

# Project 1: Scheme

## A Simple Expression Evaluator

---

*Due: Friday, 25 October at 11:59 pm*  
*Weight: 10% of your course grade.*

For the Scheme project, you will write a simple mathematical expression evaluator, a bit like Maple, but much simpler of course. It will allow the user to evaluate arithmetic expressions and define functions which he or she can later use in other expressions.

The course web page has some code that you should start from.

All your Scheme code should use only pure functional programming—*i.e.*, you should not use **loop**, **do**, any procedure with a **!** in its name, or any other procedure with side effects. All procedures without side effects are allowed. However, you may use side effects for the following situations: you should use *display* to print information for the user, and you may use **set!** to save the latest bindings in a global variable for testing purposes only (this is already done for you in the starter code—look for *last-bindings* for details). This variable will be useful to restart your evaluator where it stopped the last time you ran it, but you are not allowed to use this variable in any other way in your code.

## Statements

---

Your evaluator will accept three types of statements from the user:

- Expressions to evaluate.  
Examples: 5, a, ((3 + (4 \* 5)) + (f x))
- Variable assignments.  
Examples: (a := 5), (b := (5 + (f 4)))
- Function definitions.  
Example: ((f x) := (x \* x))

The user will type one such statement at the prompt and then hit enter, at which point your system should handle it according to its type, or print some error message if it is invalid, and then present the user with a new prompt. Table 1 indicates how your system should handle each statement type.

## Syntax

---

To simplify the parsing stage, the user will be expected to type in the input in a form that is easily

Statement Type	System Response
Expression	Evaluate the expression and display the result.
Variable assignment	Evaluate the expression on the right hand side of := and bind it to the variable on the left-hand side, either creating a new binding for this variable or replacing an existing one.
Function definition	Create a Scheme procedure corresponding to the right-hand side expression, with the specified argument list, and bind it to the specified function name so that it can be called in later expressions.

Table 1: Handling of each statement type

recognized by Scheme, a Scheme list, as you can see in the examples above. Every expression will be fully parenthesized so that you don't have to handle precedence and associativity. Arithmetic expressions will be specified using infix notation. Function calls will use regular Scheme syntax, *i.e.*, in a list with the function name as the first element of the list. With this syntax, your expression evaluator does not need a parser, since Scheme will do all the parsing work for you, but it will need to explicitly handle infix arithmetic expressions and, of course, variable assignments and function definitions.

Note that with the specification of `resolve-bindings` (see below), Scheme builtin procedures will be available to the user without any extra work on your part. This means that you can start by making your system require prefix notation from the user, get it all working, and then add the handling of infix expressions. You'll get a lot more marks for a system which works with prefix expressions, but not infix ones, than for a broken evaluator that deals with infix expressions.

## Bindings

---

A binding is a pair (*var* . *value*), where *var* is a symbol and *value* is the value associated with it. This value can be a number (the result of a variable assignment) or a function (the result of a function definition).

A list of bindings is a list of such pairs. For example, `((a . 1) (b . 2) (f . #[compound-procedure 2]))` is a list of bindings that says *a* is bound to the value 1, *b* is bound to the value 2, and *f* is bound to a function. (The starter code shows and explains how a function can be directly inserted in the bindings using `quasiquote` and `unquote`.)

In your system, you will use lists of bindings to find the value of variables and functions that may appear in expressions to evaluate. You will also modify these bindings every time the user enters a variable assignment or a function definition. However, you will do so in a pure procedural way, that is, you will not destroy the current bindings, but rather create a new binding list with the new binding added.

### Your Task

Write the following functions to manipulate bindings and lists of bindings:

- (*get-binding symbol bindings*): Returns the most recently created binding for *symbol* if such a binding exists, () otherwise.

Example: (*get-binding 'a '((a . 1) (b . 2) (c . 3))*) returns (a . 1),

Preconditions: *bindings* is a binding list, *symbol* is a symbol.

- (*set-binding symbol value bindings*): Returns a new binding list with all the bindings in *bindings* plus *symbol* bound to *value*. Note that if there is already a binding for *symbol* in *bindings*, the new binding should take precedence, *i.e.*, a subsequent call to *get-binding* should return the new binding.

Preconditions: *bindings* is a binding list, *symbol* is a symbol.

- (*bound-var binding*): Returns the symbol bound by *binding*.

Precondition: *binding* is a binding.

- (*bound-value binding*): Returns the value of *binding*.

Precondition: *binding* is a binding.

- (*validate-bindings expr bindings*): Returns *#t* iff all the symbols in *expr* can be bound.

If any symbol appearing in *expr* cannot be bound, returns *#f* and uses (*display-error 'unbound symbol*) (see the starter code) to produce an error message for each undefined symbol telling the user that they used an undefined symbol.

Preconditions: *bindings* is a binding list.

Examples (assume *bindings* has been defined to ((a . 1) (b . 2) (f . #[compound-procedure 2]))):

- (*validate-bindings 'a bindings*) returns *#t*.
- (*validate-bindings '((f a) + b) bindings*) returns *#t*.
- (*validate-bindings '(+ (g c)) bindings*) prints:  
 Undefined variable: g  
 Undefined variable: c  
 (the order of the lines doesn't matter) and returns *#f*.

- (*resolve-bindings expr bindings*): Returns an expression which corresponds to *expr* but with all bindings resolved using *bindings*.

Preconditions: *bindings* is a binding list, and (*validate-bindings expr bindings*) holds.

Every time a symbol occurs in *expr*, it should be replaced as follows:

- if the symbol has a binding in *bindings*, replace it with the bound value;
- otherwise, if the symbol is bound to a built-in function (you should use *builtin?*, defined in the starter code, to test this), leave it unchanged;
- otherwise, use (*display-error 'internal*) to produce an error message telling the user that an internal error occurred and that they should contact the system vendor for a patch. :) (And leave the symbol unchanged.)

Examples (assume *bindings* has been defined to ((a . 1) (b . 2) (f . #[compound-procedure 2]))):

- (*resolve-bindings 'a bindings*) returns 1;
- (*resolve-bindings '((f a) + b) bindings*) returns ((#[compound-procedure 2] 1) + 2);
- (*resolve-bindings '(+ (g c)) bindings*) prints two error messages and returns (+ (g c)).

Note that *validate-bindings* and *resolve-bindings* should not care whether *expr* uses infix or prefix notation.

- (*infix-to-prefix expr*): Returns *expr* with any infix subexpression converted to prefix notation. The infix operators you must support are +, −, \*, /, ^, %, <, <=, =, >, and >=.

Note that the infix operators are all built-in in Scheme, except for ^ and %. The corresponding Scheme built-in procedures for these two operators are *expt* and *modulo*, respectively. When you write the expression evaluator (see below), you will need to consider carefully how and where you should handle them. There are many different ways to do it. Use a simple solution.

## The Expression Evaluator

---

As described above, each time the user enters a statement, your evaluator must read it, resolve its symbols (handling infix operators), evaluate it and print the result. If the statement is a *set* statement, the evaluator should create an appropriate new binding and make it available for use by the user in subsequent statements.

In the starter code, you will find a function called *run*, which takes one argument, *bindings*. In its current form, it reads an expression from the user, displays it and calls itself recursively. This will be the main function of your evaluator.

### Your Task

Modify *run* to handle the three kinds of statements that your evaluator should support: expressions, variable assignments and function definitions. Once the statement is handled, call *run* recursively with the original bindings if the statement was an expression, or with the new bindings in the two other cases.

When you handle an expression, make sure you verify that it contains no unbound variables, resolve all the variables, and handle infix expressions. You should end up with an expression that you can evaluate using *eval*. (Use () as the second argument to *eval*). Finally, you should display the result for the user. Warning: be careful to handle ^ and % correctly.

When you handle a variable assignment, *i.e.*, a statement of the form (*symbol := expr*), you should handle *expr* as above, without printing the result, bind the result to *symbol*, and display the symbol to tell the user it was bound.

When you handle a function definition, *i.e.*, a statement of the form

$$((fn\ parm_1\ parm_2\ \dots\ parm_n)\ :=\ body),$$

you should:

- Preprocess *body* as an expression to evaluate (see above), making sure that *parm<sub>1</sub>*, *parm<sub>2</sub>*, ... *parm<sub>n</sub>* are not changed. But don't actually evaluate the resulting expression!  
Hint: Instead of using the current bindings, use an enhanced list of bindings which also binds each formal parameter to itself.
- Build a lambda expression with (*parm<sub>1</sub> parm<sub>2</sub> ... parm<sub>n</sub>*) as its formal parameter list and *body*, with variables substituted and infix expressions converted to prefix notation, as its body.
- Evaluate that lambda expression using *eval*, with () as the environment.

- Bind the result to the symbol *fn*.
- Display  $(fn\ parm_1\ parm_2\ \dots\ parm_n)$  to tell the user that *fn* was defined.

Make sure that your code is robust—*i.e.*, if the user enters an invalid statement, your code should produce an error message using (*display-error* 'syntax), but it should not let Scheme produce an error message.

Exception: if the user calls a function with the wrong number or kind of arguments, or if the user tries to use something that is not a function as if it was, your code may crash.

Note that during your own testing, when Scheme does produce an error message, you can restart your session by calling (*run last-bindings*).

## Sample Run

---

Here is a sample run showing what your evaluator should print to the user. The output of your system should be identical, given the same user input, except maybe for the order of the two undefined variable messages following the input  $((g\ c) + 3)$ .

```
1 ]=> (load "a2")
;Loading "a2.scm" -- done
;Value: run

1 ]=> (define bindings '((a . 1) (b . 2) (f . ,(lambda (x) (* x x)))))
;Value: bindings.

1 ]=> bindings
;Value 1: ((a . 1) (b . 2) (f . #[compound-procedure 2]))

1 ]=> (run bindings)

>((f a) + b)
3
>(a := 5)
a
>((g c) + 3)
Undefined variable: g
Undefined variable: c

>((g x) := (x + 1))
(g x)
>((g c) + 3)
Undefined variable: c

>(g a)
6
>(:= a 3)
Invalid or unsupported expression.
```

```
>(a :=)
Invalid or unsupported expression.

>(a := 5 2)
Invalid or unsupported expression.

>quit
Done!
;No value
```

## Bonus 1: Efficient Design

---

Once you have your evaluator working, you may notice that with *validate-bindings*, *resolve-bindings*, and *infix-to-prefix*, you end up scanning the user's expressions 3 times or more. For a bonus of 5%, redesign your system so that your code traverses the user's expression only once to do everything. If you attempt this bonus question, do not change your main submission of this project. Instead, make a copy called `p1b1.scm` and implement the changes there. Also, write a short paragraph describing what changes you made to make your code efficient. Submit this paragraph in file `p1b1.txt`.

Note that although the result might be slightly more efficient code, this approach will likely yield code that is more difficult to read, understand, and debug. We propose this as a challenge, but we do not suggest that it is a good idea to design code this way. In fact, it is usually worth sacrificing some efficiency (as long as the big-O time complexity of your code isn't changed) for better readability and maintainability.

## Bonus 2: Arity Checking

---

As described above, your expression evaluator does not need to check the arity of the functions called by the user of your system, or if they are even functions. So if the user types  $(\cos 3 4)$ ,  $(-)$ , or  $(3 4)$ , your system is simply expected to crash. For a bonus of 5%, read up on the MIT-Scheme built-in procedures *procedure?* and *procedure-arity-valid?* and use it to validate the user's functions calls before they are executed by Scheme. Instead of crashing in those situations, produce an appropriate error message, using *(display-error 'arity function)* if *function* is called with an invalid number of arguments, or using *(display-error 'apply expr)* if *expr* is not a function but is used as if it was. (Note that you don't need to make this work with recursive procedure definitions—that would be a much harder task!) Again, do not make these changes in your main solution, but in a separate submission called `p1b2.scm`. And again, write a short paragraph describing what changes you made to your code to implement this, and submit this paragraph in `p1b2.txt`.

## Testing

---

The testing of your system must meet the quality standards described in Diane Horton's *Software*

*Testing* document, available in the Assignments section of the course web page.

## Unit Testing

To demonstrate that your code works, you must submit a unit test suite for each of *get-binding*, *validate-bindings*, and *infix-to-prefix*. (You should of course unit test all the procedures you write, but you only need to submit a unit test suite for those three.)

Each test suite should consist of an input file with a series of commands for Scheme to run, and an output file with the result. Call the input files `test-get-binding.in`, `test-validate-bindings.in`, and `test-infix-to-prefix.in`. The output files should have the same names but with a `.out` extension. Each input file must start with the command `(load "p1")` and continue with your test suite. Use `;` to insert comments between the test cases. These comments should briefly explain your testing strategy.

To run each test suite, use the command:

```
scheme < testfile.in > testfile.out
```

## System Testing

To demonstrate that your whole system works, you must submit a test suite showing how you tested your system as a whole. You should provide sufficient testing to show that your system handles everything it should, including correct input and syntax errors from the user. Note, however, that you do not need to demonstrate situations you are not required to handle, *e.g.*, calling a function with the wrong number of arguments.

Save your system test suite in file `test-system.in`, and its output in file `test-system.out`.

Run your system test suite using the same command as the unit test suites.

You only need to submit a system test suite for your main system, not for the bonus questions.

## Silent Policy

---

The silent policy will take effect 27 hours before this assignment is due. This means that no question asked after 9pm on Thursday, 24 October will be answered, whether it is asked on the newsgroup, by e-mail or in person.

## Handing It In

---

You must submit this entire project electronically.

- Place all your code in a single file called `p1.scm`, including the starter code.
- Place all your test input and output files in `.in` and `.out` files as specified.
- Place your bonus question submissions in `p1b1.scm`, `p1b1.txt` and/or `p1b2.scm`, `p1b2.txt`.
- Submit all your files using the following command:

```
submit -c csc324h -a Scheme filename(s)
```