Solving a Set of Equations by Search

Suppose that we have an association list of known variables and their values, and a list of equations:

```
values = ((m 2) (f 8))
```

- If the desired variable has a known value, return it.
- Try to find an equation where all variables are known except one: `(= f (* m a))`
- Solve the equation for that variable: `(= a (/ f m))`
- Evaluate the right-hand side of the solved equation using the values of the known variables (function `myevalb`) to give the value of the new variable. Add that variable to the binding list:

  ```
  values = ((a 4) (m 2) (f 8))
  ```

- 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
```

Generating Code from Equations

The first programming language, `FORTRAN`, is an abbreviation for `FOR`mula `TRAN`slation. There is a long history of similarity between equations and programs.

Suppose that we have a set of physics equations, and we want a program to calculate a desired variable from given variable values.

Each time an equation is solved for a new variable,

- add the new variable to a list of variables
- `cons` the solved equation onto a list of code

If the variable that is solved for was the desired variable, `cons` a return statment onto the code.

At the end, reverse the code (`reverse`).

solveqnsc Example

`solveqnsc` creates a list of code equations which can be executed (in order) as a program to give a desired answer.

```
(solveqnsc
  codelist   ; code list, initially '()
  equations  ; list of physics equations
  knowns     ; variables that are known
  var)       ; desired variable

; a few equations for a circle
(def eqnsc
     '((= area (* pi (expt radius 2)))
       (= circumference (* pi diameter))
       (= pi 3.1415926)
       (= diameter (* 2 radius))  ) )
```

`solveqnsc` will go through the list of equations, looking for an equation with exactly one unknown. It will solve that equation and add it to its list, plus add the unknown to this list of knowns. The returned list codelist will be backwards.

solveqnsc Example

```
(def eqnsc
     '((= area (* pi (expt radius 2)))
       (= circumference (* pi diameter))
       (= pi 3.1415926)
       (= diameter (* 2 radius))  ) )

; (solveqnsc codelist equations knowns var)
(solveqnsc '() eqnsc '(radius) 'circumference)
  (solveqnsc ((= pi 3.1415926))
                 eqnsc (pi radius) circumference)
    (solveqnsc ((= area (* pi (expt radius 2)))
                (= pi 3.1415926))
                   eqnsc (area pi radius) ...)
      (solveqnsc ((= diameter (* 2 radius))
                  (= area (* pi (expt radius 2)))
                  (= pi 3.1415926))
                   eqnsc (diameter area pi radius)
        (solveqnsc ((= circumference (* pi diameter
                    (= diameter (* 2 radius))
                    (= area (* pi (expt radius 2)))
                    (= pi 3.1415926)) eqnsc
                   (circumference diameter area
                      pi radius) ,,,)
```

Eliminating Unused Equations

The opportunistic algorithm, computing whatever can be computed from the available values, may generate some equations that are valid but not needed for finding the desired value.

Optimizing compilers use two notions, *available* and *busy*.

- A value is *available* at a point p if it has been assigned a value above the point p in a program, e.g. as an argument of a subroutine or as the left-hand-side of an assignment statement.
- A value that is the left-hand-side of an assignment statement is *busy* or *live* if it will be used at a later point in the program.

We can eliminate equations whose lhs is not busy by proceeding backwards through the code:

- Initially, the desired value is busy.
- For each equation, if the lhs of the equation is a member of the busy list, keep the equation, and add its rhs variables to the busy list. Otherwise, the equation can be discarded.

filtercode Example

```
; (defn filtercode [codelist needed] ...)
(filtercode ((= circumference (* pi diameter))
             (= diameter (* 2 radius))
             (= area (* pi (expt radius 2)))
             (= pi 3.1415926))
            (circumference))
  (filtercode ((= diameter (* 2 radius))
               (= area (* pi (expt radius 2)))
               (= pi 3.1415926))
              (diameter pi circumference))
    (filtercode ((= area (* pi (expt radius 2)))
                 (= pi 3.1415926))
                (radius diameter pi circumference))
      (filtercode ((= pi 3.1415926))
                  (radius diameter pi circumference))
        (filtercode ()
                    (radius diameter pi circumference))

((= circumference (* pi diameter))
 (= diameter (* 2 radius))
 (= pi 3.1415926))
```

Equations to Code

The result of `filtercode` was:

```
((= circumference (* pi diameter))
 (= diameter (* 2 radius))
 (= pi 3.1415926))
```

We can produce a program from this list using `tojava`:

```
; (solvecode 'rtoc eqnsc '(radius) 'circumference)
("  public static double rtoc( double radius) {"
 "    double pi=3.1415926;"
 "    double diameter=2*radius;"
 "    double circumference=pi*diameter;"
 "    return circumference;"
 "}")
```

Printing these strings produces a usable Java program.