# Counting Calories with Symbolic Computing

# or "Why Symbolic Computing Matters"

**Brian Beckman, Erik Meijer**
**9 Aug 2012**

## INTRODUCTION AND ABSTRACT

We show plausible application code -- counting calories -- that a web developer might write using Jacquard, a term-rewriting system in pure JavaScript that runs both in the browser and in the server. We show how Jacquard's *symbolic computing* 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. We motivate Wolfram Research's *Mathematica* [1] as an Integrated Development Environment (IDE) [2] for Jacquard expressions.

Symbolic computation enables, amongst other capabilities, robust units conversion [3]. This lets our developer catch certain misleading bits of consumer information in a Nutrition Facts Label [4]. Such units conversions are not possible in JavaScript without a symbolic computing facility essentially equivalent to Jacquard's method of term rewriting [5]. We finally construct a Nutrition Facts Label on-the-fly from unit ingredients and a previously unknown recipe. This turns out to be a sum in a vector space with basis vectors comprising the Nutrition Facts Labels of the ingredients. We demonstrate multilevel pattern-matching to make the code concise. We also motivate remote evaluation of expressions in this scenario: bringing the computation to the data can be much cheaper than bringing the data to the computation.

We point out that symbolic computing is not exotic. Even though ordinary JavaScript programmers might not use it in their own code every day, they use it every time they edit, compile, interpret, or debug their code. Historically, symbolic computing has been a mission-critical, if invisible, part of real-world, multi-billion-dollar operating system kernels for tasks such as network and driver configuration [6] and dynamic security policy evaluation [7].

The broader point is that symbolic computing is an under-appreciated methodology, routinely applied "under the radar" for many decades but certainly ripe for much broader use in the mainstream cloud programming.

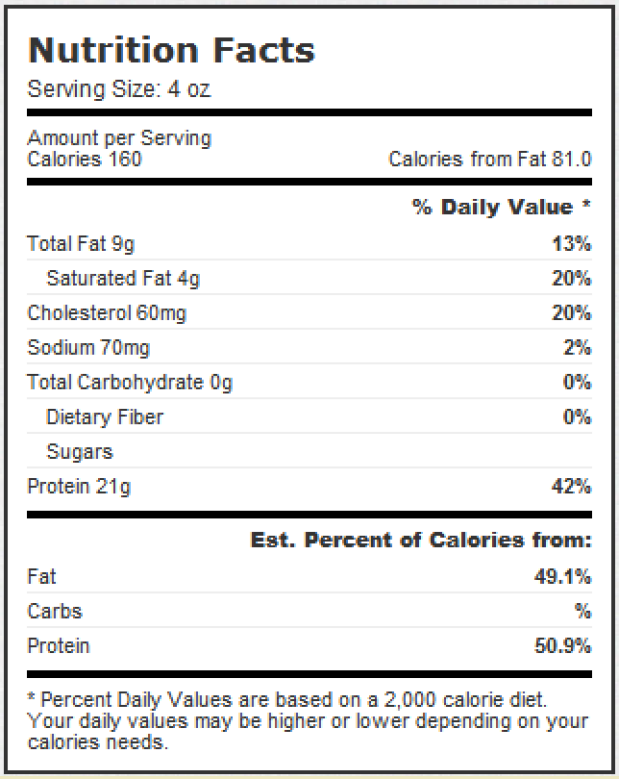
## SCENARIO

Consider the following Nutrition Facts Label, mined at random from the web, 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. Alice's questions are the following:

- Does the calorie total implied by the breakdown match the calorie count presented in the “amount per serving?”
- Does the total weight implied by the breakdown match the total weight presented in the “serving size”?

Let's show what Alice might write in JavaScript, then show a Jacquard computation designed to produce 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 NFL 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 data. She accepts that she can’t later use dot notation with such keys [8]. For uniformity of style, Alice will use only the square-bracket *indexer* notation everywhere.

The next step is to add up the weights of all the nutritional components. Alice knows that milligrams are tiny by comparison to grams, so she simply omits them in this first version, perhaps with a shade of concern that another programmer inheriting her code might erroneously put them back in.

She at least takes the 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 knows this means grams, and she carries that mentally because she doesn't have a convenient, straightforward way to carry it in her code.

## ALICE'S DREAMS

Alice takes a moment to reflect on the code she just wrote. Will it withstand criticism from other programmers?

### Repetition Considered Harmful

DRY (Don't Repeat Yourself) [9] is programmer mantra [10] nowadays. Unecessary repetition in code just increases risk of programmer error.

Even though Alice is careful to align parts of the arithmetic expression vertically [11], the repeated name of the object parameter 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 symbolic 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 [12], 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 in the object-access expression.

### 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 outweighs its superficial attractiveness. The overall concensus is to avoid it [13].

## 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!

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 -- an unrecoverable error -- an invalid program. In C#, Java, C++, and most other languages, unbound symbols don't even compile.

There is an analogy to lazy evaluation. JavaScript eagerly interprets all symbols, before they're needed. Jacquard, and term-rewriting systems in general, only interpret them when they're used. And even then, a symbol without a binding to a value is *not* an error, it's just the symbol itself as a first-class, atomic value. An expression that does not reduce further is an expression in so-called normal form [14].

### Rewriting Alice's Dreams

First, we rewrite Alice's Second Dream: the version using camelBack symbols for keys because it makes the code look like code instead of weird arithmetic on strings. 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 this dream 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 is a pair of a ***pattern*** and a ***replacement***. A rule tells the evaluator "if you can match the left-hand side, replace it with the right-hand side, after substituting any values for pattern variables." The patterns-and-rules sublanguage is similar to regular expressions familiar from JavaScript, Python, Perl, and text editors, except that it operates over other expressions in the language and not just on strings. It lets the programmer write rules that will match whole classes of richly structured expressions. It allows parts of a program to rewrite other parts of a program, to be a "compiler," if you like, for other parts of the program. It lets the programmer effortlessly create embedded domain-specific languages [15], almost without being aware of it. The pattern-and-rule facility is extraordinarily powerful.

A rule with only symbolic constants for patterns -- with no pattern variables on the left-hand side -- is equivalent to a JavaScript or JSON property, *i.e.*, key-value pair. Symbolic constants are trivial patterns that match only themselves.

The fact that we can always represent an object as a list of rules reveals the deeper fact that *objects are just functions from keys to values*, a fact that is also obvious from realizing that objects are implemented as hash tables or search 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. Each line in the Nutrition-Fact Label is a product of a number and a constant representing a unit of measure.

We use a pretty-print function from the Jacquard library to display this definition in an aligned grid. Notice that Jacquard and *Mathematica* do reduce the terms with zero coefficients:

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 | Times 4 ounce |
| AmountPerServing | Times 160 calorie |
| CaloriesFromFat | Times 81. calorie |
| SaturatedFat | Times 4 gram |
| Cholesterol | Times 60 gram milli |
| Sodium | Times 70 gram milli |
| DietaryFiber | 0 |
| Sugars | 0 |
| TotalFat | Times 9 gram |
| Protein | Times 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 [16].

- 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, as a collection of rules, 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 prefer to imagine that the expression indexes the object. Let's skip that dichotomy 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 applying 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 | Times 4 ounce |
| AmountPerServing | Times 130 calorie |
| CaloriesFromFat | Times 9. calorie |
| SaturatedFat | Times 0.4 gram |
| Cholesterol | Times 68 gram milli |
| Sodium | Times 77 gram milli |
| DietaryFiber | 0 |
| Sugars | Times 0.1 gram |
| TotalFat | gram |
| Protein | Times 27 gram |
| TotalCarbohydrate | 0 |

then we would get a different result:

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

Out[9]=
```
28 gram
```

The main point here is that *the expression for the sum of the weights* and *the expression of the data object as a list of rules* 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 a completely independent existence from the data: the best she could do was parameterize the data, and that's when she encountered her verbosity / repetition problem. Here, we don't need a function and don't need parameters. Getting rid of a function is good just because it shortens the code, and shorter code is almost always better code [17].

## Expressions are Values

We can save the expression for the total weight in a variable

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

Out[10]=
```
DietaryFiber + Protein + TotalCarbohydrate + TotalFat
```

The symbol *totalWeight* has a value, namely the symbolic, normal-form expression we've been carrying along. Symbols can rewrite to other symbolic expressions; they need not reduce all the way to numbers. Let's symbolically add the little milligram items 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 the evaluator canonically reordered “milli gram” to “gram milli.” That’s fine, since “gram milli” means “gram times milli,” and “times” is independent of order just like “plus.”

### Without Units, Exposed to Error

The fundamental symbolic nature of the computation has caught this units incompatibility for us. In Alice’s first JavaScript solution, should could only have caught this by manually checking the calculation. 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 should 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 this numerical value 160 doesn’t arise from plausible calculations over other data in the nutrition block. There is only one instance of the number 160 to be found in this sample, and that’s from this wildly mistaken computation of adding grams to milligrams.

It's a strong hypothesis that the original authors of the nutrition block added up the weights and got a wrong number, then compounded the error by putting the wrong number in the wrong slot.

### Robust Unit Conversions Close the Gap

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

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

Out[13]=
```
{gram milli → 0.001 gram}
```

applied, in series, to the expression that produces incompatible units

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

Out[14]=
```
30.13 gram
```

produces a final, meaningful result.

We see that we can easily compose ReplaceAll invocations in chains, and now we also see why canonicalizing 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.

Now that the point about canonicalization is made, we can see that our *unitsConversions* rules are actually more restrictive than necessary. 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 Jacquard and *Mathematica,* variables are mutable.

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

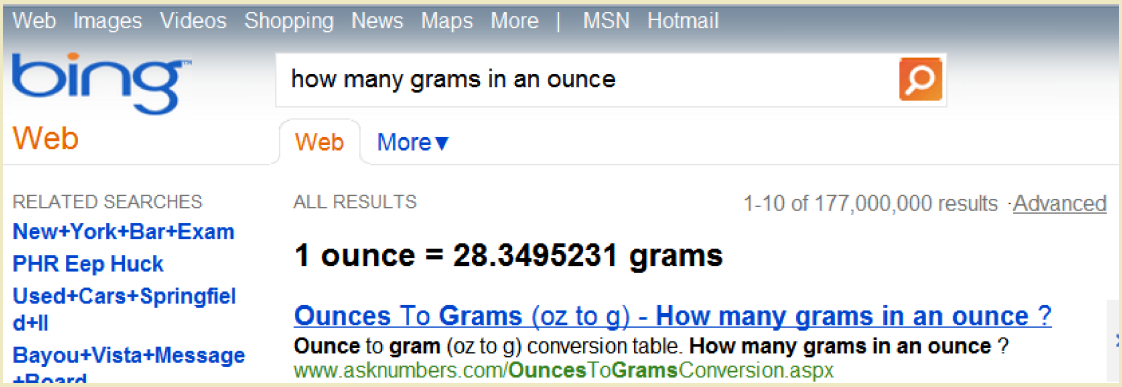
Now, as with the earlier *unitsConversion*, we get a result in a single unit of measure in the weight dimension *grams*.

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

## ALICE FINDS A FIB

The original data block declared 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 (as an aside, one of us wrote the symbolic-processing software that Bing uses to answer such questions -- that software is a spiritual ancestor of 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 → 0.0352734 ounce}
```

and apply it to the data again

In[18]:=
```
totalOunces =
 totalWeight + Cholesterol + Sodium /.
   burgerNutritionFacts /.
  unitsConversions
```

Out[18]=
```
1.06279 ounce
```

Whoa! The NFL reports that a serving size is 4 ounces, but adding up the weights of the ingredients yields only 1 ounce. So, even if we believed the reported 160 calories in the NFL, could that be for 1 ounce? If so, the real calories for 4 ounces would be over 600.

There is certainly strong incentive to fib, here. No one wants to eat a 1-ounce burger patty, so maybe just report a reasonable size like 4 ounces. That's rather standard for a serving of meat, no? But no one will eat a 600-calorie burger patty, so let's just report the calories for 1 ounce. It *could* be the result of a cascade of honest mistakes, and we've certainly seen that they're all-too-easy without symbolic computing. But it's perhaps even more plausible that there is some willful malfeasance going on.

Let’s check the calories in detail. The NFL reports 81.0 calories from fat in a serving, but it incorrectly reports 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 results page 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 label pertains to the advertised "Serving Size" of four ounces or to the implied serving size of one ounce.

Write a new expression and chain some 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 label data, 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 label 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 [18], which we encode in rules 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};
```

and apply as follows

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

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

Notice the operator "//.", shorthand for *ReplaceRepeated*, because we must keep applying the rewrite rules until nothing changes any more. The /. *ReplaceAll* operator just applies rules once.

*Not only does the implied serving size underreport the declared 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 declared 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
```

Divide by the total weight to get separated grams per ounce

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

Out[25]=
$$\frac{1}{\text{ounce}}\,0.940923\ (9\ \text{fat gram} + 21\ \text{gram protein})$$

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};
```

In[27]:=
```
(4 ounce * calorieBreakdown / totalOunces) /. calorieRules
```

Out[27]=
```
621.009 calorie
```

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

We note in passing that the burger NFL does not report any weight or calories from carbs, but the expressions we wrote are general and would handle other nutrition fact labels that do have carbs.

## 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 = {
    "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}) // gridRules
```

Out[28]=

| | |
|---|---|
| Serving Size | Times 4 ounce |
| Amount per Serving | Times 160 calorie |
| Calories from Fat | Times 81. calorie |
| Saturated Fat | Times 4 gram |
| Cholesterol | Times 60 gram milli |
| Sodium | Times 70 gram milli |
| Dietary Fiber | 0 |
| Sugars | 0 |
| Total Fat | Times 9 gram |
| Protein | Times 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 hope 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 in JavaScript. 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 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));
```

producing

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

## CREATING NFL'S ON-THE-FLY

We spent the first half of this paper demonstrating that NFLs mined off the web are untrustworthy, as an exhibition of basic technique in symbolic computing. Now, we change philosophy altogether and pursue a different agenda: using mined NFLs to generated new NFLs from recipes. We assume that either we're going to correct the mined NFLs or that we're just going to accept them as the best we can do for now. In

any event, the following is a much more adventurous use of symbolic computing, involving rules that write rules that rewrite expressions, canonicalization, normalization (in the vector-space sense), and monadic operators like **SelectMany** and **Fold**.

This following is your mom's secret recipe for Pasta Primavera. It doesn't have a published Nutrition Facts Label. Can we compute one on-the-fly?

In[30]:=
```
myRecipe={
 1.0 Tablespoon "olive oil",
 16.0 Ounce "zucchini",
 3.5 Teaspoon "salt",
 1.5 Pound "eggplant",
 1.0 "onion",
 2.0 "bell pepper",
 14.5 Ounce "stewed tomato",
 0.5 Teaspoon "black pepper",
 0.5 Teaspoon "dried basil",
 0.5 Teaspoon "sugar",
 12.0 Ounce "pasta",
 0.25 Cup "parmesan cheese"};
```

The recipe is expressed as symbolic multiplications -- products of numerical quantities, symbolic units of measure, and strings that name the ingredients. We will use patterns and rules to reduce this to an NFL in a sequence of rewrites: first, convert everything to a weight in grams; then, mine the web for unit NFLs, one for each ingredient; then multiply each unit ingredient by the quantity in the recipe; then add the NFLs component-wise. You will recognize this as constructing a vector in NFL space wherein the basis vectors are the unit ingredients.

Some of the ingredients are expressed as volumes: Tablespoon, Teaspoon, Cup. Let's mine some density facts from the web:

In[31]:=
```
density["olive oil"] = Mean[{6.68,7.67}] * Pound / Gallon;
density["salt"] = 5.69 Gram / Teaspoon;
density["black pepper"] = 2.1 Gram / Teaspoon;
density["dried basil"] = 1.0 Gram / Teaspoon;
density["sugar"] = 4.2 Gram / Teaspoon;
density["parmesan cheese"] = 88 Gram / Cup;
```

Here, we represent the facts as a lookup table just for variety. This is very similar to the way we would write it in JavaScript. This could also be represented as a list of replacement rules as we have been doing all along, and, indeed, it is internally. Since we don't need to apply these rules to expressions, but only need to look up the values corresponding to each ingredient, we can write them in a form more familiar to the JavaScript programmer in this case.

Given a target volume measure and a density fact, the following is a rule to rewrite the density fact in the form of Grams per unit of the target volume measure. The rule is written in functional notation with a patten on the left and a replacement to the right of the definition operator **:=**. The pattern will match any expression of the form *g*[*t*, *d* ∗ *w* / *v*], where *g* is the literal symbol **gramPerTargetVolumeFromDensity**, *t* is the pattern variable **targetVolume_**, which will match any expression, *d* is a number, *w* is the pattern variable **weight_**, which will match any expression, and *v* is the pattern variable **volume_**, which will match any expression. The replacement is **(d * Convert[weight, Gram]) / Convert[volume, targetVolume]**, wherein the appearances of the pattern variables without their underscores will be substituted by their values from the pattern match, and **Gram** is a symbolic constant. **Convert** rewrites to a more generalized form of the units conversion operators we used above in Alice's task, and its definition will be omitted here for brevity. We've also transitioned to capitalized names for units of measure in keeping with standard practice in international standards.

In[37]:=
```
gramPerTargetVolumeFromDensity[
  targetVolume_,
  d_?NumberQ * weight_ / volume_] :=
 (d * Convert[weight, Gram]) / Convert[volume, targetVolume]
```

The next rule matches the pattern of a numerical quantity times an ingredient name times a volume from an explicit list of symbolic constants, and will produce a rule to rewrite this quantified volume ingredient as a weight in **Grams**. Note that this is a meta-rule: a rule producing a new rule. There is an additional catch-all rule that rewrites anything that does not match that pattern into the emtpy list, **{}**. The reason for this catch all is that we are going to run this over the recipe through **SelectMany**, the composition of **map** and **flat**. **ten-once**, also known as flatmap, concatmap, and bind [19]: the mother of all monadic operators [20]. The definition of this utility function is omitted for brevity.

In[38]:=
```
weightRuleFromQuantifiedIngredientVolume[quantity_?NumberQ  *  ingredient_  *
    volume : (Teaspoon | Tablespoon | Cup | FluidOunce | Pint | Gallon)] :=

  ingredient * volume  →
   ingredient * gramPerTargetVolumeFromDensity[volume,
     density[ingredient]] * volume;
weightRuleFromQuantifiedIngredientVolume[___] := {}
```

Here are all the volume-to-weight fules from the original recipe.

In[40]:=
```
(volumeRules = SelectMany[myRecipe, weightRuleFromQuantifiedIngredientVolume]) //
 gridRules
```

Out[40]=

| | |
|---|---|
| Times \| olive oil \| Tablespoon | Times \| 12.713 \| olive oil \| Gram |
| Times \| salt \| Teaspoon | Times \| 5.69 \| salt \| Gram |
| Times \| black pepper \| Teaspoon | Times \| 2.1 \| black pepper \| Gram |
| Times \| dried basil \| Teaspoon | Times \| 1. \| dried basil \| Gram |
| Times \| sugar \| Teaspoon | Times \| 4.2 \| sugar \| Gram |
| Times \| parmesan cheese \| Cup | Times \| 88 \| parmesan cheese \| Gram |

We hope that the brevity, clarity, and flexibility of the rule - and - replacement paradigm has by now impressed you. The equivalent in JavaScript or C # would be a very considerable amount of code.

Let us do likewise for ingredients expressed as whole items: mine the web for typical weights and convert the ingredient lines into rules for converting their units and identifiers into weights. The technique will be exactly as above: using a rule-pruducing rule and **SelectMany**:

In[41]:=
```
wholeItemWeight["onion"] = (1.0 / 3) Pound;
wholeItemWeight["bell pepper"] = 0.5 Pound / 4;
```

In[43]:=
```
weightRuleFromQuantifiedWholeItemIngredient[_ * _String * _Symbol] = {};
weightRuleFromQuantifiedWholeItemIngredient[
   _?NumberQ  *  ingredient_ (* do match a pair *)] :=
  (* generate the following rule *)

  ingredient  →  ingredient * wholeItemWeight[ingredient];
weightRuleFromQuantifiedWholeItemIngredient[___] = {};
```

In[46]:=
```
(wholeItemRules = SelectMany[myRecipe, weightRuleFromQuantifiedWholeItemIngredient]) //
 gridRules
```

Out[46]=

| | |
|---|---|
| onion | Times 0.333333 onion Pound |
| bell pepper | Times 0.125 bell pepper Pound |

Now, let's apply the all the rules we just created from the original recipe and the mined volume and whole item facts to the original recipe to produce a copy of the recipe with all items as weights:

In[47]:=
```
(myRecipe /. volumeRules /. wholeItemRules) // gridRules
```

Out[47]=

| | | | |
|---|---|---|---|
| Times | 12.713 | olive oil | Gram |
| Times | 16. | zucchini | Ounce |
| Times | 19.915 | salt | Gram |
| Times | 1.5 | eggplant | Pound |
| Times | 0.333333 | onion | Pound |
| Times | 0.25 | bell pepper | Pound |
| Times | 14.5 | stewed tomato | Ounce |
| Times | 1.05 | black pepper | Gram |
| Times | 0.5 | dried basil | Gram |
| Times | 2.1 | sugar | Gram |
| Times | 12. | pasta | Ounce |
| Times | 22. | parmesan cheese | Gram |

and then convert all such weights to Grams by mapping a function over teh result above:

In[48]:=
```
(recipeInGrams =
   Map[Function[
     ingredient,
     Convert[ingredient, Gram]],
    myRecipe /. volumeRules /. wholeItemRules]) // gridRules
```

Out[48]=

| | | | |
|---|---|---|---|
| Times | 12.713 | olive oil | Gram |
| Times | 453.592 | zucchini | Gram |
| Times | 19.915 | salt | Gram |
| Times | 680.388 | eggplant | Gram |
| Times | 151.197 | onion | Gram |
| Times | 113.398 | bell pepper | Gram |
| Times | 411.068 | stewed tomato | Gram |
| Times | 1.05 | black pepper | Gram |
| Times | 0.5 | dried basil | Gram |
| Times | 2.1 | sugar | Gram |
| Times | 340.194 | pasta | Gram |
| Times | 22. | parmesan cheese | Gram |

Just for fun, let's see how much Pasta Primavera the recipe produces. We'll do so with another pattern that strips out the string identifier, keeping only the quantity and the weight, adding it all up, and converting the result to pounds.

In[49]:=
```
Convert[Apply[Plus, Cases[recipeInGrams, q_ * _String * u_Symbol → q u]], Pound]
```

Out[49]=
```
4.86806 Pound
```

The recipe produces almost five pounds. Let's assume it's designed to serve six.

Next, let's create a constructor that will build an NFL object as a list of rules, from facts mined off the web. As a side effect, we will keep a list of names of the NFLs and a lookup table that can retrieve any NFL given its name. This will help us below when we write the code to perform the vector sum from the recipe. Note the style of this constructor could be improved considerably. Its parameter list is purely positional; it does not leverage pattern-matching to be robust against mistakes in the order of arguments. We present it this way just to show an example of ordinary procedural programming in Jacquard.

In[50]:=
```
nflNames={};
createNutritionFactsLabel[name_,
servingSize_,totalCalories_,fatCalories_,
totalFat_,totalFatPercent_,saturatedFat_,
saturatedFatPercent_,transFat_,
cholesterol_,cholesterolPercent_,sodium_,
sodiumPercent_,totalCarbohydrates_,
totalCarbohydratesPercent_,dietaryFiber_,
dietaryFiberPercent_,sugars_,protein_,
proteinPercent_,vitaminAPercent_,
vitaminCPercent_,calciumPercent_,ironPercent_]:=
(AppendTo[nflNames,name];
nfls[name]={"name"→name,"serving size"→servingSize,
"total calories"→totalCalories,"fat calories"→fatCalories,
"total fat"→totalFat,"% daily total fat"→totalFatPercent,
"saturated fat"→saturatedFat,
"% daily saturated fat"→saturatedFatPercent,
"trans fat"→transFat,"cholesterol"→cholesterol,
"% daily cholesterol"→cholesterolPercent,
"sodium"→sodium,"% daily sodium"→sodiumPercent,
"total carbohydrates"→totalCarbohydrates,
"% daily carbohydrates"→totalCarbohydratesPercent,
"dietary fiber"→dietaryFiber,
"%daily dietary fiber"→dietaryFiberPercent,
"sugars"→sugars,"protein"→protein,
"% daily protein"→proteinPercent,
"vitamin A"→vitaminAPercent,"vitamin C"→vitaminCPercent,
"calcium"→calciumPercent,"iron"→ironPercent});
```

Let's demonstrate this constructor on one ingredient, inspecting the resulting object

In[52]:=
```
createNutritionFactsLabel["olive oil", 216 Gram, 1910 Calorie,
  1910 Calorie, 216 Gram, 332 Percent, 30 Gram, 149 Percent, 0 Gram,
  0 Gram, 0 Percent, 4 Milli Gram, 0 Percent, 0 Gram, 0 Percent,
  0 Gram, 0 Percent, 0 Gram, 0 Gram, 0 Percent,
  0 Percent, 0 Percent, 0 Percent, 7 Percent] // gridRules
```

Out[52]=

| | |
|---|---|
| name | olive oil |
| serving size | Times 216 Gram |
| total calories | Times 1910 Calorie |
| fat calories | Times 1910 Calorie |
| total fat | Times 216 Gram |
| % daily total fat | Times 332 Percent |
| saturated fat | Times 30 Gram |
| % daily saturated fat | Times 149 Percent |
| trans fat | 0 |
| cholesterol | 0 |
| % daily cholesterol | 0 |
| sodium | Times 4 Gram Milli |
| % daily sodium | 0 |
| total carbohydrates | 0 |
| % daily carbohydrates | 0 |
| dietary fiber | 0 |
| %daily dietary fiber | 0 |
| sugars | 0 |
| protein | 0 |
| % daily protein | 0 |
| vitamin A | 0 |
| vitamin C | 0 |
| calcium | 0 |
| iron | Times 7 Percent |

Let's run the others for side-effect, but save space by not presenting the results:

In[53]:=
```
createNutritionFactsLabel["zucchini", 124 Gram, 20 Calorie, 2 Calorie,
  0 Gram, 0 Percent, 0 Gram, 0 Percent, 0 Gram,
  0 Gram, 0 Percent, 12 Milli Gram, 1.0 Percent,
  4 Gram, 1.0 Percent, 1.0 Gram, 5 Percent, 2 Gram, 2 Gram, 0 Percent,
  5 Percent, 35 Percent, 2 Percent, 2 Percent];
(* //gridRules *)
```

In[54]:=
```
createNutritionFactsLabel["salt", 1. Cup, 0 Calorie, 0 Calorie,
  0 Gram, 0 Percent, 0 Gram, 0 Percent, 0 Gram,
  0 Gram, 0 Percent, 113 174 Milli Gram, 4716 Percent,
  0 Gram, 0 Percent, 0 Gram, 0 Percent, 0 Gram, 0 Gram, 0 Percent,
  0 Percent, 0 Percent, 7 Percent, 5 Percent];(* //gridRules *)
```

In[55]:=
```
createNutritionFactsLabel["eggplant", 82 Gram, 20 Calorie, 1.0 Calorie,
  0 Gram, 0 Percent, 0 Gram, 0 Percent, 0 Gram,
  0 Gram, 0 Percent, 2 Milli Gram, 0 Percent,
  5 Gram, 2 Percent, 3 Gram, 11 Percent, 2 Gram, 1.0 Gram, 2 Percent,
  0 Percent, 3 Percent, 1.0 Percent, 1.0 Percent];(* //gridRules *)
```

In[56]:=
```
createNutritionFactsLabel["onion", 160 Gram, 64 Calorie, 1.0 Calorie,
  0 Gram, 0 Percent, 0 Gram, 0 Percent, 0 Gram,
  0 Gram, 0 Percent, 6 Milli Gram, 0 Percent,
  15 Gram, 5 Percent, 3 Gram, 11 Percent, 7 Gram, 2.0 Gram, 0 Percent,
  0 Percent, 20 Percent, 4 Percent, 2 Percent];(* //gridRules *)
```

In[57]:=
```
createNutritionFactsLabel["bell pepper", 186 Gram, 50 Calorie, 3.0 Calorie,
  0 Gram, 1.0 Percent, 0 Gram, 0 Percent, 0 Gram,
  0 Gram, 0 Percent, 4 Milli Gram, 0 Percent,
  12 Gram, 4 Percent, 2 Gram, 7 Percent, 2 Gram, 2 Gram, 0 Percent,
  7 Percent, 569 Percent, 2 Percent, 5 Percent];(* //gridRules *)
```

In[58]:=
```
createNutritionFactsLabel["stewed tomato", 101 Gram, 80 Calorie, 24.0 Calorie,
  3 Gram, 4 Percent, 1.0 Gram, 3 Percent, 0 Gram,
  0 Gram, 0 Percent, 460 Milli Gram, 19 Percent,
  13 Gram, 4 Percent, 2 Gram, 7 Percent, 0 Gram, 2 Gram, 0 Percent,
  13 Percent, 31 Percent, 3 Percent, 6 Percent];(* //gridRules *)
```

In[59]:=
```
createNutritionFactsLabel["black pepper", 1. Tablespoon,
  16 Calorie, 2 Calorie, 0 Gram, 0 Percent, 0 Gram, 0 Percent, 0 Gram,
  0 Gram, 0 Percent, 3 Milli Gram, 0 Percent, 4 Gram, 1. Percent,
  2 Gram, 7 Percent, 0 Gram, 1. Gram, 0 Percent,
  0 Percent, 2 Percent, 3 Percent, 10 Percent];(* //gridRules *)
```

In[60]:=
```
createNutritionFactsLabel["dried basil", 1. Teaspoon,
  1.0 Calorie, 0 Calorie, 0 Gram, 0 Percent, 0 Gram, 0 Percent, 0 Gram,
  0 Gram, 0 Percent, 0 Gram, 0 Percent, 0 Gram, 0 Percent,
  0 Gram, 1.0 Percent, 0 Gram, 0 Gram, 0 Percent,
  1.0 Percent, 1.0 Percent, 1.0 Percent, 1.0 Percent];(* //gridRules *)
```

In[61]:=
```
createNutritionFactsLabel["sugar", 2 Gram, 11 Calorie, 0 Calorie,
  0 Gram, 0 Percent, 0 Gram, 0 Percent, 0 Gram,
  0 Gram, 0 Percent, 0 Gram, 0 Percent,
  3 Gram, 1.00 Percent, 0 Gram, 0 Percent, 3 Gram,
  0 Gram, 0 Percent,
  0 Percent, 0 Percent, 7 Percent, 5 Percent];(* //gridRules *)
```

In[62]:=
```
createNutritionFactsLabel["pasta",
  128 Gram, 369 Calorie, 25 Calorie,
  3 Gram, 5 Percent, 0 Gram, 2 Percent, 0 Gram,
  93 * Milli * Gram, 31 Percent, 33 Milli Gram, 1.0 Percent,
  70 Gram, 23 Percent, 0 Gram, 0 Percent, 0 Gram,
  14 Gram, 0 Percent,
  1.0 Percent, 0 Percent, 2 Percent, 24 Percent];(* //gridRules *)
```

In[63]:=
```
createNutritionFactsLabel["parmesan cheese", 100 Gram,
  431 Calorie, 251 Calorie, 29 Gram, 44 Percent, 17 Gram, 86 Percent, 0 Gram,
  88 Milli * Gram, 29 Percent, 1529 Milli Gram, 64 Percent,
  4 Gram, 1.00 Percent, 0 Gram, 0 Percent, 1 Gram,
  38 Gram, 0 Percent,
  9 Percent, 0 Percent, 111 Percent, 5 Percent];(* //gridRules *)
```

We'll add an ingredient that doesn't appear in the recipe just to test that it's ignored in the preparation of the final result:

In[64]:=
```
createNutritionFactsLabel["strange extra ingredient", 100 Gram,
  431 Calorie, 251 Calorie, 29 Gram, 44 Percent, 17 Gram, 86 Percent, 0 Gram,
  88 Gram, 29 Percent, 1529 Milli Gram, 64 Percent,
  4 Gram, 1.00 Percent, 0 Gram, 0 Percent, 1 Gram,
  38 Gram, 0 Percent,
  9 Percent, 0 Percent, 111 Percent, 5 Percent];
```

At this point, we have a recipe as a list of ingredients in grams with numerical coefficients. We also have a database of NFLs that give all the components of the label per service size.

The next step is to make a rule to canonicalize the units in an NFL. The following takes an NFL and maps a function over a transform of the NFL. The NFL is a list of rules, and the transform will also be a list of rules. The transform will do a **ReplaceAll** with a pattern that matches original rule lines in the NFL and specifies a new rule to write out. The pattern will match any rule line in the NFL of the form $k \to n * v$, where $k$ is the pattern variable **keyWithVolume_**, $n$ is a numerical amount, and $v$ is a volume chosen from an explicit list of known volumes. The overall rule in the transform will rewrite that matched rule line into one with densities transformed into weights per unit volume with a weight unit driven by the density fact, as before. Finally, the function mapped over the transformed rule lines will map the given weights into Grams. We have emphasized the rule arrows to emphasize the fact that there are three uses of rule here: one for the input rule, →; one for the interior (meta-) rule that rewrites the input rule into the output rule, written :→; and one for the output rule, another instance of →. The interior rule is a **RuleDelayed** -- that's the meaning of the strange colon-arrow. It tells the evaluator to delay evaluation of the right-hand side until rule-application time so that the density lookup will not be done too early. The normal evaluation time for the right-hand sides of Rules is at rule-definition time, and that is often fine. In this case, however, it would cause the density lookup to be done on an input of `"name"/.nfl`, which would not reduce.

```
canonicalizeUnits[nfl_] :=
 Map[Function[rule, rule[[1]] → Quiet[N@Convert[rule[[2]], Gram]]],
  (nfl /. {
     (keyWithVolume_ →
        amount_?NumberQ *
         volume : (Teaspoon | Tablespoon | Cup | FluidOunce | Pint | Gallon)) :→
      keyWithVolume → amount * volume *
        gramPerTargetVolumeFromDensity[volume, density["name" /. nfl]]})]
```

Make new a new list to contain the canonicalized NFLs (notice this does not transform the NFLs in the source lookup table / database; we do that in another step below, preferring to operate on lists to take advantage of functional-programming techniques). Suppress the output with semicolon, as it is lengthy; during development, it is useful to inspect it.

In[88]:=
```
(canonicalizedNfls = Map[canonicalizeUnits, Map[Function[name, nfls[name]], nflNames]]);
```

Calculate the ***norm*** of each NFL, that is, its serving size in Grams. Introduce two bits of space-saving syntax: `&` is a unary postfix operator that denotes its argument (on its left) to be a pure function with one

argument called `#`; `/@` is a binary infix operator that maps the function in its left-hand argument over the list in its right-hand argument:

In[89]:=
```
norms = ("serving size" / Gram /. # &) /@ canonicalizedNfls
```

Out[89]=
```
{216., 124., 273.12, 82., 160., 186., 101., 6.3, 1., 2., 128., 100., 100.}
```

Now a function (pattern and replacement rule) to scale an NFL; with one bit of logic to ignore the special `name` line in every NFL:

In[73]:=
```
scaleNfl[nfl_, scalar_] :=
 Map[Function[line, If[line[[1]] === "name",
    line, (* skip the name line *)
    line[[1]] → line[[2]] * scalar]], nfl]
```

Now normalize the NFLs by applying `scaleNfl` pairwise over the `canonicalizedNfls` and the norms. The following uses `MapThread`, which is usually known as `Zip` in functional programming:

In[74]:=
```
normalizedNfls = MapThread[scaleNfl, {canonicalizedNfls, 1 / norms}];
```

Now put the normalized NFLs into a new lookup structure, this time in the familiar form of a list of rules that map the name in the NFL to the full NFL:

In[90]:=
```
normalizedNflsObj = Map[Function[nfl, ("name" /. nfl) → nfl], normalizedNfls];
```

Now a function (pattern and replacement rule) that will match any ingredient line in a recipe, look up its corresponding NFL, and scale the NFL by the numerical coefficient of the ingredient:

```
scaledNflFromIngredient[qtty_?NumberQ * name_String * Gram] :=
 If[(name /. normalizedNflsObj) === name, (* ingredient wasn't in DB *)
  {}, (* SelectMany will flatten this out *)
  {scaleNfl[name /. normalizedNflsObj, qtty]}]
```

Apply this function to the recipe using `SelectMany`, which will flatten out any empty NFLs returned from ingredients that are not in the database. This is only one option for handling that kind of error, but it is expressed concisely.

In[92]:=
```
(scaledNfls = SelectMany[recipeInGrams, scaledNflFromIngredient]);
```

Write a function (pattern and replacement rule) that adds any two NFLs pointwise, in the manner of vectors, throwing an exception in case of error:

In[82]:=
```
sumNfls[nfl1_, nfl2_] :=
 MapThread[Function[{line1, line2},
   If[line1[[1]] === line2[[1]], (* don't add up dimensions that don't match *)
    line1[[1]] → (line1[[2]] + line2[[2]]) // Chop,
    Throw["foo"]]],  {nfl1, nfl2}]
```

Finally, `Fold` [21] [22] this binary function over the scaled NFLs of the recipe; scale the entire result by 1/6 to get one serving; drop the name line via `Rest` and display:

In[93]:=
```
scaleNfl[Fold[sumNfls, First[scaledNfls], Rest[scaledNfls]], 1 / 6] // Rest // gridRules
```

Out[93]=

| | |
|---|---|
| serving size | Times 368.019 Gram |
| total calories | Times 309.723 Calorie |
| fat calories | Times 58.4133 Calorie |
| total fat | Times 6.54603 Gram |
| % daily total fat | Times 9.89979 Percent |
| saturated fat | Times 1.59594 Gram |
| % daily saturated fat | Times 7.53584 Percent |
| trans fat | 0 |
| cholesterol | Times 0.044422 Gram |
| % daily cholesterol | Times 14.7951 Percent |
| sodium | Times 1.76965 Gram |
| % daily sodium | Times 73.6001 Percent |
| total carbohydrates | Times 53.5433 Gram |
| % daily carbohydrates | Times 17.7103 Percent |
| dietary fiber | Times 6.8463 Gram |
| %daily dietary fiber | Times 25.7301 Percent |
| sugars | Times 5.85251 Gram |
| protein | Times 12.0997 Gram |
| % daily protein | Times 2.7658 Percent |
| vitamin A | Times 13.4342 Percent |
| vitamin C | Times 107.621 Percent |
| calcium | Times 11.9031 Percent |
| iron | Times 19.6752 Percent |

Voilà -- a new, synthetic, NFL generated on the fly from an arbitrary recipe.

# SYMBOLIC COMPUTING FOR REASONING

# SUMMARY

- **The Essence of Symbolic Computing**

Symbolic computing is a general style of computing. Any program that manipulates symbols as opposed to just manipulating numbers is a symbolic program. Symbolic programs are not exotic: you use them every day. Parsers, interpreters, compilers, regular-expression libraries; stream-editing programs like Perl, Sed,

and Awk, macro processors like m4 and t4; templating programs, schema validators, all do symbolic computing. Even arcane uses of symbolic computing such as reasoning over knowledge is completely mundane, being embedded in operating-system kernels and network-security layers.

Though symbolic programs are not exotic, they are specialized; often they are tools as opposed to application-layer solutions. You may use them every day, but you might not write them every day. One message here is to encourage you to use symbolic methods more often in your own programs. Symbolic computing for robust units of measure is a no-brainer. There are many other uses: greatly reducing the size and complexity of programs via pattern-matching and rules processing; reasoning over knowledge; analysis and optimization of queries; dynamic scripting; many more.

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. This is a point that Wolfram Research makes very explicit in their documentation of *Mathematica*, and one of the reasons why we chose to make Jacquard compatible with a subset of *Mathematica*.

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 [23] or metacircular evaluation [24]. 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 an entirely different approach and treat expressions on exactly the same footing as other data. The tools for manipulating data transparently manipulate expressions, *i.e.*, programs. The fundamental unity of code and data is nothing new, it's just not in the daily vocabulary of the ordinary, workaday programmer stuck in a world where programs are always compiled in isolated sessions and executed in other sessions and environments. The gap between analyzing a program and executing a program is unbridgeable. Term-rewriting build the bridge.

### Advantages of Symbolic Computing

Advantage number 1 is in coding applications as transformations of expressions. With term-rewriting in particular, *calculating an answer* means *rewriting expressions*. This separates expressions that represent computations from the expressions that represent data objects. That's the same desideratum that gives rise to the "with" statement of JavaScript. However, the ambiguity introduced via "with" outweighs the advantage of separation of concerns in that case. Jacquard's solution is to invert the code: treat expressions as first-class; treat data as rewrite rules. This inversion allows developers to manipulate 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 instance, to track units of measure, but also to rearrange expressions for optimality, simplicity, or other concerns and remote them for data affinity or privacy. Our sample application includes information in grams, milligrams, ounces, calories, and percentages. It is much too easy to make a mistake like adding milligrams 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.


1 http://www.wolfram.com/mathematica/

2 http://en.wikipedia.org/wiki/Integrated_development_environment

3 http://physics.nist.gov/Pubs/SP811/appenB.html

4 http://www.fda.gov/Food/ResourcesForYou/Consumers/NFLPM/ucm274593.htm

5 http://rewriting.loria.fr/

6 http://www.redditmirror.cc/cache/websites/web.archive.org_84624/web.archive.org/web/20040603192757/research.microsoft.com/research/dtg/davidhov/pap.htm

7 http://secpal.codeplex.com/

8 http://stackoverflow.com/questions/4968406/javascript-property-access-dot-notation-vs-brackets

9 http://en.wikipedia.org/wiki/Don%27t_repeat_yourself

10 http://c2.com/cgi/wiki?DontRepeatYourself

11 http://en.wikipedia.org/wiki/Programming_style#Vertical_alignment

12 http://en.wikipedia.org/wiki/CamelCase

13 http://yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/

14 http://en.wikipedia.org/wiki/Normal_form_(abstract_rewriting)

15 http://en.wikipedia.org/wiki/Domain-specific_language

16 http://en.wikipedia.org/wiki/Mars_Climate_Orbiter

17 http://www.codinghorror.com/blog/2007/05/the-best-code-is-no-code-at-all.html

18 http://answers.yahoo.com/question/index?qid=20060927203122AAv1MpR

19 http://en.wikibooks.org/wiki/Haskell/Understanding_monads

20 http://community.bartdesmet.net/blogs/bart/Default.aspx?PageIndex=2

21 http://en.wikipedia.org/wiki/Fold_(higher-order_function)

22 http://reference.wolfram.com/mathematica/ref/Fold.html?q=Fold&lang=en

23 http://en.wikipedia.org/wiki/Reflection_(computer_programming)

24 http://en.wikipedia.org/wiki/Metacircular_Interpreter