- In addition, some languages permit the addition of datatypes such as matrices.
- Furthermore, many languages support "addition" of strings (we use scare-quotes because we don't really mean the mathematical concept of addition, but rather the operation performed by an operator with the syntax +). In some languages this always means concatenation; in some others, it can result in numeric results (or numbers stored in strings).

These are all different meanings for addition. *Semantics* is the mapping of syntax (e.g., +) to meaning (e.g., some or all of the above).

This brings us to our first game of:

Which of these is the same?

- `1 + 2`
- `1 + 2`
- `'1' + '2'`
- `'1' + '2'`

Now return to the question above. What semantics do we have? We've adopted whatever semantics Racket provides, because we map + to Racket's +. In fact that's not even quite true: Racket may, for all we know, also enable + to apply to strings, so we've chosen the restriction of Racket's semantics to numbers (though in fact Racket's + doesn't tolerate strings).

If we wanted a different semantics, we'd have to implement it explicitly.

Exercise

> What all would you have to change so that the number had signed- 32-bit arithmetic?

In general, we have to be careful about too readily borrowing from the host language. We'll return to this topic later [REF].

3.4 Growing the Language

We've picked a very restricted first language, so there are many ways we can grow it. Some, such as representing data structures and functions, will clearly force us to add new features to the interpreter itself (assuming we don't want to use Gödel numbering). Others, such as adding more of arithmetic itself, can be done without disturbing the core language and hence its interpreter. We'll examine this next (section 4).

4 A First Taste of Desugaring

We've begun with a very spartan arithmetic language. Let's look at how we might extend it with more arithmetic operations that can nevertheless be expressed in terms of existing ones. We'll add just two, because these will suffice to illustrate the point.

4.1 Extension: Binary Subtraction

First, we'll add subtraction. Because our language already has numbers, addition, and multiplication, it's easy to define subtraction: $a - b = a + -1 \times b$.

Okay, that was easy! But now we should turn this into concrete code. To do so, we face a decision: where does this new subtraction operator reside? It is tempting, and perhaps seems natural, to just add one more rule to our existing `ArithC` datatype.

Do Now!

What are the negative consequences of modifying `ArithC`?

This creates a few problems. The first, obvious, one is that we now have to modify all programs that process `ArithC`. So far that's only our interpreter, which is pretty simple, but in a more complex implementation, that could already be a concern. Second, we were trying to add new constructs that we can define in terms of existing ones; it feels slightly self-defeating to do this in a way that isn't modular. Third, and most subtly, there's something *conceptually* wrong about modifying `ArithC`. That's because `ArithC` represents our *core* language. In contrast, subtraction and other additions represent our user-facing, surface language. It's wise to record conceptually different ideas in distinct datatypes, rather than shoehorn them into one. The separation can look a little unwieldy sometimes, but it makes the program much easier for future developers to read and maintain. Besides, for different purposes you might want to layer on different extensions, and separating the core from the surface enables that.

Therefore, we'll define a new datatype to reflect our intended surface syntax terms:

```
(define-type ArithS
  [numS (n : number)]
  [plusS (l : ArithS) (r : ArithS)]
  [bminusS (l : ArithS) (r : ArithS)]
  [multS (l : ArithS) (r : ArithS)])
```

This looks almost exactly like `ArithC`, other than the added case, which follows the familiar recursive pattern.

Given this datatype, we should do two things. First, we should modify our parser to also parse `-` expressions, and always construct `ArithS` terms (rather than any `ArithC` ones). Second, we should implement a `desugar` function that translates `ArithS` values into `ArithC` ones.

Let's write the obvious part of `desugar`:

<*desugar*> **::=**

```
(define (desugar [as : ArithS]) : ArithC
  (type-case ArithS as
    [numS (n) (numC n)]
    [plusS (l r) (plusC (desugar l)
                        (desugar r))]
    [multS (l r) (multC (desugar l)
                        (desugar r))]
    <bminusS-case>))
```

Now let's convert the mathematical description of subtraction above into code:

<bminusS-case> **::=**

```
[bminusS (l r) (plusC (desugar l)
                      (multC (numC -1) (desugar r)))]
```

Do Now!

It's a common mistake to forget the recursive calls to `desugar` on `l` and `r`. What happens when you forget them? Try for yourself and see.

4.2 Extension: Unary Negation

Now let's consider another extension, which is a little more interesting: unary negation. This forces you to do a little more work in the parser because, depending on your surface syntax, you may need to look ahead to determine whether you're in the unary or binary case. But that's not even the interesting part!

There are many ways we can desugar unary negation. We can define it naturally as $-b = 0 - b$, or we could abstract over the desugaring of binary subtraction with this expansion: $-b = 0 + -1 \times b$.

Do Now!

Which one do you prefer? Why?

It's tempting to pick the first expansion, because it's much simpler. Imagine we've extended the `ArithS` datatype with a representation of unary negation:

```
[uminusS (e : ArithS)]
```

Now the implementation in `desugar` is straightforward:

```
[uminusS (e) (desugar (bminusS (numS 0) e))]
```

Let's make sure the types match up. Observe that `e` is a `ArithS` term, so it is valid to use as an argument to `bminusS`, and the entire term can legally be passed to `desugar`. It is therefore important to *not* desugar `e` but rather embed it directly in the generated term. This embedding of an input term in another one and recursively calling desugar is a common pattern in desugaring tools; it is called a *macro* (specifically, the "macro" here is this definition of `uminusS`).

However, there are two problems with the definition above:

1. The first is that the recursion is *generative*, which forces us to take extra care. We might be tempted to fix this by using a different rewrite:

   ```
   [uminusS (e) (bminusS (numS 0) (desugar e))]
   ```

   which does indeed eliminate the generativity.

   Do Now!

If you haven't heard of generative recursion before, read the section on it in *How to Design Programs*. Essentially, in generative recursion the sub-problem is a computed function of the input, rather than a structural piece of it. This is an especially simple case of generative recursion, because the "function" is simple: it's just the `bminusS` constructor.

Unfortunately, this desguaring transformation won't work at all! Do you see why? If you don't, try to run it.

2. The second is that we are implicitly depending on exactly what `bminusS` means; if its meaning changes, so will that of `uminusS`, even if we don't want it to. In contrast, defining a functional abstraction that consumes two terms and generates one representing the addition of the first to -1 times the second, and using this to define the desugaring of both `uminusS` and `bminusS`, is a little more fault-tolerant.

   You might say that the meaning of subtraction is never going to change, so why bother? Yes and no. Yes, it's *meaning* is unlikely to change; but no, its *implementation* might. For instance, the developer may decide to log all uses of binary subtraction. In the macro expansion, all uses of unary negation would also get logged, but they would not in the second expansion.

Fortunately, in this particular case we have a much simpler option, which is to define $-b = -1 \times b$. This expansion works with the primitives we have, and follows structural recursion. The reason we took the above detour, however, is to alert you to these problems, and warn that you might not always be so fortunate.

5 Adding Functions to the Language

Let's start turning this into a real programming language. We could add intermediate features such as conditionals, but to do almost anything interesting we're going to need functions or their moral equivalent, so let's get to it.

Exercise

Add conditionals to your language. You can either add boolean datatypes or, if you want to do something quicker, add a conditional that treats `0` as false and everything else as true.

What are the important test cases you should write?

Imagine, therefore, that we're modeling a system like DrRacket. The developer defines functions in the definitions window, and uses them in the interactions window. For now, let's assume all definitions go in the definitions window only (we'll relax this soon [REF]), and all expressions in the interactions window only. Thus, running a program simply loads definitions. Because our interpreter corresponds to the interactions window prompt, we'll therefore assume it is supplied with a set of definitions.

A *set* of definitions suggests no ordering, which means, presumably, any definition can refer to any other. That's what I intend here, but when you are designing your own language, be sure to think about this.

5.1 Defining Data Representations

To keep things simple, let's just consider functions of one argument. Here are some Racket examples:

```
(define (double x) (+ x x))
```