CS212 Handouts

Scheme Quick Reference

This page is a quick reference describing *briefly* the syntax and functionality of the more useful things in Scheme. The descriptions here are not meant to be the official documentation for the Scheme language. For more information, or notes on how something in Scheme evaluates, consult the Revised[5] Report on the Algorithmic Language Scheme. If you find any mistakes or things that need to be changed/added in this handout, send a message to Brandon.

Notation

Notation used in this handout:

- Things in `fixed` font are things that are typed in.
- Things in *italics* are parameters.
- "*expr* ..." means zero or more expressions.
- "$expr_1$ $expr_2$ ..." means one or more expressions.
- "[...]" means an optional syntax.

Identifiers

You can use letters, numbers and "=", "-", ">", "<", "/", "*", "?", "!" and "+" in identifiers (names). "?" is typically used as a suffix for a predicate, "!" for state modification functions, "->" in the middle of type conversions (as in number->string), and ":" for symbols from different packages (as in primes:fast-prime?).

Values

- **Strings** - Anything enclosed with quotation marks (e.g. "String")
- **Characters** - *#\char-name* (e.g. #\a, #\A, or #\space)
- **Numbers** - We have integers, floats, rationals, complex (3+4**i**) and +inf.0 or -inf.0. Scheme does not impose any limitation in the range of numbers.
- **Symbols** - *Unevaluated* identifiers (Can get a symbol with the quote function)

Predicates

Scheme provides a predicate for each type it supports. Predicates are followed by `?` and return either `#t` or `#f`

No input can satisfy more than one of these predicates: `boolean?`, `pair?`, `symbol?`, `number?`, `char?`, `string?`, `vector?`, `procedure?`.

Booleans

- `#t` means true
- `#f` means false

In addition, any object other than `#f` means true as well. It is good programming style however to use `#t` when you explicitly mean true.

Lists

- A `pair` is anything that can be taken apart with the `head` and `tail` functions (see below).
- The constant "`()`" or `empty` is the empty list constant.
- A `list` is either a `pair` or the empty list `()`. See the Induction Examples for the inductive definition of a properly defined list.

Special Forms

Special forms are expressions that do not follow standard evaluation rules:

- `(quote` *object*`)` or `'`*object*
  Both of these expressions return *object* (unevaluated). For instance, `'(1 2 3)` returns the list `(1 2 3)`.
- `(lambda (`*var* ...`)` *body* ...`)`
  this expression returns a procedure that takes some values as arguments, and when called, evaluates the *body* expression/s.
  For example, `(lambda (x) (* x x))` has the value of a one-argument method. To use the lambda expression, put it in the first position of a combination `((lambda (x) (* x x)) 5)` will evaluate to `25`.
- `(if` *test consequent* [*alternate*]`)`
  Evaluates *test*, and if it evaluates to true, then it evaluates *consequent*, otherwise, it evaluates *alternate*. Keep in mind that anything that is not `#f` is true, so `(if 0 1 2)` evaluates to `1`.
- `(when` *test* $body_1$ $body_2$...`)`
  Evaluates *test*, and, if true, evaluates *body*, otherwise it is unspecified.
- `(unless` *test* $body_1$ $body_2$...`)`
  Evaluates *test*, and, if false, evaluates *body*, otherwise it is unspecified. In other words, it's the opposite of `when`.
- `(cond (`$test_1$ $body_{11}$ $body_{12}$...`)`
  `(`$test_2$ $body_{21}$ $body_{22}$...`)`
  ...
  [`(else` $body_{else1}$ $body_{else2}$...`)`] `)`
  Evaluates $test_1$, $test_2$, ... until one evaluates to true, then evaluate its corresponding $body_n$. If all of the clauses are false, the value of the `cond` is unspecified. You may also use the "`else`" optional keyword to specify that the value of the `cond` is $body_{else}$ if all the `tests` are false.
- `(case` *compare*

`((`$value_{11}$ value$_{12}$...`)` $expr_1$`)`
`((`$value_{21}$ value$_{22}$...`)` $expr_2$`)`
...
`[(else` $expr_n$`)] )`

Evaluates *compare* and compares it to each $value_{ij}$ and if there's a match, $expr_i$ is evaluated and returned. If no values match, the value of the `case` is unspecified. You may also use the "`else`" optional keyword to specify that the value of the `case` is $expr_{else}$ if no values match.

- `(and` *expr* ...`)`
  Checks the *expr*s from left to right until one evaluates to false, in which case it returns `#f`. If they all return true, `and` returns the value of the last *expr*.
- `(or` *expr* ...`)`
  Checks the *expr*s from left to right until one evaluates to true, in which case it returns that value. If they all return false, `or` returns `#f`.
- `(not` *expr*`)`
  If *expr* evaluates to `#f`, `not` returns `#t`. Otherwise, `not` returns `#f`. Note that `not` is not actually a special form, but specified here next to its relatives.
- `(define` *var expr*`)`
  Bind the variable name *var* to the value of *expr*. `define` can be used inside bodies of functions, let forms and so on - in any of these cases, it is allowed only at the beginning of a block, and it is equivalent to a `letrec` binding. It is considered extremely bad style to define something more than once.
- `(let ((`var_1 $expr_1$`)`
  `(`var_2 $expr_2$`)`
  ...`)`
  body`)`
  Evaluates $expr_1$ through $expr_n$ and binds these values to val_1 through val_n respectively, then evaluates *body* in an environment with all the *vars* set.
- `(let*  ((`var_1 $expr_1$`)`
  `(`var_2 $expr_2$`)`
  ...`)`
  body`)`
  Evaluates $expr_n$ in an environment that has var_1 to var_{n-1} set and binds this value to var_n, then evaluates *body* in an environment with all the *var*s set.
- `(letrec ((`$name_1$ $expr_1$`)`
  `(`$name_2$ $expr_2$`)`
  ...`)`
  body`)`
  Evaluates each $expr_n$ in some unspecified order in such a way that each body can refer to another *name* in the `letrec`, and binds the resulting value to each respective $name_n$. It then evaluates the *body* in the new environment created. *expr*s can refer to other *expr*s if the reference is inside a `lambda` expression. One restriction on `letrec` is that it must be possible to evaluate each *expr* without assigning or referring to the value of any *name*. In the most common uses of `letrec`, all the *expr*s are lambda expressions and the restriction is satisfied automatically.
- `(set!` *var expr*`)`
  Changes the value of *var* to that of *expr*. The return value of `set!` is unspecified. **CS212 students may not use this form until covered in class.**

- `(begin` $expr_1$ $expr_2$ ... $expr_n$`)`
  Evaluates the *expr*s from left to right and returns the the value of $expr_n$. Useful for putting a sequence of expressions where only one is expected. `begin` is useful only for side-effect functions, otherwise it doesn't make much sense.
- `(delay` *expr*`)`
  Creates a promise containing *expr* unevaluated.
- `(force` *expr*`)`
  Evaluates *expr*, which should return a promise. The unevaluated contents of the promise are then evaluated and returned. `force` is the only way to evaluate the contents of a promise.
- `(write` *expr*`)` or `(display` *expr*`)`
  Evaluates *expr* and displays the result to the screen. The return value of print or display is unspecified.

Equality Testing

- `(eq?` *x y*`)`: True if *x* and *y* reference the same object.
- `(equal?` *x y*`)`: True if *x* and *y* are "congruent"; which roughly means if they print the same.
- `(=` *x y*`)`: Used only for checking equality of numbers.

List Structure Operators

- `(null?` *x*`)`: Returns `#t` if *x* is the the empty list `()` and `#f` for all other objects.
- `(cons` *x y*`)`: Returns a `pair` whose `head` is *x* and whose `tail` is *y*.
- `(head` *l*`)`: Returns the first element of the list *l*.
- `(tail` *l*`)`: Returns the tail of the list *l*.
- `(set-car!` *obj pair*`)`: `set!`s the `head` of *pair*. **Don't use until presented in class.**
- `(set-cdr!` *obj pair*`)`: `set!`s the `tail` of *pair*. **Don't use until presented in class.**
- `(list` x_1 ...`)`: Puts x_1, ... into a list.
- `(list-ref` *l k*`)`: Returns the *k*th element of the list *l*.
- `(length` *l*`)`: Returns the length of the list *l*.
- `(append` $list_1$ ...`)`: Appends the lists together.
- `(append!` $list_1$ ...`)`: Destructively appends the i^{th} list to the $(i+1)^{th}$ list. Returns the first list. **Don't use until presented in class.**
- `(reverse` *l*`)`: Turns the list *l* around.
- `(member` *obj l*`)`: Searches the list *l* for something `equal?` to *obj*, returning the sub-list whose head is *obj* if found, else `#f`.
- `(assoc` *obj l*`)`: For this function to work, *l* must be a list of pairs. `assoc` searches the list looking for a pair whose head is `equal?` to *obj*, returning the pair whose head is *obj* if found, else `#f`.
- `(memq` *obj l*`)`: Same as `member`, except it uses the `eq?` function.
- `(assq` *obj l*`)`: Same as `assoc`, except it uses the `eq?` function.

Arithmetic and Numeric Operators

- `(=` *x y*`)`, `(<` *x y*`)`, `(>` *x y*`)`, `(<=` *x y*`)`, `(>=` *x y*`)`: Various forms of numerical comparison.

- `(max` x_1 ...`)`, `(min` x_1 ...`)`: Return the maximum, minimum of all x's.
- `(gcd` x_1 ...`)`, `(lcm` x_1 ...`)`: Return the greatest common divisor, least common multiple of all x's. The result is always non-negative.
- `(+` x_1...`)`: Addition.
- `(*` x_1...`)`: Multiplication.
- `(-` x_1 x_2`)`: Subtraction.
- `(/` x_1 x_2`)`: Division - result type depends on the input argument types.
- `(expt` x_1 x_2`)`: Power operator. Raises x_1 to the x_2 power.
- `(-` x`)`: Negation -x.
- `(/` x`)`: Reciprocal 1/x.
- `(quotient` n_1 n_2`)`: Quotient after integer division. (Return value has sign of product of arguments).
- `(remainder` n_1 n_2`)`: Remainder after integer division. (Return value has sign of dividend).
- `(modulo` n_1 n_2`)`: Clock arithmetic, n_1 mod n_2. (Return value has sign of divisor).
- `(zero?` x`)`, `(positive?` x`)`, `(negative?` x`)`: Checks whether x=0, x>0 or x<0.
- `(even?` x`)`, `(odd?` x`)`: Checks whether x is even or odd
- `(1+` x`)`, `(1-` x`)`: Shorthand for `(+` x `1)` and `(-` x `1)`
- `(abs` x`)`: Absolute value of x.
- `(floor` x`)`: Integer closest to x whose absolute value <= |x|. Result type depends upon x.
- `(ceiling` x`)`: Integer closest to x whose absolute value >= |x|. Result type depends upon x.
- `(round` x`)`: Integer closest to x, rounding to even when x is halfway between two integers.
- `(sqrt` x`)`: Square root.
- `(sin` x`)`, `(cos` x`)`, `(tan` x`)`: Trigonometric functions.
- `(asin` x`)`, `(acos` x`)`, `(atan` x`)`: Inverse trigonometric functions.
- `(exp` x`)`: The mathematical constant e raised to the power x
- `(log` x`)`: The natural logaritm of x. `(log 0)` returns -inf.0.

Vector Operators

- `(make-vector` n [i]`)`: Creates a new vector of n elements (if i is given the initial values of the elements are all i, otherwise all the elements are undefined), indexed from *0* to *n-1*.
- `(vector` x_1 ...`)`: Evalutes x_1, ... and puts them into a vector.
- `(vector-length` *vec*`)`: Returns the length of the vector *vec*, which is the number of elements it can store, (using constant time).
- `(vector-ref` *vec i*`)`: Get *i*'th element of *vec* (in constant time). *i* must be a valid index.
- `(vector-set!` *vec i val*`)`: Store *val* at element *i* of *vec* (in constant time).

Higher Order Functions

- `(apply` *proc arg-list*`)`: Call *proc* with *arg-list* as arguments.
- `(map` *func* $list_1$ ...`)`: Applies *func* element-wise to the elements of $list_1$, ... (that is, the first of each, the second of each, etc.) and collects the results in a list. The order it does it is not specified and probably not the one you want, but the order they appear in the result list is the right one. If more than one list is given, each list must be of the same length. There should be the same number of lists as there are parameters for *func*.

- `(filter` *pred l*`)`: Returns the list of elements from *l* which satisfy *pred*.

Last update: 01/19/00 11:35 PM by BRB
Written by Brandon Bray and Eli Barzilay