In [2]:
```
using NamedArrays      # make sure you run Pkg.add("NamedArrays") first!

# import Stigler's data set
raw = readdlm("stigler.csv",',');
(m,n) = size(raw)

n_nutrients = 2:n      # columns containing nutrients
n_foods = 3:m          # rows containing food names

nutrients = raw[1,n_nutrients][:]   # the list of nutrients (convert to 1-D array)
foods = raw[n_foods,1][:]           # the list of foods (convert to 1-D array)

# lower[i] is the minimum daily requirement of nutrient i.
lower = Dict( zip(nutrients,raw[2,n_nutrients]) )

# data[f,i] is the amount of nutrient i contained in food f.
data = NamedArray( raw[n_foods,n_nutrients], (foods,nutrients), ("foods","nutrients") );


#----Model----#
using JuMP
m = Model()

#Amount of daily food types in the diet
@defVar(m, diet[foods] >= 0)

#Total amount of daily nutrient n in the diet
@defExpr(totNutrient[n in nutrients], sum{data[f, n]*diet[f], f in foods})

#Meet RDA
@addConstraint(m, meetRDA[n in nutrients], totNutrient[n] >= lower[n])

#Minimize annual diet cost
@setObjective(m, Min, sum{diet[f]*1*365, f in foods})

solve(m)
#Display results
println("Annual diet cost: ", getObjectiveValue(m), " dollars")
println()
println("Daily diet composition (in dollar unit)")
println(getValue(diet))
```

```
Annual diet cost: 39.66173154546625 dollars

Daily diet composition (in dollar unit)
diet: 1 dimensions, 77 entries:
 [                 Apples] = 0.0
 [         Asparagus (can)] = 0.0
 [                  Bacon] = 0.0
 [                Bananas] = 0.0
 [                 Butter] = 0.0
 [                Cabbage] = 0.011214435246144865
 [                Carrots] = 0.0
 [                 Celery] = 0.0
 [        Cheese (Cheddar)] = 0.0
 [               Chocolate] = 0.0
 [             Chuck Roast] = 0.0
 [                  Cocoa] = 0.0
 [                 Coffee] = 0.0
 [              Corn (can)] = 0.0
 [             Corn Flakes] = 0.0
 [               Corn Meal] = 0.0
 [              Corn Syrup] = 0.0
 [                  Cream] = 0.0
 [                 Crisco] = 0.0
 [                   Eggs] = 0.0
 [  Evaporated Milk (can)] = 0.0
 [             Green Beans] = 0.0
 [       Green Beans (can)] = 0.0
 [             Ham, smoked] = 0.0
 [            Hominy Grits] = 0.0
 [        Lamb Chops (Rib)] = 0.0
 [                   Lard] = 0.0
 [             Leg of Lamb] = 0.0
 [                 Lemons] = 0.0
 [                Lettuce] = 0.0
 [       Lima Beans, Dried] = 0.0
 [            Liver (Beef)] = 0.0018925572907052643
 [                Macaroni] = 0.0
 [              Mayonnaise] = 0.0
 [                   Milk] = 0.0
 [                Molasses] = 0.0
 [       Navy Beans, Dried] = 0.061028563526693246
 [           Oleomargarine] = 0.0
 [                 Onions] = 0.0
 [                Oranges] = 0.0
 [           Peaches (can)] = 0.0
 [          Peaches, Dried] = 0.0
 [           Peanut Butter] = 0.0
 [             Pears (can)] = 0.0
 [              Peas (can)] = 0.0
 [             Peas, Dried] = 0.0
 [         Pineapple (can)] = 0.0
 [                  Plate] = 0.0
 [              Pork Chops] = 0.0
 [         Pork Loin Roast] = 0.0
 [    Pork and Beans (can)] = 0.0
 [                Potatoes] = 0.0
 [              Pound Cake] = 0.0
 [           Prunes, Dried] = 0.0
 [          Raisins, Dried] = 0.0
 [                Rib Roast] = 0.0
 [                    Rice] = 0.0
 [        Roasting Chicken] = 0.0
 [             Rolled Oats] = 0.0
 [             Round Steak] = 0.0
 [               Rye Bread] = 0.0
 [      Salmon, Pink (can)] = 0.0
 [                Salt Pork] = 0.0
 [           Sirloin Steak] = 0.0
 [           Soda Crackers] = 0.0
 [                 Spinach] = 0.005007660466725203
 [    Strawberry Preserves] = 0.0
 [                   Sugar] = 0.0
 [          Sweet Potatoes] = 0.0
 [                     Tea] = 0.0
 [       Tomato Soup (can)] = 0.0
 [          Tomatoes (can)] = 0.0
 [            Veal Cutlets] = 0.0
[Wheat Cereal (Enriched)] = 0.0
[ Wheat Flour (Enriched)] = 0.02951906167648827
[ White Bread (Enriched)] = 0.0
[       Whole Wheat Bread] = 0.0
```

Solving the problem involves coming up with the list of amounts of each food types in the diet. For each nutrient, the amount from each food portion is summed up and the sum must meet the RDA. The model minimizes the total annual cost of the daily diet.
The cheapest diet founds costs $39.66173154546625 annually, less than Stigler's $39.93. This diet consists entirely in cabbages, beef liver, dried navy beans, spinach and enriched wheat flour.

In [10]:
```
using NamedArrays      # make sure you run Pkg.add("NamedArrays") first!

# import Stigler's data set
raw = readdlm("stigler.csv",',');
(m,n) = size(raw)

n_nutrients = 2:n      # columns containing nutrients
n_foods = 3:m          # rows containing food names

nutrients = raw[1,n_nutrients][:]   # the list of nutrients (convert to 1-D array)
foods = raw[n_foods,1][:]           # the list of foods (convert to 1-D array)

# lower[i] is the minimum daily requirement of nutrient i.
lower = Dict( zip(nutrients,raw[2,n_nutrients]) )

# data[f,i] is the amount of nutrient i contained in food f.
data = NamedArray( raw[n_foods,n_nutrients], (foods,nutrients), ("foods","nutrients") );

#Foods that are not vegan or gluten-free
bannedFoods = []
#Add banned foods to the list
for f in 1:4
    push!(bannedFoods, foods[f])
end

for f in 9:40
    push!(bannedFoods, foods[f])
end

#----Model----#
using JuMP
m = Model()

#Amount of daily food types in the diet
@defVar(m, diet[foods] >= 0)

#Total amount of daily nutrient n in the diet
@defExpr(totNutrient[n in nutrients], sum{data[f, n]*diet[f], f in foods})

#Meet RDA
@addConstraint(m, meetRDA[n in nutrients], totNutrient[n] >= lower[n])
#No non-gluten-free and no-non-vegan food in the diet
@addConstraint(m, glutenFreeVeganOnly[b in bannedFoods], diet[b] == 0)

#Minimize annual diet cost
@setObjective(m, Min, sum{diet[f]*1*365, f in foods})

solve(m)
println("Annual vegan and gluten-free diet cost: ", getObjectiveValue(m), " dollars")
println()
println("Daily diet composition (in dollar unit)")
println(getValue(diet))
```

```
Annual vegan and gluten-free diet cost: 45.58854783427841 dollars

Daily diet composition (in dollar unit)
diet: 1 dimensions, 77 entries:
 [                 Apples] = 0.0
 [         Asparagus (can)] = 0.0
 [                  Bacon] = 0.0
 [                Bananas] = 0.0
 [                 Butter] = 0.0
 [                Cabbage] = 0.011313245088275924
 [                Carrots] = 0.0
 [                 Celery] = 0.0
 [        Cheese (Cheddar)] = 0.0
 [               Chocolate] = 0.0
 [             Chuck Roast] = 0.0
 [                  Cocoa] = 0.0
 [                 Coffee] = 0.0
 [              Corn (can)] = 0.0
 [             Corn Flakes] = 0.0
 [               Corn Meal] = 0.005344246335991793
 [              Corn Syrup] = 0.0
 [                   Cream] = 0.0
 [                 Crisco] = 0.0
 [                   Eggs] = 0.0
 [  Evaporated Milk (can)] = 0.0
 [             Green Beans] = 0.0
 [       Green Beans (can)] = 0.0
 [             Ham, smoked] = 0.0
 [            Hominy Grits] = 0.0
 [        Lamb Chops (Rib)] = 0.0
 [                    Lard] = 0.0
 [             Leg of Lamb] = 0.0
 [                 Lemons] = 0.0
 [                Lettuce] = 0.0
 [      Lima Beans, Dried] = 0.0
 [           Liver (Beef)] = 0.0
 [               Macaroni] = 0.0
 [             Mayonnaise] = 0.0
 [                  Milk] = 0.0
 [               Molasses] = 0.0
 [      Navy Beans, Dried] = 0.10306689112726253
 [          Oleomargarine] = 0.0
 [                Onions] = 0.0
 [               Oranges] = 0.0
 [          Peaches (can)] = 0.0
 [         Peaches, Dried] = 0.0
 [          Peanut Butter] = 0.0
 [            Pears (can)] = 0.0
 [             Peas (can)] = 0.0
 [            Peas, Dried] = 0.0
 [        Pineapple (can)] = 0.0
 [                  Plate] = 0.0
 [             Pork Chops] = 0.0
 [        Pork Loin Roast] = 0.0
 [   Pork and Beans (can)] = 0.0
 [               Potatoes] = 0.0
 [             Pound Cake] = 0.0
 [          Prunes, Dried] = 0.0
 [         Raisins, Dried] = 0.0
 [              Rib Roast] = 0.0
 [                   Rice] = 0.0
 [       Roasting Chicken] = 0.0
 [             Rolled Oats] = 0.0
 [             Round Steak] = 0.0
 [               Rye Bread] = 0.0
 [     Salmon, Pink (can)] = 0.0
 [              Salt Pork] = 0.0
 [          Sirloin Steak] = 0.0
 [          Soda Crackers] = 0.0
 [                Spinach] = 0.005175748501287311
 [   Strawberry Preserves] = 0.0
 [                  Sugar] = 0.0
 [         Sweet Potatoes] = 0.0
 [                    Tea] = 0.0
 [      Tomato Soup (can)] = 0.0
 [         Tomatoes (can)] = 0.0
 [           Veal Cutlets] = 0.0
[Wheat Cereal (Enriched)] = 0.0
[ Wheat Flour (Enriched)] = 0.0
[ White Bread (Enriched)] = 0.0
[      Whole Wheat Bread] = 0.0
```

This model is similar to the model in 1(a), but with an additional list of non-vegan and non gluten-free foods. A constraint is added such that all foods that are in the list amount to zero in the diet list.
This diet costs $45.58854783427841 annually. It consists only of cabbages, corn meal, dried navy beans, and spinach.

In [1]:
```
using NamedArrays      # make sure you run Pkg.add("NamedArrays") first!

# import Stigler's data set
raw = readdlm("stigler.csv",',');
(m,n) = size(raw)

n_nutrients = 2:n      # columns containing nutrients
n_foods = 3:m          # rows containing food names

nutrients = raw[1,n_nutrients][:]   # the list of nutrients (convert to 1-D array)
foods = raw[n_foods,1][:]           # the list of foods (convert to 1-D array)

# lower[i] is the minimum daily requirement of nutrient i.
lower = Dict( zip(nutrients,raw[2,n_nutrients]) )

# data[f,i] is the amount of nutrient i contained in food f.
data = NamedArray( raw[n_foods,n_nutrients], (foods,nutrients), ("foods","nutrients") );


#----Dual Model----#
using JuMP
m = Model()

#Lambdas corresponding to RDA constraints
@defVar(m, lambda[nutrients] >= 0)

#Dual Constraints
@addConstraint(m, dualConst[f in foods], sum{data[f, n]*lambda[n], n in nutrients} <= 365)

#Maximize dot product of RDA and lambda matrices
@setObjective(m, Max, sum{lower[n]*lambda[n], n in nutrients})

solve(m)
#Display results
println("Lambda values:")
println()
println(getValue(lambda))
```

```
Lambda values:

lambda: 1 dimensions, 9 entries:
 [ Ascorbic Acid (mg)] = 0.052602893142533885
 [        Calcium (g)] = 11.5842654076576
 [    Calories (1000)] = 3.1992787637880906
 [          Iron (mg)] = 0.0
 [        Niacin (mg)] = 0.0
 [        Protein (g)] = 0.0
 [    Riboflavin (mg)] = 5.970681935235992
 [      Thiamine (mg)] = 0.0
 [Vitamin A (1000 IU)] = 0.1460849434297644
```

To find the maximum price one is willing to pay for a calcium supplement pill the dual of the original problem is used. The lambda value associated with the calcium intake constraint is 11.5842654076576. Because calcium is measure in grams, the maximum price for a calcium supplement pill containing 500 mg is $5.7921327038288 (11.5842654076576 / 2).

In [2]:
```
#---Data---#
#Months
month = [1, 2, 3, 4]

#Projects
project = [:p1, :p2, :p3]

#Deadlines(months from start time)
deadline = Dict(:p1 => 3, :p2 => 4,:p3 => 2)

#Labor requirement of projects
laborRequirement = Dict(:p1 => 8, :p2 =>10, :p3 => 12)

#Total worker available per month
laborAvailable = 8

#Number of workers allowed on project in a given month
laborLimit = 6

# ---Model---#
using JuMP
m = Model()

#Labor used for a project in a month
@defVar(m, 0 <= laborUsed[project, month] <= laborLimit)

#Total labor used in a month
@defExpr(monthlyTotal[mn in month], sum{laborUsed[p, mn], p in project})

#Limit on total labor used monthly
@addConstraint(m, monthlyLimit[mn in month], monthlyTotal[mn] <= laborAvailable)
#Each project must get enough labor for completion
@addConstraint(m, completion[p in project], sum{laborUsed[p, mn], mn in month}>= laborRequirement[p]
)
#Work must be done by the deadlines
@addConstraint(m, meetDeadline[p in project], sum{laborUsed[p, i], i in (deadline[p] + 1):length(mon
th)} == 0)

#Minimize total labor used
@setObjective(m, Min, sum(monthlyTotal))

#Display result
solve(m)
println("[project, month]")
println(getValue(laborUsed))
```

```
[project, month]
laborUsed: 2 dimensions, 12 entries:
 [p1,1] = 0.0
 [p1,2] = 2.0
 [p1,3] = 6.0
 [p1,4] = 0.0
 [p2,1] = 2.0
 [p2,2] = 0.0
 [p2,3] = 2.0
 [p2,4] = 6.0
 [p3,1] = 6.0
 [p3,2] = 6.0
 [p3,3] = 0.0
 [p3,4] = 0.0
```

The problem is solved using a two dimensional array showing the labor used for each task (row) for each week(column). The deadlines of tasks are represented as stipulations that labor used for certain rows for the task must be zero. Task completion means that the total labor used in a column must add to the task's completion requirement. The limit of eight workers per month means that labor total of each row is limited to eight.

The result shows that each project received all the labor it requires by its respective deadline, while adhering to all labor constraints.

In [2]:
```
#---Data---#
#Employees
employee = [:Manuel, :Luca, :Jule, :Michael, :Malte, :Chris, :Spyros,
            :Mirjam, :Matt, :Florian, :Josep, :Joel, :Tom, :Daniel, :Anne]

#Interview time slots
time = [1000, 1020, 1040, 1100, 1120, 1140, 1200, 1300, 1320, 1340, 1400, 1420, 1440]

#Schedule of availability(row: employees, column: time)
availability =
           [0 0 1 1 0 0 0 1 1 0 0 0 0;
            0 1 1 0 0 0 0 0 1 1 0 0 0;
            0 0 0 1 1 0 1 1 0 1 1 1 1;
            0 0 0 1 1 1 1 1 1 1 1 1 0;
            0 0 0 0 0 0 1 1 1 0 0 0 0;
            0 1 1 0 0 0 0 0 1 1 0 0 0;
            0 0 0 1 1 1 1 0 0 0 0 0 0;
            1 1 0 0 0 0 0 0 0 0 1 1 1;
            1 1 1 0 0 0 0 0 0 1 1 0 0;
            0 0 0 0 0 0 0 0 1 1 0 0 0;
            0 0 0 0 0 0 0 1 1 1 0 0 0;
            1 1 0 0 0 1 1 1 1 0 0 1 1;
            1 1 1 0 1 1 0 0 0 0 0 1 1;
            0 1 1 1 0 0 0 0 0 0 0 0 0;
            1 1 0 0 1 1 0 0 0 0 0 0 0;]

#Number of employees that must meet for lunch
lunchEmployees = 3

#Column index of lunch hour
lunchIndex = 7

#---Model---#
using JuMP
m = Model()

#The interview schedule
@defVar(m, schedule[1:length(employee), 1:length(time)] >= 0)

#There must be enough employees for lunch
@addConstraint(m, lunchNumber, sum{schedule[e, lunchIndex], e in 1:length(employee)} == lunchEmployees)

#There must be only one employee scheduled at each interview slot (before and after lunch hour)
@addConstraint(m, oneEmployee1[t in 1:(lunchIndex-1)], sum{schedule[e,t], e in 1:length(employee)} == 1)
@addConstraint(m, oneEmployee2[t in 8:length(time)], sum{schedule[e,t], e in 1:length(employee)} == 1)

#Only assign on available times
@addConstraint(m, onlyAvailable[e in 1:length(employee), t in 1:length(time)],
availability[e, t] - schedule[e, t] >= 0)

#Assign each worker at least once
@addConstraint(m, onlyOnce[e in 1:length(employee)], sum{schedule[e, t], t in 1:length(time)} >= 1)

#Schedule by mimimizing assignments
@setObjective(m, Min, sum(schedule))

solve(m)
#Display schedule
println("Employees to meet: ")
for t in 1:length(time)
    print(time[t], ": ")

    for e in 1:length(employee)
        if(getValue(schedule[e,t]) == 1)
            print(employee[e], " ")
        end
    end
    println()
end
```

```
Employees to meet:
1000: Anne
1020: Chris
1040: Manuel
1100: Daniel
1120: Spyros
1140: Tom
1200: Malte Josep Joel
1300: Florian
1320: Michael
1340: Luca
1400: Matt
1420: Mirjam
1440: Jule
```

The solution involves two 2-D arrays (rows for employees, columns for time slots), the availability schedule and the interview schedule. Both use 1 and 0 to represent availability and unavailability respectively. Solving the problem means filling up the interview schedule array with 1's and 0's. There can only be a 1 on the interview array if there is a 1 on the availability array on the same index. The total on the lunchtime column must add to up to 3. That there must only be one interviewer each time means the total of interview time columns must be 1. That each employee must meet the candidate at least once means the total in rows must be at least one.

In this schedule, all employees will meet the candidate at times they are available, and there will be three employees meeting the candidate for lunch at 12 pm. Thus a feasible interview schedule exists given the constraints.

In [1]:
```
#---Data---#
#Agencies
agency = collect(1:10)

#Position of agencies, (x, y) coordinates
position = [0 0; 20 20; 18 10; 30 12; 35 0; 33 25; 5 27; 5 10; 11 0; 2 15]

#Car requirement of agencies
carsNeeded = [10, 6, 8, 11, 9, 7, 15, 7, 9, 12]

#Cars present currently
carsInitially = [8, 13, 4, 8, 12, 2, 14, 11, 15, 7]

#Transportation cost per car per mile
mileCost = 0.50

#Ratio of road distance to Euclidean distance
ratio = 1.3


#---Model---#
using JuMP
model = Model()

#Flow of cars: [source, target, amount]
@defVar(model, flow[agency, agency] >= 0)

#Cost of transporting a car between two agencies
@defExpr(cost[s in agency, r in agency],  sqrt( (position[s, 1] - position[r, 1])^2
+ (position[s, 2] - position[r, 2])^2)*ratio*mileCost*flow[s,r])

#Inflows to an agency
@defExpr(inflow[r in agency], sum{flow[s, r], s in agency})

#Outflows out of an agency
@defExpr(outflow[s in agency], sum{flow[s, r], r in agency})

#Cars present at an agency
@defExpr(carsNow[a in agency], carsInitially[a] + inflow[a] - outflow[a])

#Agencies must get the cars they need
@addConstraint(model, meetNeed[a in agency], carsNow[a] >= carsNeeded[a])

#Constraint on outflow of cars from an agency
@addConstraint(model, supply[a in agency], outflow[a] <= carsInitially[a])

#Minimize transportation cost
@setObjective(model, Min, sum(cost))

solve(model)

#Display non-zero flows
println("Flow of cars: ")
for s in agency
    for r in agency
        if(getValue(flow[s,r]) > 0)
            println("Agency $s to agency $r: ", getValue(flow[s,r]))
        end
    end
end
```

```
Flow of cars:
Agency 2 to agency 3: 1.0
Agency 2 to agency 6: 5.0
Agency 2 to agency 7: 1.0
Agency 5 to agency 4: 3.0
Agency 8 to agency 10: 5.0
Agency 9 to agency 1: 2.0
Agency 9 to agency 3: 3.0
Agency 9 to agency 8: 1.0
```

The solution involves a 2 dimensional matrix (agencies by agencies), that stores the number of cars moved from the row agencies to the column agencies. The model keeps track of cars present now at agencies. This is the original amount plus car inflows minus car outflows. This must meet the amount required by each agency. The outflow from an agency a is the total on ath row, while the inflow into agency a is the total on the ath column.
The model minimizes total transport cost, which is the total of the flows multiplied by flow distance, 1.3 (ratio between Euclidean to road distance) and $0.50 (cost per mile).

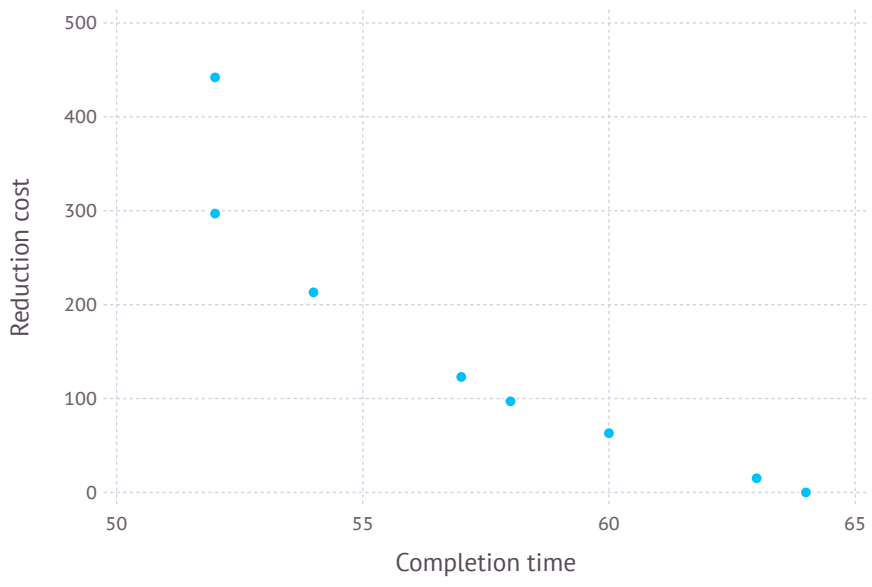
In [1]:
```
#Tasks
task = [:t1, :t2, :t3, :t4, :t5, :t6, :t7, :t8,
        :t9, :t10, :t11, :t12, :t13, :t14, :t15, :t16, :t17, :t18]

#Project time spans
sp = [2 16 9 8 10 6 2 2 9 5 3 2 1 7 4 3 9 1]
span = Dict(zip(task,sp))

#Task predecessors
pr = ( [], [:t1], [:t2], [:t2], [:t3], [:t4, :t5], [:t4], [:t6], [:t4, :t6], [:t4], [:t6],
       [:t9],[:t7], [:t2], [:t4, :t14], [:t8, :t11, :t14], [:t12], [:t17])
pred = Dict(zip(task,pr))

#Maximum task week reduction
mR = [0 3 1 2 2 1 1 0 2 1 1 0 0 2 2 1 3 0]
maxRed = Dict(zip(task, mR))

#Reduction cost per week
rC = [0 30 26 12 17 15 8 0 42 21 18 0 0 22 12 6 16 0]
redCost = Dict(zip(task, rC))


using JuMP
m = Model()

#Start time of tasks
@defVar(m, startTime[task] >= 0 )

#Predecessors must be completed before each task
@addConstraint(m, completePred[t in task,p in pred[t]], startTime[t] >= startTime[p] + span[p])

#Minimimize completion time of final task
@setObjective(m, Min, startTime[task[end]] + span[task[end]])

solve(m)
#Display completion time
println("Completion time: ",getObjectiveValue(m), " weeks")
```

```
Completion time: 64.0 weeks
```

The solution uses a list of the start time of tasks. The values are chosen such that for a particular task, its start time must be greater than the start time of its predecessor task and the predecessor's time span. The optimization involves minimizing the end time of the final task.

In [43]:
```
using JuMP
#---Data---#
#Tasks
task = [:t1, :t2, :t3, :t4, :t5, :t6, :t7, :t8, :t9,
        :t10, :t11, :t12, :t13, :t14, :t15, :t16, :t17, :t18]

#Project time spans
sp = [2 16 9 8 10 6 2 2 9 5 3 2 1 7 4 3 9 1]
span = Dict(zip(task,sp))

#Task predecessors
pr = ( [], [:t1], [:t2], [:t2], [:t3], [:t4, :t5], [:t4], [:t6], [:t4, :t6], [:t4], [:t6],
       [:t9],[:t7], [:t2], [:t4, :t14], [:t8, :t11, :t14], [:t12], [:t17])
pred = Dict(zip(task,pr));

#Maximum task week reduction
mR = [0 3 1 2 2 1 1 0 2 1 1 0 0 2 2 1 3 0]
maxRed = Dict(zip(task, mR))

#Reduction cost per week
rC = [0 30 26 12 17 15 8 0 42 21 18 0 0 22 12 6 16 0]
redCost = Dict(zip(task, rC))

#---Model---#

function stadiumBuilding(tradeoff)   #tradeoff is the tradeoff parameter weighted on the reduction cost
    m = Model()

    #Start time of tasks
    @defVar(m, startTime[task] >= 0)
    #Actual span of tasks with reductions
    @defVar(m, actualSpan[task] >= 0)

    #Total reduction cost
    @defExpr(totRedCost, sum{(span[t] - actualSpan[t])*redCost[t], t in task})
    #Total construction time
    @defExpr(totTime,startTime[task[end]] + actualSpan[task[end]])

    #Predecessors must be completed before each task
    @addConstraint(m, completePred[t in task,p in pred[t]], startTime[t] >= startTime[p] + actualSpan[p])
    #Constraints on task reduction
    @addConstraint(m, redCon[t in task], span[t] - actualSpan[t] <= maxRed[t])
    @addConstraint(m, redCon2[t in task], span[t] >= actualSpan[t])

    #Minimimize reduction cost and completion time
    @setObjective(m, Min, totTime + tradeoff*totRedCost)

    solve(m)
    return(getValue(totTime), getValue(totRedCost))
end

#Plot trade-off curve
tr = 0          #Tradeoff parameter for the model
j1 = []         #Stores completion times
j2 = []         #Stores reduction cost
totRedCost

#Load up values
for i in 1:70
    result = stadiumBuilding(tr)
    push!(j1, result[1])
    push!(j2, result[2])
    tr = tr + 0.001
end

#Plot tradeoff
using Gadfly
plot(x = j1, y = j2, Guide.xlabel("Completion time"), Guide.ylabel("Reduction cost"))
```

Out[43]:



The model is modified from the one in 5(a) in that now the optimization involves minimizing the sum of completion time and reduction cost, the latter of which is weighted with a tradeoff parameter. Because now tasks can be expedited a distinction is made between original time spans and actual time spans of tasks. The difference between the two figures into the reduction cost. The model is now within a function. The tradeoff parameter is varied, and the resulting completion times and reduction costs are stored and plotted against each other.

In [1]:
```
#---Data---#
#Tasks
task = [:t1, :t2, :t3, :t4, :t5, :t6, :t7, :t8, :t9,
        :t10, :t11, :t12, :t13, :t14, :t15, :t16, :t17, :t18]

#Project time spans
sp = [2 16 9 8 10 6 2 2 9 5 3 2 1 7 4 3 9 1]
span = Dict(zip(task,sp))

#Task predecessors
pr = ( [], [:t1], [:t2], [:t2], [:t3], [:t4, :t5], [:t4], [:t6], [:t4, :t6], [:t4], [:t6],
       [:t9],[:t7], [:t2], [:t4, :t14], [:t8, :t11, :t14], [:t12], [:t17])
pred = Dict(zip(task,pr))

#Maximum task week reduction
mR = [0 3 1 2 2 1 1 0 2 1 1 0 0 2 2 1 3 0]
maxRed = Dict(zip(task, mR))

#Reduction cost per week
rC = [0 30 26 12 17 15 8 0 42 21 18 0 0 22 12 6 16 0]
redCost = Dict(zip(task, rC))


#---Model without task reduction---#
using JuMP
m = Model()

#Start time of tasks

@defVar(m, startTime[task] >= 0 )
#Predecessors must be completed before each task
@addConstraint(m, completePred[t in task,p in pred[t]], startTime[t] >= startTime[p] + span[p])

#Minimimize completion time of final task
@setObjective(m, Min, startTime[task[end]] + span[task[end]])

#Display completion time
solve(m)
println("Original completion time: ",getObjectiveValue(m), " weeks")

#---Model with task reduction----#
m2 = Model()

incentive = 30000                          #Incentive per week early

@defVar(m2, startTime[task] >= 0 )         #Start time of tasks
#Actual span of tasks with reductions
@defVar(m2, actualSpan[task] >= 0)

#Total incentive received
@defExpr(totalIncentive, (getObjectiveValue(m) - (startTime[task[end]] + actualSpan[task[end]]))*inc
entive)
#Total reduction cost
@defExpr(totalRedCost, sum{(span[t] - actualSpan[t])*redCost[t], t in task})

#Predecessors must be completed before each task
@addConstraint(m2, completePred[t in task,p in pred[t]], startTime[t] >= startTime[p] + actualSpan[p
])
#Constraints on task reduction
@addConstraint(m2, redCon[t in task], span[t] - actualSpan[t] <= maxRed[t])
@addConstraint(m2, redCon2[t in task], span[t] >= actualSpan[t])

@setObjective(m2, Max, totalIncentive - totalRedCost)    #Maximize profit

solve(m2)
#Display completion time
println("Completion time with task reduction: ",
getValue(startTime[task[end]]) + getValue(actualSpan[task[end]]), " weeks")
```

```
Original completion time: 64.0 weeks
Completion time with task reduction: 52.0 weeks
```

The model is modified by adding factors of revenue and cost. Also added was a variable for actual time span of tasks which includes original time spans and time reductions taken. The total incentive received is the bonus multiplied by the difference between
the end date of the original model and the end date of this model. The cost is the difference between original and actual time spans multiplied by the per week reduction cost for that task. The total profit is the incentive received minus the total cost incurred from speeding up tasks. The optimization involves maximizing profit.