word "conservative" has, however, become a term-of-art to refer to a GC technique that operates in an *uncooperative* (and hopefully not *hostile*) run-time system.

11.6 Precise Garbage Collection

In conventional GC terminology, the opposite of "conservative" is *precise*. This, too, is a misnomer, because a GC cannot be precise, i.e., both sound and complete. Rather, precision here is a statement about the ability to identify references: when confronted with a value, a precise GC knows exactly what is and isn't a reference, and where the references are. This removes the monumental effort that a conservative GC has to put into guessing non-references (and hoping to eliminate as many potential references as possible through this process).

Within the space of precise GC, which is what most contemporary language run-time systems use, there is a wide range of implementation techniques. I refer you to Paul Wilson's survey (which, despite its relative age in this fast-moving field, remains an excellent resource), as well as the book and other materials from Richard Jones. In particular, for a quick and readable overview of a generational garbage collector, read *Simple Generational Garbage Collection and Fast Allocation*.

12 Representation Decisions

Go back and look again at our interpreter for function as values [REF]. Do you see something curiously non-uniform about it?

Do Now!

No, really, do. Do you?

Consider how we chose to represent our two different kinds of values: numbers and functions. Ignoring the superficial `numV` and `closV` wrappers, focus on the underlying data representations. We represented the interpreted language's numbers as Racket numbers, but we did not represent the interpreted language's functions (closures) as Racket functions (closures).

That's our non-uniformity. It would have been more uniform to use Racket's representations for both, or also to *not* use Racket's representation for either. So why did we make this particular choice?

We were trying to illustrate and point, and that point is what we will explore right now.

12.1 Changing Representations

For a moment, let's explore numbers. Racket's numbers make a good target for reuse because they are so powerful: we get arbitrary-sized integers (*bignums*), rationals (which benefit from the bignum representation of integers), complex numbers, and so on. Therefore, they can represent most ordinary programming language number systems. However, that doesn't mean they are what we *want*: they could be too little or too much.

- They are too much if what we want is a more restricted number system. For instance, Java prescribes a very specific set of fixed-size representations (e.g., `int` is specified to be 32-bit). Numbers that fall outside these sets cannot be directly represented as numbers, and arithmetic must respect these sets (e.g., overflowing so that adding `1` to `2147483647` does *not* produce `2147483648`).

- They are too little if we want even richer numbers, whether quaternions or numbers with associated probabilities.

Worse, we didn't even stop and ask what we wanted, but blithely proceeded with Racket numbers.

The reason we did so is because we weren't really interested in the study of numbers; rather, we were interested in programming language features such as functions-as-values. As language designers, however, you should be sure to ask these hard questions up front.

Now let's talk about our representation of closures. We could have instead represented closures by exploiting Racket's corresponding concept, and correspondingly, function application with unvarnished Racket application.

Do Now!

Replace the closure data structure with Racket functions representing functions-as-values.

Here we go:

```
(define-type Value
  [numV (n : number)]
  [closV (f : (Value -> Value))])

(define (interp [expr : ExprC] [env : Env]) : Value
  (type-case ExprC expr
    [numC (n) (numV n)]
    [idC (n) (lookup n env)]
    [appC (f a) (local ([define f-value (interp f env)]
                        [define a-value (interp a env)])
                  ((closV-f f-value) a-value))]
    [plusC (l r) (num+ (interp l env) (interp r env))]
    [multC (l r) (num* (interp l env) (interp r env))]
    [lamC (a b) (closV (lambda (arg-val)
                         (interp b
                                 (extend-env (bind a arg-val)
                                             env))))]))
```

Exercise

Observe a curious shift. In our previous implementation, the environment was extended in the `appC` case. Here, it's extended in the `lamC` case. Is one of these incorrect? If not, why did this change occur?

This is certainly concise, but we've lost something very important: *understanding*. Saying that a source language function corresponds to `lambda` tells us virtually nothing: if we already knew precisely what `lambda` does we might not be studying it, and if we didn't, this mapping would teach us absolutely nothing (and might, in fact, pile confusion on top of our ignorance). For the same reason, we did not use Racket's state to understand the varieties of stateful operators [REF].

Once we've understood the feature, however, we should feel to use it as a representation. Indeed, doing so might yield much more concise interpreters because we aren't doing everything manually. In fact, some later interpreters [REF] will become virtually unreadable if we did not exploit these richer representations. Nevertheless, exploiting host language features has perils that we should safeguard against.

It's a little like saying, "Now that we understand addition in terms of increment-by-one, we can use addition to define multiplication: we don't have to use only increment-by-one to define it."

12.2 Errors

When programs go wrong, programmers need a careful presentation of errors. Using host language features runs the risk that users will see host language errors, which they will not understand. Therefore, we have to carefully translate error conditions into terms that the user of our language will understand, without letting the host language "leak through".

Worse, programs that should error might not! For instance, suppose we decide that functions should only appear in top-level positions. If we fail to expressly check for this, desugaring into the more permissive `lambda` may result in an interpreter that produces answers where it should have halted with an error. Therefore, we have to take great care to permit *only the intended surface language to be mapped to the host language*.

As another example, consider the different mutation operations. In our language, attempting to mutate an unbound variable produces an error. In some languages, doing so results in the variable being defined. Failing to pin down our intended semantics is a common language designer error, saying instead, "It is whatever the implementation does". This attitude (a) is lazy and sloppy, (b) may yield unexpected and negative consequences, and (c) makes it hard for you to move your language from one implementation platform to another. Don't ever make this mistake!

12.3 Changing Meaning

Mapping functions-as-values to `lambda` works especially because we intend for the two to *have the same meaning*. However, this makes it difficult to change the meaning of what a function does. Lemme give ya' a hypothetic: suppose we wanted our language to implement dynamic scope. In our original interpreter, this was easy (almost too easy, as history shows). But try to make the interpreter that uses `lambda` implement dynamic scope. It can similarly be difficult or at least subtle to map eager evaluation onto a language with lazy application [REF].

Don't let this go past the hypothetical stage, please.

Exercise

Convert the above interpreter to use dynamic scope.

The point is that the raw data structure representation does not make anything especially easy, but it usually doesn't get in the way, either. In contrast, mapping to host language features can make some intents—mainly, those match what the host language already does!—especially easy, and others subtle or difficult. There is the added danger that we may not be certain of what the host language's feature does (e.g., does its "lambda" actually implement static scope?).

The moral is that this is a good property to exploit only we want to "pass through" the base language's meaning—-and then it is especially wise because it ensures that we don't accidentally change its meaning. If, however, we want to exploit a significant part of the base language and only augment its meaning, perhaps other implementation strategies [REF] will work just as well instead of writing an interpreter.

12.4 One More Example

Let's consider one more representation change. What is an environment?

An environment is a *map* from names to values (or locations, once we have mutation). We've chosen to implement the mapping through a data structure, but...do we have another way to represent maps? As functions, of course! An environment, then, is a function that takes a name as an argument and return its bound value (or an error):

```
(define-type-alias Env (symbol -> Value))
```

What is the empty environment? It's the one that returns an error no matter what name you try to look up:

```
(define (mt-env [name : symbol])
  (error 'lookup "name not found"))
```

(In principle we should put a type annotation on the return, and it should be `Value`, except of course thiis is vacuous.) Extending an environment with a binding creates a function that takes a name and checks whether it is the name just extended; if so it returns the corresponding value, otherwise it punts to the environment being extended:

```
(define (extend-env [b : Binding] [e : Env])
  (lambda ([name : symbol]) : Value
    (if (symbol=? name (bind-name b))
        (bind-val b)
        (lookup name e))))
```

Finally, how do we look up a name in an environment? We simply *apply* the environment!

```
(define (lookup [n : symbol] [e : Env]) : Value
  (e n))
```

And with that, we're done!