Commands

Racket

Functions

```
(define (<name> <args>) <body>) => define function
ex: (define (foo x) (+ x 1))

(lambda (<args>) <body>) => anonymous function
ex: (lambda (x) (+ x 1))

(let ([<name1> <val1>] [<name2> <val2>] ...) <body>) => define local variables
ex: (let ([x 1] [y 2]) (+ x y)) => 3
```

Datatypes

```
'() = null => empty list
'(1 2 3) => list of 1, 2, 3

`name => evaluate name
ex: `(1 2 ,(+ 1 2)) => (1 2 3)
ex: `test => `test
ex: `12 => 12

"string" => string

#t = true => bool true
#f = false => bool false
```

Comparisons

```
(if <cond> <then> <else>) => if statement
ex: (if (< x 0) (- x) x)

(cond (<cond1> <then1>) (<cond2> <then2>) ... (else <thenFinal>)) => if-else-if
statement
ex: (cond [(< x 0) (- x)] [else x])
Note: [] is equivalent to ()

(equal? <val1> <val2>) => true if equal
ex: (equal? 1 1) => true
```

```
(= <num1> <num2>) => true if numerically equal
ex: (= 1 1) => true

(zero? <num>) = (= <num> 0) => true if zero
ex: (zero? 0) => true

(even? <num>) => true if even
ex: (even? 2) => true

(odd? <num>) => true if odd
ex: (odd? 1) => true

(positive? <num>) => true if positive
ex: (positive? 1) => true

(negative? <num>) => true if negative
ex: (negative? -1) => true

(list? <val>) => true if list
ex: (list? '(1 2 3)) => true

(number? <val>) => true if number
ex: (number? 1) => true

(string? <val>) => true if string
ex: (string? "test") => true

(</<=/>/>= <num1> <num2>) => true if numerically less than, less than or equal to,
greater than, greater than or equal to
ex: (< 1 2) => true

(and <cond1> <cond2> ...) => true if all conditions are true
ex: (and (< x 0) (< y 0))

(or <cond1> <cond2> ...) => true if any condition is true
ex: (or (< x 0) (< y 0))

(not <cond>) => true if cond is false
```

Numeric Operations

```
(+/-/*// <num1> <num2> ...) => arithmetic
ex: (+ 1 2 3 4) => 10

(quotient <num1> <num2>) => integer division
ex: (quotient 5 2) => 2

(remainder <num1> <num2>) => remainder with sign of num1
ex: (remainder -5 3) => -2
```

```
(modulo <num1> <num2>) => remainder with sign of num2
ex: (modulo -5 3) => 1

(abs <num>) => absolute value
ex: (abs -5) => 5

(sqrt <num>) => square root
ex: (sqrt 4) => 2

(expt <num> <power>) => num to the power of power
ex: (expt 2 3) => 8

(round <num>) => round to nearest integer (round to even)
ex: (round 1.5) => 2

(floor <num>) => floor
ex: (floor 1.5) => 1

(ceiling <num>) => ceiling
ex: (ceiling 1.5) => 2

(truncate <num>) => round towards zero
ex: (truncate 1.9999) => 1
```

Lists

```
(cons <elem> <list>) => add elem to front of list
ex: (cons 1 '(2 3)) => (1 2 3)

(car <list>) => first element of list
ex: (car '(1 2 3)) => 1

(cdr <list>) => list without first element / second element of pair
ex: (cdr '(1 2 3)) => (2 3)

(append <list1> <list2> ...) => add lists in order
ex: (append '(1 2) '(3 4)) => (1 2 3 4)

(length <list>) => length of list
ex: (length '(1 2 3)) => 3

(empty? <list>) => true if empty
ex: (empty? '()) => true
ex: (empty? '(1 2)) => false
(null? <list>) => true if empty

(reverse <list>) => reverse list
ex: (reverse '(1 2 3)) => (3 2 1)
```

```
(member <elem> <list>) => returns list starting with elem or false
ex: (member 2 '(1 2 3)) => '(2 3)

(map <func> <list>) => apply func to each element of list
ex: (map (lambda (x) (+ x 1)) '(1 2 3)) => (2 3 4)

(filter <func> <list>) => return list of elements for which func returns true
ex: (filter positive? '(1 -2 3)) => (1 3)

(remove <elem> <list>) => remove first occurence of elem from list
ex: (remove 2 '(1 2 2 3)) => (1 2 3)

(andmap <func> <list>) => true if func returns true for all elements of list
ex: (andmap positive '(1 2 3)) => true

(ormap <func> <list>) => true if func returns true for any element of list
ex: (ormap positive '(1 -2 -3)) => false

(apply <func> <v1> <v2> ... '(lst1 lst2 ...) ) => (func v1 v2 ... lst1 lst2 ...)
ex: (apply + 1 2 '(3 4)) => 10

(foldl <func> <init> <list1> <list2> ...) => like map, but takes last result as last
argument
ex: (foldl cons '() '(1 2 3)) => '(3 2 1)

(foldr <func> <init> <list1> <list2> ...) => like foldl, but lists are traversed in
reverse order
ex: (foldr cons '() '(1 2 3)) => '(1 2 3)

(flatten <list>) => flatten list
ex: (flatten '(1 (2 3) 4)) => '(1 2 3 4)
```

Strings

```
(string-length <string>) => length of string
ex: (string-length "hello") => 5

(string-append <string1> <string2> ...) => concatenate strings
ex: (string-append "hello" "world") => "helloworld"

(string=? <string1> <string2>) => true if strings are equal
ex: (string=? "hello" "hello") => true

(string<? <string1> <string2>) => true if strings are in lexicographical order
ex: (string<? "a" "b") => true
```

Prolog

Legend: func\args function and number of arguments it takes

Facts

```
name(<args>). => asserts name(<args>) as a fact / rule

,/2 => arg1 and arg2
;/2 => arg1 or arg2
\+/1 => true if arg fails
not/1 => \+/1
A :- B. => A <- B

assert/1 = assertz/1 => asserts fact as last clause
ex: assertz(parent('Bob', 'Jane')).

asserta/1 => asserts fact as first clause

!/0 => cut, prevents backtracking

A => Variable (Capitalized)
a => Atom (Lowercase)
_A => Singleton Variable (starts with underscore); don't care about what it is bound
to
_ => like _A, but can match different things in the same rule
ex: gives(X,_,_) => arg 2 and 3 can be different

true/0 => always succeeds
false/0 => fails
repeat/0 => always succeeds, infinite choicepoints

inf/0 => positive infinity
```

Comparisons

```
dif/2 => true if both args are different (prefer this to \=, \== or =\=), args don't
need to be grounded

=/2 => unifies both args
\=/2 => \+ arg1 = arg2

is/2 => evaluates arg2 and unifies it with arg1 with the result (arg2 must be
grounded, arg1 preferably unbound)

==/2 => true if both args are equivalent
\==/2 => \+ arg1 == arg2
```

```
=:=/2 => true if both args evaluate to the same number
=\=/2 => true if both args evaluate to different numbers

</2 => true if arg1 evaluates to a lesser number than arg2
>/2 => true if arg1 evaluates to a greater number than arg2
=</2 => true if arg1 evaluates to less than or equal to arg2
>=/2 => true if arg1 evaluates to greater than or equal to arg2

between/3 => true if arg3 is between arg1 and arg2 (if arg3 is a free variable
generates all possible values)
ex: between(1, 3, 2). => true

var/1 => true if arg is currently a free variable
nonvar/1 => true if arg is not a free variable
```

Arithmetic

```
+/2 => addition
-/2 => subtraction
*/2 => multiplication
//2 => division
///2 => integer division
mod/2 => modulo
rem/2 => remainder
```

Lists

```
member/2 => true if arg1 is a member of arg2
ex: member(1, [1, 2, 3]). => true

length/2 => true if arg2 is the length of arg1
ex: length([1, 2, 3], 3). => true

append/3 => true if arg1 and arg2 appended together is arg3
ex: append([1, 2], [3, 4], X). => X = [1, 2, 3, 4].

reverse/2 => true if arg2 is the reverse of arg1
ex: reverse([1, 2, 3], [3, 2, 1]). => true

last/2 => true if arg2 is the last element of arg1
ex: last([1, 2, 3], 3). => true

nth0/3 => true if arg3 is the arg1's element of arg2 (0-indexed)

maplist/2 => true if arg1 succeeds for each element of arg2
ex: maplist(between(1,5), [1, 2, 3]). => true
```

```
sum_list/2 => true if arg2 is the sum of arg1's elements
ex: sum_list([1, 2, 3], 6). => true

[H|T] => head and tail of pair
ex: [H|T] = [1, 2, 3]. => H = 1, T = [2, 3].

[] => empty list

[a,b,c] => list, equals [a|[b|[c|[]]]]
```

Special

```
findall/3 => true if arg3 is a list of all possible values of arg1 that satisfy arg2
ex: findall(X, between(1, 3, X), L). => L = [1, 2, 3].
```