Negation

```
CLIPS> (clear)
CLIPS> (defrule green-light
             (light green)
         =>
             (printout t "Walk" crlf))
CLIPS> (defrule red-light
            (light red)
          =>
            (printout t "Don't Walk" crlf))
CLIPS> (defrule walk-sign
            (walk-sign-says dont-walk)
          =>
            (printout t "Don't Walk" crlf))
CLIPS> (assert (light red))
<Fact-0>
CLIPS> (run)
Don't Walk
CLIPS> (defrule walk
            (light ~green)
         =>
            (printout t "Don't Walk" crlf))
CLIPS> (assert (light broken))
<Fact-0>
CLIPS> (run)
Don't Walk
CLIPS> (assert (light yellow))
<Fact-1>
CLIPS> (run)
Don't Walk
CLIPS>
```

Here, the negation operator was used to handle the cases, if anything other than green, don't walk.

Exercise: Resolving Conflicts

```
CLIPS> (clear)
CLIPS> (defrule green-light
             (light green)
         =>
             (printout t "Walk" crlf))
CLIPS> (defrule red-light
            (light red)
          =>
            (printout t "Don't Walk" crlf))
CLIPS> (defrule walk-sign
            (walk-sign-says dont-walk)
          =>
            (printout t "Don't Walk" crlf))
CLIPS> (assert (light green) (walk-sign-says dont-walk))
<Fact-2>
CLIPS> (run)
Don't Walk
Walk
CLIPS>
```

Here we have the situation that there is a conflict between the light and the walk signal. What is wanted is a set of rules that check both the lights and the walk signals and allows walking only if both say it is okay.

Conditionals

```
CLIPS> (clear)
CLIPS> (defrule cautious
           (light yellow|blinking-yellow)
        =>
           (printout t "Be Cautious" crlf))
CLIPS> (assert (light yellow))
<Fact-0>
CLIPS> (run)
Be Cautious
CLIPS> (assert (light red))
<Fact-1>
CLIPS> (run)
CLIPS> (assert (light blinking-yellow))
<Fact-2>
CLIPS> (run)
Be Cautious
CLIPS>
```

Saving Result of Conditional

```
CLIPS> (clear)
CLIPS> (defrule cautious
          (light ?color&yellow|blinking-yellow)
         =>
          (printout t "Be cautious because light is " ?color crlf))
CLIPS> (defrule not-yellow-red
          (light ?color&~red&~yellow)
          =>
         (printout t "Go, since light is " ?color crlf))
CLIPS> (assert (light red))
<Fact-0>
CLIPS> (run)
CLIPS> (assert (light yellow))
<Fact-1>
CLIPS> (run)
Be cautious because light is yellow
CLIPS> (assert (light green))
<Fact-2>
CLIPS> (run)
Go, since light is green
CLIPS> (assert (light purple))
<Fact-3>
CLIPS> (run)
Go, since light is purple
CLIPS>
```

Simple Arithmetic

```
CLIPS> (clear)
CLIPS> (defrule addition
(numbers ?x ?y)
=>
(assert (answer-plus (+ ?x ?y))))
CLIPS> (assert (numbers 2 3))
<Fact-0>
CLIPS> (run)
CLIPS> (facts)
f-0     (numbers 2 3) CF 1.00
f-1     (answer-plus 5) CF 1.00
For a total of 2 facts.
CLIPS>
```

Note:
LISP structure for functions
answer-plus is a fact, not a function call like +

```
CLIPS> (clear)
CLIPS> (deffacts database
                (stock A 2.0)
                (stock B 5.0)
                (stock C 7.0))
CLIPS> (defrule addition
          (numbers ?x ?y)
          (stock ?ID =(sqrt (+ (** ?x 2) (** ?y 2))))
        =>
          (printout t "Stock ID= " ?ID crlf))
CLIPS> (reset)
CLIPS> (assert (numbers 3 4))
<Fact-4>
CLIPS> (run)
Stock ID= B
CLIPS>
```

sqrt, plus, exponentiation, =(

Arithmetic operations beyond two argument (LISP-like)

```
CLIPS> (clear)
CLIPS> (defrule addition
               (numbers ?x ?y ?z)
        =>
               (assert (answer-plus (+ ?x ?y ?z))))
CLIPS> (assert (number 1 2 3))
<Fact-0>
CLIPS> (run)
CLIPS> (facts)
f-0     (number 1 2 3) CF 1.00
For a total of 1 fact.
CLIPS>
```

bind

```
CLIPS> (clear)
CLIPS> (defrule addition
             (numbers ?x ?y)
          =>
             (assert (answer (+ ?x ?y)))
             (bind ?answer (+ ?x ?y))
             (printout t "answer is " ?answer crlf))
CLIPS> (assert (numbers 2 2))
<Fact-0>
CLIPS> (run)
answer is 4
CLIPS> (facts)
f-0     (numbers 2 2) CF 1.00
f-1     (answer 4) CF 1.00
For a total of 2 facts.
```

bind with multiple field values

```
CLIPS> (defrule bind-values-demo
            (initial-fact)
          =>
            (bind ?duck-bachelors (create$ Dopey Dorky Dinky))
            (bind ?happy-bachelors-mv (create$ Dopey))
            (bind ?none (create$))
            (printout t
                "duck-bachelors " ?duck-bachelors crlf
                "duck-bachelors-no-() " (implode$ ?duck-bachelors) crlf
                "happy-bachelors-mv " ?happy-bachelors-mv crlf
                "none " ?none crlf))
CLIPS> (reset)
CLIPS> (run)
duck-bachelors (Dopey Dorky Dinky)
duck-bachelors-no-() Dopey Dorky Dinky
happy-bachelors-mv (Dopey)
none ()
CLIPS>
```

Note use of (initial-fact), as starting point

```
CLIPS> (defrule calculate-hypontenuse
            (dimensions ?base ?height)
         =>
            (printout t "Hypontenuse=" (hypontenuse ?base ?height) crlf))
CLIPS> (assert (dimensions 3 4))
<Fact-0>
CLIPS> (run)
Hypontenuse=5.0
CLIPS>
```

Note that a function and a fact look similar