Scheme

doc. version 1.1

Arithmetic expressions	
Written version	**Scheme version**
5+6+7	(+ 5 6 7)
5(6+7)	(* 5 (+ 6 7))
¾	(/ 3 4) or 3/4
| -4 | (absolute value)	(abs -4) -> 4
$\sqrt{2}$	(sqrt 2) -> 1.4142135623730951
$\sqrt{-1} = 0 + 1i$	(sqrt -1) -> 0+1i
2^3	(expt 2 3) -> 8
4.87^2	(sqr 4.87) -> 23.7169
Remainder when 15 is divided by 6 Also written: 15 mod 6	(remainder 15 6) -> 3
15/6 after removing the remainder Also written: $\left\lfloor \frac{15}{6} \right\rfloor$	(quotient 15 6) -> 2

Defining variables and functions	
Mathematical version	**Scheme version**
a = 5.6	(define a 5.6)
b = 2a+3	(define b (+ (* 2 a) 3)
f(x) = x+1	(define (f x) (+ x 1)) or (define f (lambda (x) (+ x 1)))
f(5) -> 6	(f 5) -> 6
g(x,y) = 2x + y	(define (g fred harry) (+ (* 2 fred) harry))
$h(x, y) = (2x + y)^2$	(define (h a ron) (sqr (+ (* 2 a) ron))) or we can use the function g above (if we've previously defined it)… (define (h a ron) (sqr (g a ron)))
h(2,3) -> 49	(h 2 3) -> 49

Boolean datatype	**Scheme written version**
TRUE	#t or true
FALSE	#f or false

Comparison operators for numbers	**Scheme version**
> (greater than)	(> 5 4) -> #t
>= (greater than or equal to)	(>= 5 6) -> #f
= (equal to) (but only for numbers)	(= 5 5.0) -> #t (in WeScheme)
< (less than)	(< 7 6) -> #f
<= (less than or equal to)	(<= 6 6) -> #t

Comparison operator for all simple datatypes (including numbers)	**Scheme version**
Equal	(equal? 4 4.0) -> #t (in WeScheme) (equal? 4 4.0) -> #f (in SchemingBat) (equal? 4 4.0) -> #f (in DrRacket) (equal? #t #f) -> #f

Logical conjunction operators	**Scheme version**
AND	(and #t #t) -> #t (and #f #t) -> #f (and (= 4 4) (> 4 5)) -> #f (and #t #t #f #t #t) -> #f
OR	(or #f #t) -> #t (or #f #f) -> #f (or (= 4 4) (> 4 5)) -> #t (or #f #t #f) -> #t
NOT	(not (= 4 4)) -> #f (correction)

Decisions using "if"

The IF expression is a list with 4 parts:
(if test-part true-part false-part)

The **test-part** is an expression that must return #t or #f. For instance, these three are valid tests:

```
(> 4 3)
(<= x 0)
(=  x  (+ 1 y))
```

Or more complicated ones:

```
(and (>= x 5) (<= x 10))  ; is x between 5 and 10?
(or (< x 5) (> x 10))  ; is x not between 5 and 10?
```

In fact, any expression that asks a yes/no question is a valid test.

The **true-part** is an expression that will be evaluated if the **test-part** is True, and its answer will be what the entire if-expression returns

Likewise, the **false-part** will be an expression that will be used (evaluated) if the **test-part** is False, and that will be the answer.

Examples:

(if (> 4 3) (+ 3 1) 18) -> 4
that's because the **test-part**, (> 4 3), is true and so we evaluate the **true-part** (+ 3 1) and that's our answer

(if (= 5 6) (+ 3 1) 18) -> 18
..because the **test-part**, which is (= 5 6), is false, and so the **false-part,** which is 18 is the answer

Functions making decisions using IF	

abs(x) ; the absolute value	(define (abs x)
	  (if (>= x 0) ; test-part
	  x ; true-part
	  (* -1 x) ; false-part
	  )
	)

A function f(x), which is equal to:
$f(x) = \begin{cases} x^2 + 1 \ \ if \ x \geq 3 \\ x - 2 \ if \ x < 3 \end{cases}$
	(define (f x)
	  (if (>= x 3)
	  (+ (* x x) 1)
	  (- x 2)
	  )
	)
	Or, written on a single line:
	(define (f x) (if (>= x 3) (+ (* x x) 1) (- x 2)))

A function grade(score) which is:
$grade(score)$
$= \begin{cases} 0 \ if \ score < 65 \\ 1 \ if \ score \geq 65 \ and < 90 \\ 2 \ if \ score \geq 90 \end{cases}$
	(define (grade score)
	  (if (< score 65)
	  0
	  (if (and (>= score 65) (< score 90))
	  1
	  2
	  )
	  )
	)