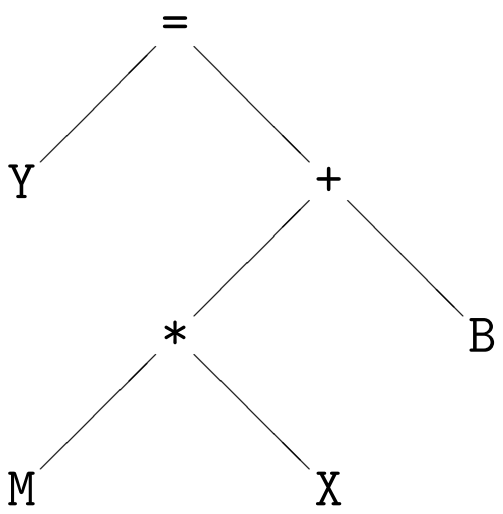
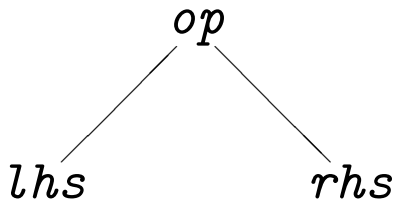
Solving Equations

Simple equations can be solved by search, using rules of algebra to transform equations into equivalent forms until an equation for the desired variable is produced.

We will think of the same data structure in several ways:

- **Equation:** $y = m \cdot x + b$
- **List structure:** `(= Y (+ (* M X) B))` or `(` *op lhs rhs* `)` recursively
- **Tree:**





- **Executable code:** `eval` can evaluate an expression using a set of variable bindings.

Solving an Equation by Search

We can perform algebraic operations by manipulating the list structure representation of an expression tree (taking apart the original tree and constructing a new tree). To solve an equation `e` for a desired variable `v`:

- **Base cases:**
  - If the `lhs` of `e` is `v`, return `e`.
  - If the `rhs` of `e` is `v`, rewrite `e` to switch the `lhs` and `rhs` of `e`, and return that.
  - If only an undesired variable or constant is on the right, (`rhs` is not `consp`), fail by returning `null`.
- **Recursive case:** Rewrite `e` using an algebraic law, and try to solve that equation. Return the first result that is not `null`.

  Often, there are two possible ways to rewrite an equation; it is necessary to try both. Thus, the process will be a binary tree search analogous to the robot mouse searching for cheese.

Examples: Base Cases

```
>(solve '(= x 3) 'x)

(= X 3)

>(solve '(= 3 x) 'x)

(= X 3)

>(solve '(= 3 y) 'x)

NIL
```

Recursive Cases: Operators

The recursive case has a `rhs` that is an operation:
`(=` α `(` *op* β γ `))`

We are hoping that the desired variable will be somewhere in β or γ; to get to it, we must apply some kind of inverse operation to both sides of the equation to get rid of *op* and isolate β or γ.

In general, there may be two inverse operations to try.

We can produce the result of the inverse operation by constructing a new equation from the given one, e.g., given:
`(=` α `(+` β γ `))`

we can construct two new possibilities:

`(= (-` α β `)` γ `)` (subtract β from both sides)

`(= (-` α γ `)` β `)` (subtract γ from both sides)

After making a new equation, we simply call `solve` to try to solve *that* equation. We return the first solution that is not `null`.

Recursive Tree Search

In effect, the search process will rewrite the original equation in every possible legal way. Most of these will not be what we want, and will fail, but one of them will be solved for the desired variable.

```
>(solve '(= y (+ x b)) 'x)

  1> (SOLVE (= Y (+ X B)) X)
    2> (SOLVE (= (- Y X) B) X)
    <2 (SOLVE NIL)
    2> (SOLVE (= (- Y B) X) X)
    <2 (SOLVE (= X (- Y B)))
  <1 (SOLVE (= X (- Y B)))

(= X (- Y B))

>(solve '(= y (+ (* m x) b)) 'x)

(= X (/ (- Y B) M))
```

Big O and Termination

We want to make sure that we cannot get into a loop by transforming an equation endlessly.

Well-founded Ordering: If a program has an input that is finite and gets smaller in each recursion, and the program stops when the input reaches a lower boundary, then the program is guaranteed to terminate.

Our program assumes that initially the `lhs` is only a single variable. Each recursive step makes the `rhs` smaller.

We don't have to worry about Big O for this problem because the number of operations is limited by the size of the expression tree, which is always small.

Solving a Physics Problem

With the collection of programs that we now have, solving a physics problem becomes easy:

- Make a list (set) of the variables in the problem (desired variable and variables whose values are given).
- Find an equation that involves those variables.
- Solve the equation for the desired variable.
- `eval`uate the `rhs` of the equation for the given values.

Solving Sets of Equations

Given:

- a set of equations

`fall:`

```
((= gravity '(q 9.80665 (/ m (* s s))))
 (= horizontal-velocity '(q 0 (/ m s)))    ; default
 (= height   (* 1/2 (* gravity (expt time 2))))
 (= velocity (* gravity time))             ; vertical
 (= kinetic-energy
    (* 1/2 (* mass (expt total-velocity 2))))
 (= horizontal-distance (* horizontal-velocity
                           time))
 (= total-velocity
    (sqrt (+ (expt velocity 2)
             (expt horizontal-velocity 2)))))
```

- a set of variables with known values:
  `((TIME 4))`
- a variable whose value is desired: `HEIGHT`

Solving a Set of Equations by Search

- Try to find an equation where all variables are known except one.

  ```
  (= F (* M A))
  ```

- Solve the equation for that variable.

  ```
  (= A (/ F M))
  ```

- Substitute the known values into the right-hand side of the solved equation (Lisp function `sublis`).

  ```
  (= A (/ 8 2))
  ```

- Evaluate the resulting expression (Lisp function `eval`) to give the value of the new variable. Add that variable to the binding list.

  ```
  (= A 4)
  ```

- Keep trying until you get the value of the variable you want (or quit if you stop making any progress).

Solving Physics Story Problems

By combining the techniques we have discussed with a simple English parser, a remarkably small Lisp program can solve physics problems stated in English:

```
>(phys '(what is the area of a circle
              with radius = 2))

12.566370614359172

>(phys '(what is the circumference of a circle
              with area = 12))

12.279920495357862

>(phys '(what is the power of a lift
              with mass = 5 and height = 10
              and time = 4))

122.583125
```