Scheme Basics in Kawa

Much of this handout was leveraged off of some LISP handouts written by Todd Feldman and Nick Parlante.

Getting Started

We'll be relying on an open-source application called Kawa to write and test our Scheme code. Kawa, like virtually all Scheme environments, operates like a UNIX shell in that you type in commands, hit return, and expect something to happen in return. You launch the Kawa interpreter by typing **/usr/class/cs107/bin/kawa** at the command prompt. We've installed Kawa on the elaines, the pods, and the myths, so you'll need to work on one of those machines.

```
jerry> /usr/class/cs107/bin/kawa
#|kawa:1|#
```

You interact with the Kawa environment by typing something in and then allowing Kawa to read what you typed, evaluate it, and then print a result. Sometimes, what you type is so trivial that the result is precisely what you typed in.

```
jerry> /usr/class/cs107/bin/kawa
#|kawa:1|# 4
4
#|kawa:2|# 3.14159
3.14159
#|kawa:3|# "Ben Newman has big hair"
Ben Newman has big hair
#|kawa:4|# #f
#f
#|kawa:5|# #t
#t
```

Okay, you're not impressed yet, but there really is an intellectually compelling explanation for what just happened. You see, everything you type in has to evaluate to a single value. When the expressions we type in are constants, Kawa just echoes them right back at you. Constants (be they integer, floating point, string, or Boolean) evaluate to themselves.

Typically, you get the Scheme environment to do some work for you. By typing out more elaborate expressions involving a function call, Scheme evaluates the call and eventually prints whatever the expression evaluated to.

```
#|kawa:6|# (+ 1 2 3 4 5)
15
#|kawa:7|# (+ (* -40 (/ 9 5)) 32)
-40
```

```
#|kawa:8|# (* 1 2 3 4 5 6 7 8 9 10)
3628800
#|kawa:9|# (/ 10 3)
10/3
#|kawa:10|# (quotient 10 3)
3
#|kawa:11|# (remainder 10 3)
1
#|kawa:12|# (positive? 45)
#t
#|kawa:13|# (negative? 0)
#f
#|kawa:14|# (zero? -45)
#f
#|kawa:15|# (sqrt 2)
1.4142135623730951
#|kawa:16|# (expt 7 (expt 5 3))
433765494809799328253735475726318825169783299462040510174489301774456943272
099416808967219221175890932080 7
```

Each of the nine examples above invokes some built-in function. Not surprisingly, Scheme provides every mathematical operator imaginable. Any reasonable language needs to.

What **is** surprising is how we invoke a function in the first place. Function calls are expressed as lists, where function name and arguments are bundled in between open and close parentheses. All functions, including the binary ones like `+` and `/`, are invoked in prefix form. Each of the top-level items making up the function call gets evaluated, and once all sub-expressions have been evaluated, the top-level expression gets evaluated too.

When you've had it and want to do something else, you leave the environment by calling **`exit`**.

```
#|kawa:17|# (exit)
jerry>
```

Scheme Data Types

Scheme is actually a little more sophisticated than we're going to make it out to be. In fact, we're going to pretend—at least for a little while—that Scheme offers up only two types of data: **primitive**s and **list**s. Primitives are the types that can't be subdivided: integers, floating point numbers, rationals, characters, strings, and Booleans. Lists comprise the generic abstraction Scheme offers up for aggregating data. Scheme itself allows for arrays (called vectors), and Kawa takes aggregation even further by supporting records and classes. Initially, we're going to ignore everything else and pretend that lists are all we have for bundling multiple pieces of information. It's not at all an unreasonable thing to do, because Scheme programmers use the list more than all of the other aggregate types combined. (In fact, LISP is short for **Lis**t **P**rocessing, and Scheme and LISP are fundamentally the same language.)

What primitives are there? Pretty much the ones you'd expect, plus a few others:

Numbers: The set of all numeric constants includes:

> `42`, `2.818`, `11/5`, and `0.707-0.707i`.
>
> Numbers can be either integral (`42`), real (`2.818`), rational (`11/5`), or complex (`0.707-0.707i`). You shouldn't need to worry about which subtype you're dealing with, since Scheme just manages the automatic conversions between them. Scheme provides the full spectrum of mathematical functions: `+`, `-`, `*`, `/`, `quotient`, `remainder`, `modulo`, `sqrt`, `expt`, `exp`, `sin`, `atan`, and a good number of other ones. As expressions, numeric constants evaluate to themselves.

Booleans: Scheme has a strong data type for the Boolean, just like C++ and Java. The Scheme equivalents of true and false are `#t` and `#f`, although everything other than `#f` is understood to be logically equivalent to `#t`. Of course, Boolean expressions evaluate to Boolean values and influence what code is executed when and how many times. Scheme provides the full bag of logical operators for compound expressions: `and`, `or`, and `not`. As expressions, Boolean constants evaluate to themselves.

Characters: Examples of character constants are:

> `#\b`, `#\&`, `#\newline`, and `#\space`
>
> Not surprisingly, these are the textual atoms that make up strings. We don't have much reason to subdivide strings during our two-week crash course, so this is really just mentioned for completeness. Note that `#\t`

and `#\f` are character constants, but `#t` and `#f` are Boolean constants. As Scheme expressions, character constants evaluate to themselves.

Strings: String constants are linear sequences of zero or more characters, and are delimited by double quotes. As expressions, string constants evaluate to themselves. Examples of string constants include:

`"David Hall likes Scheme"`,
`"1234567890!@#$%^&*()"`,
`"Nancy Pelosi's \"San Francisco Values\""`, and
`""`.

Strings are bona fide data types with all forms of language support. Need to compare two strings? Use `string=?` (or `string<?`. Or `string>=?` Or `string-ci=?` if you're looking for case insensitive comparisons.) Want to concatenate 10 `"Not it! "`'s together? Just use `string-append`. Need to take the `"fun"` out of `"dysfunctional"`? Rely on `substring`.

It's not important to know all of the `string`-related functions just yet. Just understand that everything necessary to manipulate `string`s is either provided or easily implemented in terms of what is.

Symbols: Symbols are sequences of characters (without the double quotes) that serve as general purpose identifiers. Examples include:

`i`, `power-set-of-rest`, `partition`, `is-compatible?`, `fraction->real`,
`not`, `null?`, `list`, `string=?`, `+`, `remainder`, `and` `symbol->string`

The first row lists symbols that were invented for the purposes of the handout, but the second row lists just a few of the Scheme symbols for built-in Scheme functions. You have a lot more flexibility in how you spell out your symbols, in that you can use virtually any visible character you can type. Convention dictates that you stick to lower case letters and dashes for the most part. Symbols identifying predicate functions generally end in a question mark. Functions designed to convert one type to another typically include `->` right in the middle of their names.

Lists

A list is a sequence of zero or more Scheme expressions, where each sub-expression can be either a primitive or another list. In ASCII form, a list is represented by a left parenthesis `'('` followed by the ASCII forms of all the elements separated by spaces, followed by a right parenthesis `')'`. Here are a few examples:

```
(1 2 3)
(1 2.5 22/7)
("hi" (1 2 3 4) 42/5)
("hi" ((((1 2 3 (4))))) 42/7)
```

All four of these lists contain three elements, although only the first one is homogenous. Each of the sub-expressions can be pretty much anything you want, as long as each of the atoms are spelled out properly and all of those parentheses are balanced and properly nested.

Function Evaluation

Function calls are also expressed using list syntax. When you type in a list at the prompt and hit return, the expression evaluator reads it, and by default interprets it as a function call. It takes the first expression to be the function and assumes the remaining expressions are the arguments. The evaluator works by recursively evaluating the arguments, and then evaluating the function using the recursively generated arguments.

```
#|kawa:1|# (+ 1 10 100)
111
#|kawa:2|# (exp (- 3 2))
2.7182818284590455
#|kawa:3|# (sqrt (+ (* 5 5) (* 12 12)))
13.0
#|kawa:4|# (substring (string-append "candice" "bergen") 4 11)
iceberg
```

Since Scheme expressions are fully parenthesized, we don't bother with precedence rules. There's nothing to disambiguate.

Quote

Since the evaluator interprets lists as function calls and tries to evaluate them, we need a way to express a list *without* evaluating it. When a quoted expression is given to the evaluator, the expression is passed through intact. In particular, lists are passed through as lists instead of being interpreted as function calls. The quote is your way of telling the interpreter to take what you said verbatim.

```
#|kawa:5|# (+ 1 1/2 1/3 1/4 1/5 1/6 1/7 1/8 1/9 1/10)
7381/2520
#|kawa:6|# '(+ 1 1/2 1/3 1/4 1/5 1/6 1/7 1/8 1/9 1/10)
(+ 1 1/2 1/3 1/4 1/5 1/6 1/7 1/8 1/9 1/10)
#|kawa:7|# (1 2 3)
Invalid parameter, was: gnu.math.IntNum
java.lang.ClassCastException: gnu.math.IntNum
        at gnu.expr.ApplyExp.apply(ApplyExp.java:70)
        at gnu.expr.ModuleExp.evalModule(ModuleExp.java:156)
        at kawa.Shell.run(Shell.java:229)
        at kawa.Shell.run(Shell.java:172)
        at kawa.Shell.run(Shell.java:159)
        at kawa.repl.main(repl.java:744)
#|kawa:8|# '(1 2 3)
(1 2 3)
```

As harmless as expression 7 looks, it's way wrong. The symbol 1 occupies the position normally occupied by a function symbol. That 1 is evaluated, it evaluates to the number 1, and it's quickly determined that the number 1 isn't associated with any function definition. You get some hint into the implementation of the Kawa environment—it's implemented in Java and Scheme integers are backed by an **`IntNum`** class in the **`gnu.math`** package. But that's actually irrelevant, save for the fact that the function call was bogus, Kawa threw a fit in the form of an exception, and you got to see it. Notice that expression 8 gets it right—or at least it avoids the problems associated with expression 7.

Manipulating Lists

Since most everything in Scheme is expressed as a list, we need to know how to break them down into their constituent parts and build up new ones. We'll soon see that even function definitions are expressed as lists, so you need to be fluent in the core subset of built-ins that help you manipulate lists. Here's an overview of Lists 101.

cons, car, and cdr

`cons` is the basic list construction function. **`cons`** takes an element and a list and constructs a new list, one longer than the old one, which has the element pushed on the front of the list. Elements in lists may be of any type: numbers, strings, symbols, other lists, and so forth and need not all be of the same type (a list can contain any combination of numbers, symbols, other lists, etc.)

```
#|kawa:1|# (cons 1 '(2 3))
(1 2 3)
#|kawa:2|# (cons "Beyonce" '("Christina" "Whitney" "Kelly"))
(Beyonce Christina Whitney Kelly)
#|kawa:3|#   (cons 'eenie (cons 'meenie (cons 'miney (cons 'mo
'())))))
(eenie meenie miney mo)
#|kawa:4|# (cons '(a b) '(c (d (e))))
((a b) c (d (e)))
```

While **cons** builds lists out of elements, **car** and **cdr** pulls elements out of lists. **car** returns the first element of a list, and **cdr** returns the list without the **car**. The argument not only needs to be a list, but it needs to be a non-empty one, else you should expect more tantrums.

```
#|kawa:1|# (car '(1 2 3 4))
1
#|kawa:2|# (cdr '(1 2 3 4))
(2 3 4)
#|kawa:3|# (cons (car '(1 2 3 4)) (cdr '(1 2 3 4)))
(1 2 3 4)
#|kawa:4|# (car '((1 2) 3 4))
(1 2)
#|kawa:5|# (cdr '((1 2) 3 4))
(3 4)
#|kawa:6|# (car '(singleton))
singleton
#|kawa:7|# (cdr '(singleton))
()
#|kawa:8|# (car '())
Argument #1 (()) to 'car' has wrong type (gnu.lists.LList)
        at gnu.mapping.MethodProc.matchFailAsException(MethodProc.java:97)
        at gnu.mapping.Procedure.checkN(Procedure.java:380)
        at gnu.expr.ApplyExp.apply(ApplyExp.java:70)
        at gnu.expr.ModuleExp.evalModule(ModuleExp.java:156)
        at kawa.Shell.run(Shell.java:229)
        at kawa.Shell.run(Shell.java:172)
        at kawa.Shell.run(Shell.java:159)
        at kawa.repl.main(repl.java:744)
#|kawa:9|# (cdr '())
Argument #1 (()) to 'cdr' has wrong type (gnu.lists.LList)
        at gnu.mapping.MethodProc.matchFailAsException(MethodProc.java:97)
        at gnu.mapping.Procedure.checkN(Procedure.java:380)
        at gnu.expr.ApplyExp.apply(ApplyExp.java:70)
        at gnu.expr.ModuleExp.evalModule(ModuleExp.java:156)
        at kawa.Shell.run(Shell.java:229)
        at kawa.Shell.run(Shell.java:172)
        at kawa.Shell.run(Shell.java:159)
        at kawa.repl.main(repl.java:744)
#|kawa:9|# (cdr 'not-a-list)
Argument #1 (not-a-list) to 'cdr' has wrong type (java.lang.String)
        at gnu.mapping.MethodProc.matchFailAsException(MethodProc.java:97)
        at gnu.mapping.Procedure.checkN(Procedure.java:380)
        at gnu.expr.ApplyExp.apply(ApplyExp.java:70)
        at gnu.expr.ModuleExp.evalModule(ModuleExp.java:156)
        at kawa.Shell.run(Shell.java:229)
        at kawa.Shell.run(Shell.java:172)
        at kawa.Shell.run(Shell.java:159)
        at kawa.repl.main(repl.java:744)
```

By cascading calls to **car** and **cdr**, you can extract individual items and sublists. If you need the fourth element of a list, **cdr** three times, and take the **car**.

```
#|kawa:10|# (car (cdr (cdr (cdr '(1 2 3 4 5 6 7)))))
4
```

Other combinations lead to different elements, as illustrated here:

```
#|kawa:11|# (cdr (cdr (car (car '(((1 2 4 5) 6 (7 (8)) 9) 10 11
12)))))
(4 5)
#|kawa:12|# (cdr (car (cdr (car (("Ilya" ("Aman" "Tony" "Sarah")
"Austin"))))))
(Tony Sarah)
```

You'll be delighted to learn that up to four cascading **car** and **cdr** calls can be abbreviated with a single function. If you're pining for the **cdr** of the **cdr**, then you really want **cddr**. If you need to produce the **(Tony Sarah)** list above, then forego the **(cdr (car (cdr (car** approach and adopt **cdadar**.

```
#|kawa:13|# (cddr '(1 2 (3) 4))
((3) 4)
#|kawa:14|# (cdadar '(("Ilya" ("Aman" "Tony" "Sarah") "Genaro")))
(Tony Sarah)
```

The order of the a's and the d's within something like **cddar** or **cadadr** is taken as the order you'd otherwise have typed **car**s and **cdr**s out.

Alternate List Constructors

Sometimes the **cons** configuration of combining an element and a list is awkward. In that case the functions **list** and **append** may be what you need. **list** merges several elements into one big list while **append** merges several lists together into a single one. Like **cons**, both **append** and **list** allocate memory to construct their results.

list

Takes any number of values of any type and binds them all together in a new list. The number of arguments to list will be the number of elements in the new list.

```
#|kawa:1|# (list 1 3 5 (+ 4 3) '(/ 18 2))
(1 3 5 7 (/ 18 2))
#|kawa:2|# (list '() '() '())
(() () ())
```

append

Takes any number of **lists** and merges their elements together into a new list. The length of the resulting list will be the sum of the lengths of the given lists.

```
#|kawa:3|# (append '(1 2) '(3 4) '(5 6 7))
(1 2 3 4 5 6 7)
#|kawa:4|# (append '(1 2) '((1 2)) '((1) (2)))
(1 2 (1 2) (1) (2))
```

Boolean Functions

Scheme includes the usual Boolean operators as well as many built-in predicates which return a truth value to test some condition:

and Takes any number of expressions and returns false if and only if one or more of the arguments is false. As with most languages, Scheme promises to short circuit: the evaluation stops as soon as it finds a false expression.

```
#|kawa:1|# (and (< (expt 2 10) 1000) (string<? "mellow"
"fellow"))
#f
#|kawa:2|# (and (> (expt 2 10) 1000) (string<? "mellow"
"fellow"))
false
#|kawa:3|# (and (> (expt 2 10) 1000) (string>? "mellow"
"fellow"))
true
#|kawa:4|# (and)
#t
```

Don't ask why Kawa prints **#t** and **#f** in some cases and **true** and **false** in others, because I don't know the answer. Nothing ever bad happens because of it, though, so it just appears to be an inconsistency. I've confirmed that all four results work as Booleans where they need to.

or Takes any number of expressions and returns false if and only if every single one of its arguments evaluates to false as well. **or** also short-circuits and comes back with an answer as soon as if figures it out.

```
#|kawa:1|# (or (< (sqrt (* 2 2 2)) 3) (string>=? "james"
"bond"))
#t
#|kawa:2|# (or (< (sqrt (* 2 2 2)) 3) (string<=? "james"
"bond"))
#t
#|kawa:3|# (or (> (sqrt (* 2 2 2)) 3) (string<=? "james"
"bond"))
false
#|kawa:4|# (or)
#f
```

not Generates the logical inverse of its argument. Real shocker.

```
#|kawa:32|# (not (string=? "abcde" "abcde"))
#f
#|kawa:33|# (string=? "abcde" "abcde")
#t
#|kawa:34|# (not (string=? "abcde" "abcde"))
#f
```

```
#|kawa:35|# (not (not (string=? "abcde" "abcde")))
#t
#|kawa:36|# (not (or (> 3 4) (< 8 9)))
#f
```

list? Returns true if and only if its argument evaluates to a list.

```
#|kawa:1|# (list? '(1 2 3))
#t
#|kawa:2|# (list? '())
#t
#|kawa:3|# (list? '((1 2) "hello" ("gorgeous")))
#t
#|kawa:4|# (list? 11)
#f
#|kawa:5|# (list? 11/3)
#f
#|kawa:6|# (list? 4.3+4.3i)
#f
#|kawa:7|# (list? "curious george")
#f
```

string? Returns true if and only if its argument evaluates to a string.
char? Returns true if and only if its argument evaluates to a character.
number? Returns true if and only if its argument evaluates to some type of number.
integer? Returns true if and only if its argument evaluates to an integer.
rational? Returns true if and only if its argument evaluates to an integer or a rational.
real? Returns true if and only if its argument evaluates to a real number, be it integral, rational, or floating float.
complex? Returns true if and only if its argument evaluates to some form of a number. Since the domain of complex numbers includes the domain of all reals, which includes the domain of all rationals, which includes the domain of all integers, pretty much anything that passes a **number?** test also passes the **complex?** test.

You'll need to intuit how the above behave, or you'll need to play with Scheme for a while get the hang of them. Or you can just wing it when the assignment goes out.

boolean? Returns true if and only if its argument evaluates to a bona fide Boolean result (and not just one that can be interpreted as a Boolean.)

```
#|kawa:1|# (boolean? #t)
#t
#|kawa:2|# (boolean? #\t)
#f
#|kawa:3|# (boolean? "true")
#f
#|kawa:4|# (boolean? (integer? 99.8877665))
#t
#|kawa:5|# (boolean? (string<=? "candy" "cane"))
#t
```

symbol? Returns true if and only if its argument evaluates to a symbol.

```
#|kawa:1|# (symbol? "stringy")
#f
#|kawa:2|# (symbol? 'stringy)
#t
#|kawa:3|# (symbol? 'car)
#t
#|kawa:4|# (symbol? car)
#f
```

Note that **'car**, because it's quoted, evaluates to itself. But **car** isn't quote, so it evaluates to the code it's a symbol for.

procedure? Returns true if and only if its argument evaluates to a procedure, which in layman's terms means: if and only if it's the name of an actual function.

```
#|kawa:1|# (procedure? 'car)
#f
#|kawa:2|# (procedure? car)
#t
#|kawa:3|# (procedure? append)
#t
#|kawa:4|# (procedure? procedure?)
#t
#|kawa:5|# (procedure? "Hey, I'm a procedure!")
#f
```

There's no fooling **procedure?**. It knows the difference between the procedure bound to **car** and the symbol generated by **'car**. And look at expression 4! Talk about an ability to introspect. Wives across America are asking themselves: why can't their husbands be more like **procedure?**?

null? Returns true if and only if its argument evaluates to the empty list.

zero? Returns true if a number is logically equivalent to zero.

positive? Returns true if a real number is greater than zero. Chokes on complex numbers, unless its imaginary part is zero.

negative? Returns true if a real number is less than zero. Chokes on complex numbers, unless its imaginary part is zero.

odd?, **even?** Returns true if and only if the argument evaluates to an integer and the result if odd, even, respectively.

Comparators

equal? Takes two arguments and returns true if and only they represent the same value. The arguments can be strings, number, characters, lists—anything. **equal?** needs to traverse each of its two arguments to decide if the two are structurally identical and contain the same

primitives in all the same places. This version always does the right thing, even if it's more time consuming than things like **eq?** and **eqv?**

string=? String-specific version of **equal?**. And because the set of a strings forms a total order, Scheme includes **string<?**, **string<=?**, **string>?**, and **string>=?** as well. If case insensitivity is your thing, then you can use **string-ci=?**, **string-ci>?**, **string-ci>=?**, **string-ci<?**, and **string-ci<=?** instead.

= Number-specific version of **equal?**. And because they're numbers, **<**, **<=**, **>**, and **>** make sense as well. The relational operators work for integers, rationals, reals—even complex numbers. Physicists and EEs finally have a language that loves them back.

These particular relational operators can take two **or more** arguments. The expression evaluates to true if an only if the relational operator is respected by all neighboring argument pairs. Typically, you'll just exercise the two-argument option, but there are novel scenarios where programmers can levy off the n-arity of the relops to confirm that, say, and list of scores is sorted from high to low.

```
#|kawa:1|# (= 1 2 3)
#f
#|kawa:2|# (= 1 (- 3 2) (* 8 1/8) (remainder 16 3))
#t
#|kawa:3|# (<= 1 2 3 4 5)
#t
#|kawa:4|# (<= 1 2 3 4 -5)
#f
```

if and cond

if The **if** form in Scheme is most similar to the ternary **?:** operator in C and C++. **if** takes the form of an expression, as if **if** is some function being invoked. **if** requires three expressions as arguments: the first is the test, the second is the expression to be evaluated if the test passes (or, more specifically—if the result of the first expression is anything other than **#f**), and the third is the expression to be evaluated if the test fails. All three expressions are required—in particular, the third expression is required, because the entire **if** expression needs to evaluate to something in the event that the test fails. Interestingly enough, the two expressions needn't evaluate to the same data type.

```
#|kawa:1|# (if (string<? "apple" "kiwi") "banana" "tangelo")
banana
#|kawa:2|# (if (< 1 2 3 4 3 2 1) "sorted" '("not" "sorted"))
(not sorted)
#|kawa:3|# (if (list? car) "list"
```

```
#|(---:4|#     (if (symbol? car) "symbol" "???"))
???
```

Note that the last expression was complex enough that I took two lines to type it in. Once we start writing functions, we'll be using several lines to spell out our implementations. Don't go thinking entire Scheme programs are stretched over 50000 columns of a single line.

cond

The general form of the **cond** expression is:

```
(cond (test-1 consequent-1)
      (test-2 consequent-2)
      ...
      (test-n consequent-n)
      (else   consequent-default))
```

cond is the generalization of **if**, and works well when you'd otherwise need a chain of nested **if** expressions. The tests are all executed in the order presented until one of them evaluates to true. The expression paired up with the first successful test is then evaluated, and its results is the result of the entire **cond** expression. If all explicit tests fail, the expression attached to the **else** symbol gets evaluated as a last resort. **cond** needs to evaluate to something, and the **else** pair makes sure that happens.

```
#|kawa:1|# (cond ((procedure? cond) "Tokyo")
#|(---:2|#       ((symbol? 'loop) "Prague")
#|(---:3|#       ((integer? 222) "Berlin")
#|(---:4|#       (else "Amsterdam"))
Prague
```

The **(procedure? cond)** expression evaluated to false (**if** and **cond** are special forms, just like **and** and **or** are), so **"Toyko"** is passed over. The **(symbol? 'loop)** test succeeds, the string constant **"Prague"** evaluates to itself like all good string constants do, and the entire **cond** expression evaluates to **"Prague"**. The fact that **(integer? 222)** **would** evaluate to true is irrelevant, because evaluation never even gets that far. **cond** comes with its own form of short-circuit evaluation.