we will quote them. Second, once the *body* expression is evaluated (in the macro environment augmented with the maps from parameter names to the quoted arguments), it's assumed that the result will be a quoted Scheme expression. You need to `unquote` it and feed it into the evaluator one more time, this time in the original environment.

By doing this, we effectively allow "higher-order" or "reflective" programming. The arguments we pass to a macro can be expressions themselves, but we quote them ahead of time so that they are not evaluated before the macro does its work. The macro will place those expressions for us (possibly even changing them first!), then subsequently call `unquote` on everything to free them up for evaluation. Then the final evaluation happens and the expressions we passed in as arguments to the macro are now free (unquoted) to reduce to their actual values in the original environment. In this way, a programmer can, using the Scheme language directly, manipulate Scheme programs without any extra machinery. This is really quite powerful.

Here are some things you can do with macros. Are you sad that there's no `if` construct? No problem! Want a function that changes pessimistic expressions into optimistic ones? Macros to the rescue!

```
scheme> (defmacro if (con then else) `(cond (,con ,then 't ,else)))
if
scheme> (def a 5)
a
scheme> (if (> a 2) 10 20)
10
scheme> (if (< a 2) 10 20)
20
scheme> (define fact (n) (if (< n 1) 1 (* n (fact (- n 1)))))
fact
scheme> (fact 10)
3628800
scheme> (defmacro mkplus (e) (if (eq? (car e) '-) (cons '+ (cdr e)) e))
mkplus
scheme> (mkplus (- 5 4))
9
```

Most dialects of Lisp that are used in real life have a large part of their standard libraries written using macros.

Try writing a macro called `mk-inc`. It will create incrementers for you. Give it the name of the incrementer you want and how much it will increment. (You don't have to turn this part in. It's just to help you test your code and have fun.)

```
scheme> (mkinc foo 10)
foo
scheme> (foo 1)
11
```

[5] Where *homo* means "same" and *iconic* means "form."

[6] In some Scheme versions they use the symbol `defsyntax`.

5 Where to go from here

You are done with the MP. But there are many things you could still do to improve the language. Here's some ideas if you want to continue playing, and certainly you can add your own.

- The parser only accepts one line of input at a time.
- Add strings.
- Add the ability to execute multiple expressions.
- Add file I/O.

Another interesting exercise is to write `eval` in Scheme itself. This creates a *meta-circular interpreter*.

6 One last thing...

Be sure to add, commit, and push your work to turn it in. In fact, you should do this frequently so you have a backup in case your computer gets dropped into a volcano or something.

Be sure your final submission compiles. **Code that does not compile will result in a zero!** If you cannot get a specific function to work (say `liftIntBoolOp`), but a lot of the functionality works without it, remember that you can define it as `undefined` to make it compile. You will have to provide the type-signature of the function for Haskell to know how to compile it.

6.1 Checklist

For your grade, make sure that you have taken care of the following:

Parsing - 25 points Your solution must be able to parse all three types of expressions. We have already handled integers for you, but you must finish symbols and s-expressions. All of the following data-types should be producible by your parser:

- `IntExp :: Integer -> Exp`
- `SymExp :: String -> Exp`
- `SExp :: [Exp] -> Exp`

You should at least have parsers for integers, symbols, forms, quotes, quasi-quotes, and unquotes.

Environment - 20 points Your solution must provide the default `runtime` environment with appropriate definitions of the primitive operators shown above. They are listed here again.

- `"+"`: Integer addition
- `"-"`: Integer subtraction
- `"*"`: Integer multiplication
- `">"`: Integer greater than
- `"<"`: Integer less than
- `">="`: Integer greater than or equal
- `"<="`: Integer less than or equal
- `"="`: Integer equal
- `"!="`: Integer not equal
- `"and"`: Boolean and
- `"or"`: Boolean or
- `"not"`: Boolean not (only operates on first element)

- `"eq?"`: Integer and symbol equality
- `"list"`: Construct a list from arguments
- `"car"`: Extract first element of a cons cell
- `"cdr"`: Extract second element of a cons cell

Evaluation - 35 points In addition to integers and symbols, you must be able to evaluate the following forms (which are represented as s-expressions):

- `(define` f `(`$args$`)` exp`)`: Function definition
- `(def` x exp`)`: Constant definition
- `(lambda (`$args$`)` exp`)`: Anonymous function definition
- `(quote` exp`)`: Quote an expression
- `(cond (`c_1 e_1 $\cdots$ c_n e_n`))`: Conditional form
- `(let ((`v_1 e_1`)` $\cdots$ `(`v_n e_n`))` exp`)`: let form
- `(cons` car cdr`)`: cons form
- `(eval` exp`)`: unquote then evaluate exp
- `(quasiquote` exp`)`: Quasiquote an expression
- `(defmacro` exp`)`: Macro definition
- `(`f $args$`)`: Application form. f could be a `PrimVal` for primitive operators, a `Closure` for user-defined functions, or a `Macro` for user-defined macros. Remember that f must be evaluated first to decide if it is a `PrimVal`, `Closure`, or `Macro` and the entire form handled accordingly.

Check that your `eval` function can somehow produce all of the following data-constructors:

- `IntVal :: Integer -> Val`
- `SymVal :: String -> Val`
- `ExnVal :: String -> Val`
- `PrimVal :: ([Val] -> Val) -> Val` (via a lookup)
- `Closure :: [String] -> Exp -> Env -> Val` (via lookup or `lambda`)
- `DefVal :: String -> Val -> Val`
- `ConsVal :: Val -> Val -> Val`
- `Macro :: [String] -> Exp -> Env -> Val` (via a lookup)

Printing - 5 points Your REPL must appropriately handle printing of each `Val`. Do this by defining the `Show` instance over all data-constructors of type `Val`. We have done `IntVal` for you.

REPL - 15 points Your REPL must be modified to correctly handle the special case where `eval` results in a `DefVal` by inserting the definition into the environment of future evaluations.