Counting Calories with Symbolic Computing

Brian Beckman, Erik Meijer

13 April 2012

INTRODUCTION

We show plausible application code -- counting calories -- that a developer might write using the Merino platform. We show how *symbolic computing* with Jacquard makes the code *shorter*, more *flexible*, more *robust*; easier to *create*, to *understand*, to *modify*, and to *reuse* than the equivalent computation written in ordinary JavaScript.

In fact, the symbolic computation enables trivial automatic units conversions that let our developer easily catch a misleading bit of consumer information. Such units conversions are not possible in JavaScript without implementing a symbolic computing facility essentially equivalent to Jacquard's fundamental method of *term rewriting*.

SCENARIO

Consider the following nutrition-information block, which purports to record, for a hamburger patty, the nutritional component breakdown (fat / protein / carbs) and the calorie proportions for each component.

Our developer, let's call her Alice, wants to write some analytics over this data to find out if it's accurate and / or misleading.

- Does the calorie total in the breakdown match the calorie count given in the "amount per serving?"
- Does the total weight implied in the breakdown match the total weight in a "serving size"?
- Do the calories implied by the proportions in the breakdown, applied to the serving size, match the calorie "amount per serving?"

Let's show what Alice might write in JavaScript, then show a Jacquard computation that produces the same result, and finally compare and contrast.

Nutrition Facts

Serving Size: 4 oz

Amount per Serving	
Calories 160	Calories from Fat 81.0
	% Daily Value *
Total Fat 9g	13%
Saturated Fat 4g	20%
Cholesterol 60mg	20%
Sodium 70mg	2%
Total Carbohydrate 0g	0%
Dietary Fiber	0%
Sugars	
Protein 21g	42%
	Est. Percent of Calories from:
Fat	49.1%
Carbs	%
Protein	50.9%

* Percent Daily Values are based on a 2,000 calorie diet. Your daily values may be higher or lower depending on your calories needs.

The first step is to encode the data in a JavaScript object. Alice is careful to keep the units of measure in comments for mental tracking and for informing other developers of this secret info:

```
var burgerNutritionFacts001 =
{ 'Serving Size'        :   4   /* ounce */,
  'Amount per Serving' : 160   /* calorie */,
  'Calories from Fat'  :  81.0 /* calorie */,
  'Saturated Fat'      :   4   /* gram */,
  'Cholesterol'        :  60   /* milligram */,
  'Sodium'             :  70   /* milligram */,
  'Dietary Fiber'      :   0   /* gram */,
  'Sugars'             :   0   /* gram */,
  'Total Fat'          :   9   /* gram */,
  'Protein'            :  21   /* gram */,
  'Total Carbohydrate' :   0   /* gram */
};
```

Alice chooses to preserve the spaces in the object keys such as “Serving Size” and “Total Fat” for direct correspondence to the source data. She accepts that she can’t later use dot notation with such keys. *burgerNutritionFacts001.Protein* will work because the key *Protein* does not contain any spaces, and *burgerNutritionFacts001[‘Protein’]* is completely equivalent. However, because of the space character in the key ‘Total Fat’, the dot notation won’t work and the only way to access the value of that property is *burgerNutritionFacts001[‘Total Fat’]*. For uniformity of style, Alice will use only the square-bracket *indexer* notation everywhere.

The next step is to add up the weights in grams of all the nutritional components. Alice knows that milligrams are tiny by comparison to grams, so she simply sidesteps them in this first version. She also takes the prudent preventative of packaging the computation in a function, parameterized by the data object, for reuse on other nutrition blocks.

```
var addWeights001 = function(nutritionFacts) {
  return nutritionFacts['Total Fat'] +
```

```
          nutritionFacts['Dietary Fiber'] +
          nutritionFacts['Protein'] +
          nutritionFacts['Total Carbohydrate'];
};

document.writeln(addWeights001(burgerNutritionFacts001));
```

The result is 30.

ALICE'S DREAMS

Before continuing with her planned analysis, Alice reflects a little on the code she just wrote. Will it withstand criticism from other programmers? Is the same idea useful again in new functions?

Repetition Considered Harmful

DRY (Don't Repeat Yourself) is programmer mantra nowadays. Unecessary repetition in code just increases risk of programmer error.

Even though Alice is careful to align parts of the arithmetic expression, the repeated name of the object in every term in the sum is striking. This seems to be gratuitous, useless repetition. If there were hundreds of properties, it would be oppressive. Can she get rid of the repetition? Suppose she could write

```
var addWeights001 = function(nutritionFacts) {
  return nutritionFacts[
    'Total Fat' + 'Dietary Fiber' + 'Protein' + 'Total Carbohydrate"];
};
```

And *why not* have expressions inside the square brackets? The meaning is completely obvious. But she can't make it work, and for multiple reasons.

Alice's First Dream

Generally, it's a benefit to have an infix operator for concatenating strings. Unfortunately, JavaScript overloaded the + operator instead of introducing a new one, and the expression

```
nutritionFacts[
    'Total Fat' + 'Dietary Fiber' + 'Protein' + 'Total Carbohydrate']
```

evaluates to

```
nutritionFacts['Total FatDietary FiberProteinTotal Carbohydrate'] ~~>
undefined
```

Her dream code is **syntactically legal, and completely wrong**. She can't write this.

Alice Regrets Whitespace

Alice continues to dream. If she hadn't preserved space characters in the keys, but gone to camelBack, she could have written

```
var burgerNutritionFacts002 =
{ ServingSize        :   4   /* ounce */,
```

```
  AmountPerServing   : 160   /* calorie */,
  CaloriesFromFat    :  81.0 /* calorie */,
  SaturatedFat       :   4   /* gram */,
  Cholesterol        :  60   /* milligram */,
  Sodium             :  70   /* milligram */,
  DietaryFiber       :   0   /* gram */,
  Sugars             :   0   /* gram */,
  TotalFat           :   9   /* gram */,
  Protein            :  21   /* gram */,
  TotalCarbohydrate  :   0   /* gram */
};

var addWeights002 = function(nutritionFacts) {
  return nutritionFacts.TotalFat +
         nutritionFacts.DietaryFiber +
         nutritionFacts.Protein +
         nutritionFacts.TotalCarbohydrate;
};

document.writeln(addWeights002(burgerNutritionFacts002));
```

The result is still 30.

There is a cost in going to camelBack, however. There is more code to develop, test, build, manage, deploy, and maintain -- code that creates objects from strings retrieved from the internet, which must build camelBack symbols like “CaloriesFromFat” from standardized strings like “Calories from Fat.” Removing spaces isn’t enough: the internal word “from” must be capitalized. Her original decision to use string keys instead of camelBack completely avoided all that cost, but maybe it will be worth it if there is a way to get rid of the repetition.

Alice’s Second Dream

So she imagines what she could write with the camelBack conversion. Her dream code would be:

```
var addWeights002 = function (nutritionFacts) {
  return
    nutritionFacts.(TotalFat + DietaryFiber + Protein + TotalCarbohydrate);
};
```

Now that’s sweet, but not syntactically legal ... unless ... the “with” statement!

```
var addWeights002 = function (nutritionFacts) {
  with(nutritionFacts) {
    return TotalFat + Protein + DietaryFiber + TotalCarbohydrate;
  }
};

document.writeln(addWeights002(burgerNutritionFacts002));
```

The result is still 30.

Alas, this is not an acceptable solution. It has removed *any* reference to the object from the expression in the return statement, and thus rendered the expression fundamentally ambiguous. The value of the term *TotalFat* could come from the global environment, or from an outer nesting of local environments, or from outer *with* statements. There is no way to tell from local inspection of the expresion. This is too much exposure to scoping errors.

There has been much pain in the JavaScript community about the *with* statement. The ambiguity it injects outweights its superficial attractiveness. The overall concensus is to avoid it.

"with" Statement Considered Harmful

See this article by Douglas Crockford, a recognized authority in the JavaScript community, for more about why the "with" statement is not acceptable in JavaScript.

EXPRESSIONS MUST STAND ALONE

Can we remove any reference to the target object from the expression and NOT introduce ambiguity? It seems the only way would be to treat the expression itself as a standalone, first-class object, reserving ANY interpretation of the expression until it's used. But that's what symbolic computing means! We just have to figure out how to apply the data to the expression or *vice versa* to get the needed interpretation in the right place and time.

There is no native way in JavaScript to do this. The interpretation of all symbols must be known prior to run time. A symbol that does not have a value generates a deep exception -- it's an unrecoverable error -- an invalid program. In C#, unbound symbols don't even compile.

In term-rewriting, expressions that don't evaluate to something else just evaluate to themselves. A symbol without a binding to a value is NOT an error, it's just the symbol itself as a first-class, atomic value. To interpret an expression that does not reduce further -- an expression in so-called normal form -- we combine it with other expressions that rewrite parts or all of it. That's why such systems are called term-rewriting systems.

Rewriting Alice's Dreams

First, we rewrite Alice's Second Dream: the version using camelBack symbols for keys. Later, we show the exact same code using string keys with internal whitespace, allowing Alice to get rid of her string-to-symbol conversion code, restoring her First Dream.

In the offing, we sneak in units of measure, and conclude by showing what might be needed in ordinary JavaScript to include such a facility.

Take the nutrition data and write them as rules. A rule tells the evaluator "if you can match the left-hand side, please replace it with the right-hand side." A rule is exactly equivalent to a JavaScript or JSON property, *i.e.*, key-value pair. In fact, Jacquard & *Mathematica* serialize JSON objects in and out as a list of rules.

Rules map property keys (string or symbol) to values (arbitrary expressions). The fact that we can always represent objects as rules reveals the deeper fact that objects are just functions from keys to values, a fact that is also obvious from noting that objects are implemented as hash tables or red-black trees, other representations for functions from keys to values.

Notice in the following that we do not comment out the units of measure: they're just symbolic constants in normal form.

We use a pretty-print function from the Jacquard library to grid out this definition:

In[2]:=
```
(burgerNutritionFacts = {
    ServingSize → 4 ounce,
    AmountPerServing → 160 calorie,
    CaloriesFromFat → 81.0 calorie,
    SaturatedFat → 4 gram,
    Cholesterol → 60 milli gram,
    Sodium → 70 milli gram,
    DietaryFiber → 0 gram,
    Sugars → 0 gram,
    TotalFat → 9 gram,
    Protein → 21 gram,
    TotalCarbohydrate → 0 gram}) // gridRules
```

Out[2]=

ServingSize	4 ounce
AmountPerServing	160 calorie
CaloriesFromFat	81. calorie
SaturatedFat	4 gram
Cholesterol	60 gram milli
Sodium	70 gram milli
DietaryFiber	0
Sugars	0
TotalFat	9 gram
Protein	21 gram
TotalCarbohydrate	0

Now consider the following expression. Notice that it just evaluates to itself, after being reordered into "canonical order." This happens to be alphabetical order in this case. Canonical order helps to test structural equality of expressions, needed for pattern matching, so Jacquard and *Mathematica* always do it by default. There is no difficulty here, since the order of terms in a sum does not matter.

In[3]:=
```
TotalFat + DietaryFiber + Protein + TotalCarbohydrate
```

Out[3]=
```
DietaryFiber + Protein + TotalCarbohydrate + TotalFat
```

What can we do with this expression? Apply the data to it, which we do with the ReplaceAll operation:

In[4]:=
```
ReplaceAll[TotalFat + DietaryFiber + Protein + TotalCarbohydrate, burgerNutritionFacts]
```

Out[4]=
```
30 gram
```

This means "30 times grams;" it's a multiplication expression. Notice that *gram* gets carried along as a "dead" symbolic constant -- one in normal form, for which no further interpretation is available or desired. Not easy in JavaScript, but very valuable. It's the kind of thing that would have saved a couple of billion dollars in the 1999 crash of the Mars Climate Observer.

- quote

  Specifically, the flight system software on the Mars Climate Orbiter was written to calculate thruster performance using the metric unit Newtons (N), while the ground crew was entering course correction and thruster data using the Imperial measure Pound-force (lbf). This error has since been known as the metric mixup and has been carefully avoided in all missions since by NASA.

- end quote

We can do some things to shorten this. First, we can use the shorthand infix operator, “/.”, instead of the direct call to ReplaceAll:

In[5]:=
```
TotalFat + DietaryFiber + Protein + TotalCarbohydrate /. burgerNutritionFacts
```
Out[5]=
```
30 gram
```

This code is quite close to Alice’s Second Dream, except the object comes after the expression. That’s appropriate since we’re applying the object to the expression. But it’s not an important limitation since we can design our own operators to write things in the opposite order should we need. Let’s skip that for now.

The Computation is Just Another Expression

We’ve alread seen that

In[6]:=
```
TotalFat + DietaryFiber + Protein + TotalCarbohydrate
```
Out[6]=
```
DietaryFiber + Protein + TotalCarbohydrate + TotalFat
```

is an expression, just one that doesn’t have another value. But the application of rules

In[7]:=
```
TotalFat + DietaryFiber + Protein + TotalCarbohydrate /. burgerNutritionFacts
```
Out[7]=
```
30 gram
```

is also just another expression, this time with a reduced value revealed by the *burgerNutritionFacts*. If we applied a different nutrition block, say this one for a chicken breast:

Nutrition Facts

Serving Size: 4oz

Amount per Serving
Calories 130 Calories from Fat 9.0

	% Daily Value *
Total Fat 1g	1%
Saturated Fat 0.4g	2%
Cholesterol 68mg	22%
Sodium 77mg	3%
Total Carbohydrate 0g	0%
Dietary Fiber 0g	0%
Sugars 0.1g	
Protein 27g	54%

	Est. Percent of Calories from:
Fat	7.7%
Carbs	0%
Protein	92.3%

* Percent Daily Values are based on a 2,000 calorie diet. Your daily values may be higher or lower depending on your calories needs.

In[8]:=

```
(chickenNutritionFacts = {
    ServingSize → 4 ounce,
    AmountPerServing →  130 calorie,
    CaloriesFromFat → 9.0 calorie,
    SaturatedFat → 0.4 gram,
    Cholesterol → 68 milli gram,
    Sodium → 77 milli gram,
    DietaryFiber → 0 gram,
    Sugars → 0.1 gram,
    TotalFat → 1 gram,
    Protein → 27 gram,
    TotalCarbohydrate → 0 gram}) // gridRules
```

Out[8]=

ServingSize	4 ounce
AmountPerServing	130 calorie
CaloriesFromFat	9. calorie
SaturatedFat	0.4 gram
Cholesterol	68 gram milli
Sodium	77 gram milli
DietaryFiber	0
Sugars	0.1 gram
TotalFat	gram
Protein	27 gram
TotalCarbohydrate	0

In[9]:=
```
TotalFat + DietaryFiber + Protein + TotalCarbohydrate /. chickenNutritionFacts
```
Out[9]=
```
28 gram
```

we would get a different answer.

The main point here is that **the expression and the data blocks are independent**, even of any parameters. We may reuse them separately and combine them in arbitrary ways. Alice's original JavaScript code had this independence, too, but she had to package the expression in a function. She could not give the expression an independent existence from the data: the best she could do was parameterize the data, and that's when she encountered her verbosity / repetition problem.

Expressions are Values

We can save the expression itself in a variable

In[10]:=
```
totalWeight = TotalFat + DietaryFiber + Protein + TotalCarbohydrate
```
Out[10]=
```
DietaryFiber + Protein + TotalCarbohydrate + TotalFat
```

Now the symbol *totalWeight* has a value, namely the symbolic, normal-form expression we've been carrying along. Symbols can rewrite to other symbols; they need not reduce all the way to numbers. Let's symbolically add the little milligram things to it:

In[11]:=
```
totalWeight + Cholesterol + Sodium
```
Out[11]=
```
Cholesterol + DietaryFiber + Protein + Sodium + TotalCarbohydrate + TotalFat
```

And apply the data again:

In[12]:=
```
totalWeight + Cholesterol + Sodium /. burgerNutritionFacts
```
Out[12]=
```
30 gram + 130 gram milli
```

We see we have incompatible units -- and, as an aside, that *Mathematica* canonically reordered our "milli gram" to "gram milli." That's fine, since "gram milli" means "gram times milli," and "times" is independent of order like "plus."

The fundamental symbolic nature of the computation has caught this units incompatibility problem for us. In Alice's first JavaScript solution, she could only have caught this by manual inspection of the code. We could have ended up with a horribly incorrect answer -- 160 grams -- and had no way to track it down other than by manual labor.

In fact, we can look very suspiciously at this 160. Notice that the original nutrition block states that 160 is the number of calories in a serving. We find later that not only is this number off by nearly a factor of four, but that the number 160 is not implied by any other data in the nutrition block. There is only one instance of 160 to be found in this sample, and that's from this wildly mistaken computation: adding grams to milligrams. It's a strong hypothesis that the original authors of the nutrition block made this mistake.

However, with symbolic computing, we can easily write some rules to convert our units to compatible forms:

In[13]:=
```
unitsConversions = {
  milli gram → gram / 1000.0
 }
```
Out[13]=
```
{gram milli → 0.001 gram}
```

In[14]:=
```
totalWeight + Cholesterol + Sodium /.
  burgerNutritionFacts /.
 unitsConversions
```
Out[14]=
```
30.13 gram
```

We see that we can easily compose ReplaceAll invocations in chains, and now we see why canonicalizing expressions is valuable. We wrote “milli gram” but *Mathematica* saw and matched “gram milli.” Without canonicalizing, our “milli gram” would not have matched *Mathematica*’s “gram milli” and our rule would not have applied.

In fact, our *unitsConversions* rules are too restrictive. A milli anything is 1/1000 of the same thing; we don’t need the “gram” at all. Let’s simplify the *unitsConvsersions*. Giving a new value -- a new list of rules -- to the variable *unitsConversions* just replaces the old value of the variable.

In[15]:=
```
unitsConversions = {
  milli → 1 / 1000.0
 }
```
Out[15]=
```
{milli → 0.001}
```

Functional programmers call this "point-free form".

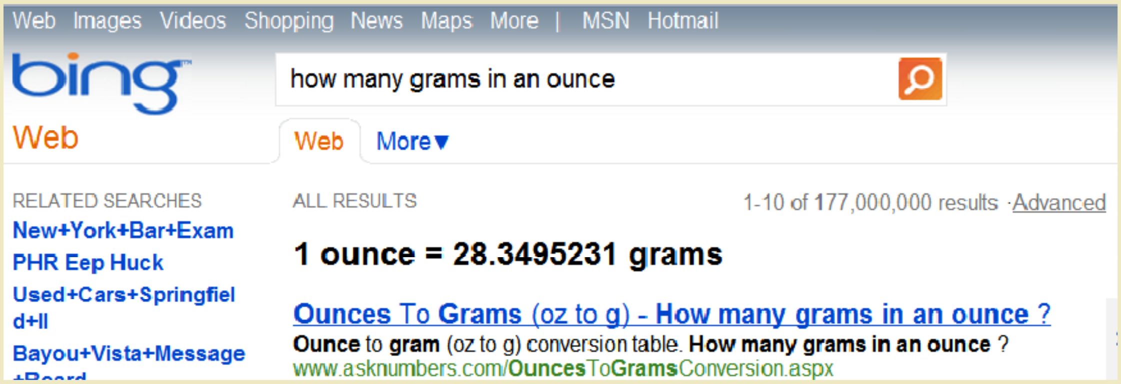
Now we get a result in a single unit of measure in the weight dimension: grams.

In[16]:=
```
totalWeight + Cholesterol + Sodium /.
  burgerNutritionFacts /.
 unitsConversions
```
Out[16]=
```
30.13 gram
```

ALICE FINDS A FIB

The original data block said that a serving size is four ounces, and we added the total weight from components to a little over 30 grams. This doesn’t seem like four ounces. Let’s add another rule to our units conversions to get everything into ounces.

Ask Bing how many grams there are in an ounce (one of us wrote the symbolic-processing software that Bing uses to answer such questions -- this software is a spiritual precursor to Jacquard)



Now capture this in our *unitsConversions* rule block:

In[17]:=

```
unitsConversions = {
  milli → 1 / 1000.0,
  gram → ounce / 28.35
 }
```

Out[17]=

```
{milli → 0.001, gram → 0.0352733686 ounce}
```

and apply it to the data again

In[18]:=

```
totalOunces =
 totalWeight + Cholesterol + Sodium /.
   burgerNutritionFacts /.
  unitsConversions
```

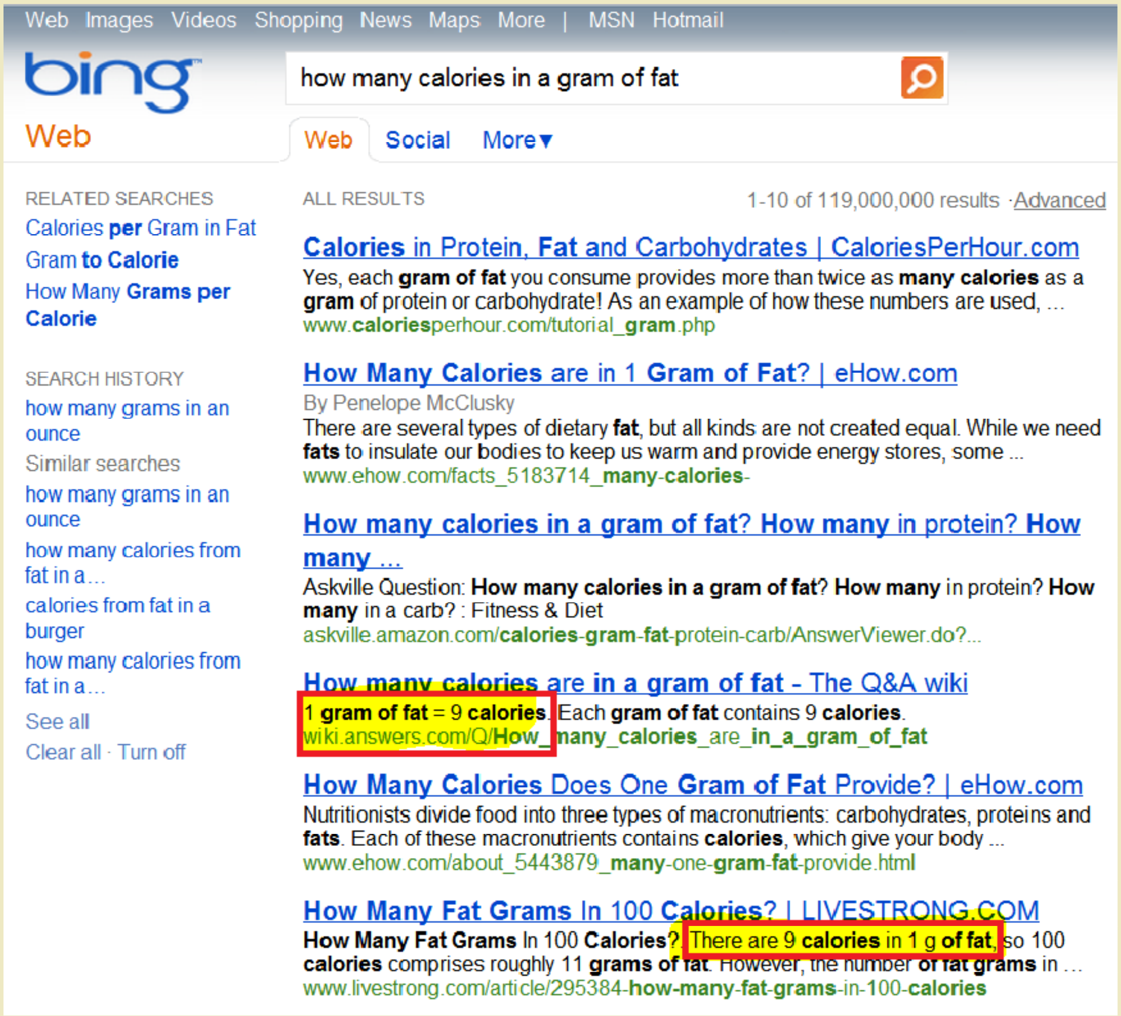
Out[18]=

```
1.0627866 ounce
```

Whoa! **The nutrition block is reporting the weight of 1/4 of a standard serving size**. How about the calories?

It’s reporting 81.0 calories from fat in a serving, but it’s incorrectly reporting that a serving has 9 grams of fat; it should be closer to 36 grams. Does the advertised “81 calories” pertain to the advertized serving of four ounces or to the implied serving of one ounce?

Let's ask Bing how many calories there are in a gram of fat:



This time, we have to go down the SERP to find the answers, but we have two that agree. There are about 9 calories per gram of fat, and that's good enough for us to find out whether the original information in the nutrition block pertains to the advertised "Serving Size" of four ounces or to the implied serving size of one ounce.

Let's write a new expression and add new rules to keep careful track, inline this time (meaning we don't need to assign these rules to a variable). Extract just the *TotalFat* from the nutritionFacts, convert those grams to grams of fat, then convert the grams of fat to calories:

In[19]:=
```
TotalFat /.
   burgerNutritionFacts /.
  {gram → gram fat} /.
 {gram fat → 9 calorie}
```

Out[19]=
```
81 calorie
```

Ok, looks like the nutrition block reports the number of calories from fat in the implied serving size of around 1 ounce.

We need different conversions for different nutritional components: carbohydrates and proteins. A little searching gives us this page, which we encode as

In[20]:=
```
fatRules = {gram → gram fat, gram fat → 9 calorie}
proteinRules = {gram → gram protein, gram protein → 4 calorie}
carbRules = {gram → gram carbs, gram carbs → 4 calorie}
```

Out[20]=
```
{gram → fat gram, fat gram → 9 calorie}
```

Out[21]=
```
{gram → gram protein, gram protein → 4 calorie}
```

Out[22]=
```
{gram → carbs gram, carbs gram → 4 calorie}
```

and apply as follows

In[23]:=
```
(TotalFat /. burgerNutritionFacts //. fatRules) +
 (Protein /. burgerNutritionFacts //. proteinRules) +
 (TotalCarbohydrate /. burgerNutritionFacts //. carbRules)
```

Out[23]=
```
165 calorie
```

Notice we used the operator "//.", which is shorthand for *ReplaceRepeated*, because we need to keep applying the rewrite rules until nothing changes any more. The other /. *ReplaceAll* operator just applies rules once.

Not only does the implied serving size underreport the weight by a factor of four, but it underreports the actual calories in the underreported weight by 5 calories.

This is why we suspect that the reported 160 calories in a serving is really the result of incorrectly adding 30 grams (the implied weight, not the advertized weight of 4 ounces) to 130 milligrams (the reported weight of the small stuff). Doing the arithmetic on the implied calories results in 165, not 160.

Let's compute the calories in an actual serving of 4 ounces and THEN decide whether we want to eat the burger. This time, we won't *ReplaceRepeated* because we want the fat and protein separated

In[24]:=
```
calorieBreakdown =
 (TotalFat /. burgerNutritionFacts /. fatRules) +
  (Protein /. burgerNutritionFacts /. proteinRules) +
  (TotalCarbohydrate /. burgerNutritionFacts /. carbRules)
```

Out[24]=
```
9 fat gram + 21 gram protein
```

Now divide by the total weight to get separated grams per ounce

In[25]:=
```
calorieBreakdown / totalOunces
```

Out[25]=

$$\frac{0.940922668\ (9\ \text{fat gram} + 21\ \text{gram protein})}{\text{ounce}}$$

Multiply by ServingSize, retrieved from the original block, and apply just the calorie rules:

In[26]:=

```
calorieRules = {
  gram fat → 9 calorie,
  gram protein → 4 calorie,
  gram carbs → 4 calorie}
```

Out[26]=

```
{fat gram → 9 calorie, gram protein → 4 calorie, carbs gram → 4 calorie}
```

In[27]:=

```
(4 ounce * calorieBreakdown / totalOunces) /. calorieRules
```

Out[27]=

```
621.008961 calorie
```

Uh, oh. That is a different story. Perhaps the salad with lemon instead of dressing would be a better lunch.

DOING IT WITH STRING KEYS

It's equally possible to do the entire scheme above using strings with embedded spaces for keys instead of using symbols in camelBack. Consider the following:

In[28]:=

```
(burgerNutritionFacts = {
    "ServingSize" → 4 ounce,
    "Amount per Serving" → 160 calorie,
    "Calories from Fat" → 81.0 calorie,
    "Saturated Fat" → 4 gram,
    "Cholesterol" → 60 milli gram,
    "Sodium" → 70 milli gram,
    "Dietary Fiber" → 0 gram,
    "Sugars" → 0 gram,
    "Total Fat" → 9 gram,
    "Protein" → 21 gram,
    "Total Carbohydrate" → 0 gram}) // gridRules
```

Out[28]=

ServingSize	4 ounce
Amount per Serving	160 calorie
Calories from Fat	81. calorie
Saturated Fat	4 gram
Cholesterol	60 gram milli
Sodium	70 gram milli
Dietary Fiber	0
Sugars	0
Total Fat	9 gram
Protein	21 gram
Total Carbohydrate	0

And our first computation as follows:

In[29]:=
```
"Total Fat" + "Dietary Fiber" + "Protein" + "Total Carbohydrate" /. burgerNutritionFacts
```
Out[29]=
```
30 gram
```

All the rest can be done similarly. This works because Jacquard and *Mathematica* do not overload + for string concatenation, but rather use a primitive *StringJoin* function and a different infix operator, namely <>.

ALICE USES JACQUARD

We expect that the advantages of automated arithmetic over symbolic units and dimensions are obvious at this point.

An Exercise

We leave it as an exercise to the reader to reproduce the computations above in native JavaScript, including at least some of the symbolic manipulation of units of measure. It will take you quite a lot of code just to catch errors, and if you go all the way to doing arithmetic with units, you will have implemented a decent fraction of the core capability of a general symbolic-computing system. Our suggestion is to begin with a version of the nutrition-facts data block similar to the following:

```
var burgerNutritionFacts003 =
{ 'Serving Size'        : [  4,    'ounce'     ],
  'Amount per Serving' : [160,    'calorie'   ],
  'Calories from Fat'  : [ 81.0, 'calorie'   ],
  'Saturated Fat'      : [  4,    'gram'      ],
  'Cholesterol'        : [ 60,    'milli gram'],
  'Sodium'             : [ 70,    'milli gram'],
  'Dietary Fiber'      : [  0,    'gram'      ],
  'Sugars'             : [  0,    'gram'      ],
  'Total Fat'          : [  9,    'gram'      ],
  'Protein'            : [ 21,    'gram'      ],
  'Total Carbohydrate' : [  0,    'gram'      ]
};
```

Just Use It

Alice doesn't want to do this exercise because she has access to Jacquard APIs. Assume that we have a JavaScript object *jqd* whose methods are those APIs (documented elsewhere). She does her computations as follows. Starting with her original JavaScript object for the nutrition facts, she gets a rules form:

```
var burgerRules = jqd.RulesFromObject(burgerNutritionFacts001);
```

She now gets a symbolic form of the weight-extraction expression:

```
var totalWeight = jqd.Expression("'Total Fat' + 'Dietary Fiber' + 'Protein' + 'Total
Carbohydrate');
```

Next, she applies the object to the expression

```
var burgerImpliedWeight = jqd.ReplaceAll(totalWeight, burgerRules);
// or jqd.Expression('totalWeight /. burgerRules')
```

```
console.logJacquardFullForm(burgerImpliedWeight)
```

which produces the following on the console

```
Times[30, gram]
```

She now encodes her unit conversions

```
var unitConversions = jqd.RulesFromObject({
  milli : 1/1000.0,
  gram  : jqd.Expression('ounce / 28.35')});
```

and applies them

```
var totalOunces = jqd.Expression(
  "totalWeight + 'Cholesterol' + 'Sodium' /.
      burgerRules /.
    unitConversions");
console.logJacquardFullForm(totalOunces);
```

Producing

```
Times[1.0627866, ounce]
```

Side stepping the intermediate, exploratory computation with ReplaceRepeated, she creates some more components of the computation:

```
var fatRules     = jqd.Expression("gram -> gram fat");
var proteinRules = jqd.Expression("gram -> gram protein");
var carbRules    = jqd.Expression("gram -> gram carbs");

var calorieBreakdown = jqd.Expression("
  (TotalFat          /. burgerNutritionFacts /. fatRules) +
  (Protein           /. burgerNutritionFacts /. proteinRules) +
  (TotalCarbohydrate /. burgerNutritionFacts /. carbRules)");

var calorieRules = jqd.Expression("calorieRules = {
  gram fat     -> 9 calorie,
  gram protein -> 4 calorie,
  gram carbs   -> 4 calorie}");
```

And finishes up with this:

```
console.logJacquardInputForm(jqd.Quotient(calorieBreakdown, totalOunces));
```

producing

```
(0.940922668436774*(9*fat*gram + 21*gram*protein))/ounce
```

and, playing with various options

```
console.logJacquardFullForm(
  jqd.ReplaceAll(
    jqd.Expression(4 ounce * calorieBreakdown/totalOunces),
    calorieRules))q;
```

producing

```
Times[621.008961,calorie]
```

SUMMARY

The Essence of Symbolic Computing

Symbolic computing is a general category of computing methods. Any program that manipulates symbols as opposed to manipulating numerical data is a symbolic program. Parsers, interpreters, compilers, regular-expression libraries; stream editing programs like Perl and Awk, macro processors like m4 and t4; templating programs, schema validators, all do symbolic computing.

Term-rewriting is a particular method of symbolic computing. It is almost universally used in theorem provers, model checkers, and computer algebra systems. It has many properties that make it suitable for more general computation. In particular, functional programming, object-oriented programming, logic programming, and even ordinary imperative programming are easy to embed in term-rewriting.

The essence of symbolic computing is treating expressions as independent, standalone objects, available at run time for manipulation and application. In ordinary programming languages like C# and JavaScript, the only way to manipulate expressions at run time is via reflection or metacircular evaluation. These techniques make code generation available at run time, but it is often so much work that it's not worth it; your programs become miniature compilers or interpreters. Programmers often work around the lack by just writing more application code.

Term-rewriting systems like Jacquard take another approach and treat expressions on exactly the same footing as other data. Jacquard is inspired by one of the most accepted and time-tested term-rewriting systems in the field: *Mathematica*.

Advantages of Symbolic Computing

Advantage number 1 is in coding applications as transformations of expressions. With term-rewriting in particular, calculating an answer means replacing terms in expressions with values from data. Code is less repetitive when it separates property-value access (projection) from business logic. That's the genesis of constructs like the “with” statement. However, the ambiguity introduced via “with” outweighs the advantage of reduction in repetitiveness.

Jacquard's solution is to invert the code: treat expressions as first-class; treat data as rewrite rules. This inversion allows developers to modify expressions and objects-as-rules independently.

The sophisticated JavaScript programmer packages expressions in functions and gains some independence from data access that way, but Jacquard removes unnecessary intermediary functions and gives direct access to expressions. That opens up scenarios like partial evaluation not available without symbolic computing.

Advantage number 2 is in symbolic arithmetic, for instace, to track units of measure. Our application includes information in grams, ounces, calories, and percentages. It is too easy to make a mistake like adding ounces to grams, or multiplying by percentages instead of by fractions. With ordinary JavaScript, the developer can only track units of measure mentally while writing the code or externally on paper or in comments. A sophisticated JavaScript programmer might record units of measure in strings and use string matching to detect errors. This is half way to symbolic computing, but includes no ability to do arithmetic.

Jacquard's symbolic arithmetic can perform routine conversions automatically and brings mistakes to the surface where they are easy to correct without backtracking through external mental or paper processes.