The Ambiguous Choice Operator and Continuations

In the previous lectures, we used macros to create an implementation of streams, a lazy data structure that decouples the creation of data from the consumption of that data.

In this section, we'll explore this idea in a different context. In particular, we will create data with expressions that use *non-deterministic choice*, and consume data using the function `next!` that we previously wrote . Our implementation of such expressions will introduce one new technical concept in programming language theory: the **continuation**, a representation of the control flow at a given point in the execution of a program.

The choice operator `-<`

The operator `-<` is pronounced "amb", short for the word "ambiguous". We will use this operator to *generate data* and to denote *possible choices* of values. The intention is that any place in our code where a single expression is used, we would like to instead provide a number of possible choices of values.

This operator therefore takes an arbitrary number of argument subexpressions, representing possible values for the whole `-<` expression. It then returns a *stream* of potential answers. We access the stream using the `next!` function that we previously defined. Here's a very simple example.

```
> (define g (-< 1 2 (+ 3 4))) ; Create a stream of potential answers
> (next! g) ; A call to (next!) returns an answer, and updates the stream
1
> (next! g)
2
> (next! g)
7
> (next! g) ; Here our constant 'DONE shows that there are no more choices
'DONE
```

The macro `next!` is what we have previously defined. We will need to make some changes to this macro later on, but let's start with the definition that we had previously:

```
(define-syntax next!
  (syntax-rules ()
    [(next! <g>)
     (if (s-null? <g>)
         'DONE
         (let* ([tmp <g>])
           (begin
             (set! <g> (s-rest <g>))
             (s-first tmp))))]))
```

What about the `-<` operator? So far, it sounds exactly like the `make-stream` macro that we wrote earlier, which is not very exciting. But we want `-<` to do a little more work: we would like the `-<` operator to be able to *capture the surrounding computation*.

As an example, we would like this expression:

```
(+ 10 (-< 1 2 (+ 3 4)))
```

. . . to create a stream containing the values 11, 12, and 17. In order to support this new functionality, we need to do something else. In particular, we need some way of saving the *computational context* surrounding the call to `-<`. That is, we need to be able to store the fact that the computation `(lambda (x) (+ 10 x))`, also written as `(+ 10 _)` needs to be performed *after* the call to `-<`. This brings us to the realm of continuations.

Continuation (from Prof. David Liu's notes)

In our studies to date, we have largely been passive participants in the operational semantics of our languages, subject to the evaluation orders of function calls and special syntactic forms. Even with macros, which do allow us to manipulate evaluation order, we have done so only by rewriting expressions into a fixed set of built-in macros and function calls. However, Racket has a data structure to directly represent (and hence manipulate) control flow from within a program: the continuation. Consider the following simple expression:

```
(+ (* 3 4) (first (list 1 2 3)))
```

By now, you should have a very clear sense of how this expression is evaluated, and in particular the *order in which* the subexpressions are evaluated.[1] For each subexpression s, we define its **continuation** to be a representation of what remains to be evaluated after s itself has been evaluated. Put another way, the continuation of s is a representation of the control flow from immediately after s has been evaluated, up to the final evaluation of the enclosing top-level expression (or enclosing continuation delimiter, a topic we'll explore later this chapter). For example, the continuation of the subexpression `(* 3 4)` is, in English, "evaluate `(first (list 1 2 3))` and then add it to the result." We represent this symbolically using an underscore to represent the "hole" in the remaining expression: `(+ _ (first (list 1 2 3)))`.[2] While it might seem like we can determine continuations simply by literally replacing a subexpression with an underscore, this isn't exactly true. The continuation of the subexpression `(first (list 1 2 3))` is `(+ 12 _)`; because of left-to-right evaluation order in function calls, the `(* 3 4)` is evaluated *before* the call to `first`, and so the continuation of the latter call is simply "add `12` to the result."

Since both identifiers and literal values are themselves expressions, they too have continuations. The continuation of 4 in the above expression is `(+ (* 3 _) (first (list 1 2 3)))`, and the continuation of `first` is `(+ 12 (_ (list 1 2 3)))`. Finally, the continuation of the whole expression `(+ (* 3 4) (first (list 1 2 3)))` is simply "return the value," which we represent simply as `_`.

Warning: students often think that an expression's continuation includes the evaluation of the expression itself, which is incorrect—an expression's continuation represents only the control flow *after* its evaluation. This is why we replace the expression with an underscore in the above representation! An easy way to remember this is that an expression's continuation doesn't depend on the expression itself, but rather the expression's position in the larger program.

`shift`: reifying continuations (from Prof. David Liu's notes)

So far, continuations sound like a rather abstract representation of control flow. What's quite spectacular about Racket is that it it provides ways to access continuations during the execution of a program. We say that Racket *reifies* continuations, meaning it exposes continuations as values that a program can access and manipulate as easily as numbers and lists. We as humans can read a whole expression and determine the continuations for each of its subexpressions (like we did in the previous section). How can we write a *program* that does this kind of meta-analysis for us?

Though it turns out to be quite possible to implement reified continuations purely in terms of functions, this is beyond the scope of the course.[3] Instead, we will use Racket's syntactic form `shift` to capture continuations for us.

Note: `shift` is imported from `racket/control`; include `(require racket/control)` for all code examples in this section.

Here is the relevant syntax pattern:

```
(shift <id> <body> ...)
```

A `shift` expression has the following denotational semantics:

1. The current continuation of the `shift` expression is bound to `<id>`. By convention, we often use `k` for the `<id>`.
2. The `<body> ...` is evaluated (with `<id>` in scope).
3. The current continuation is **discarded**, and the value returned is simply the value of the last expression in `<body> ...`.

[1] For example, the fact that `(* 3 4)` is evaluated before the name `first` is looked up in the global environment.
[2] We can also represent the continuation as a unary function `(lambda (x) (+ x (first (list 1 2 3))))`.
[3] Those interested should look up *continuation-passing style*.

The first two points are pretty straightforward (the binding of the name and evaluation of an expression); the third point is the most surprising, as it causes the value of the `shift` expression to "escape" any enclosing expressions.[4] Let's start just by illustrating the second and third points, without worrying about the continuation binding.

```
> (shift k (+ 4 5))  ; Ignore the k; the body of the shift is evaluated and returned.
9
> (* 100 (shift k (+ 4 5)))  ; The shift's continuation, (* 100 _), is discarded!
9
```

Now let's return to point 1 above; the identifier `k` is bound to the `shift` expressions continuation. In the first example, this was simply the identity continuation ("return the value"), and in the second example, this was the continuation `(* 100 _)`. Of course, just binding the continuation to the name `k` is not very useful. Moreover, `k` is local to the `shift` expression, so we cannot refer to it after the expression has been evaluated. We'll use mutation to save the stored continuation.[5] Note that as in the previous section, `shift` can take multiple `<body>` expressions, evaluating each one and returning the value of the last one.

```
> (define saved-cont (void))
> (* 100
     (shift k
            ; Store the bound continuation in the global variable.
            (set! saved-cont k)
            ; Evaluate and return this last expression.
            (+ 4 5)))
9
```

As expected, when we evaluate the expression, the `(* 100 _)` is discarded, and the `(+ 4 5)` is evaluated and returns. But now we have a global variable storing the bound continuation—let's check it out.

```
> saved-cont
#<procedure>
```

Racket stored the continuation `(* 100 _)` as the unary function `(lambda (x) (* 100 x))`—and we can now call it, just as we would any other function.

```
> (saved-cont 9)
900
```

Pretty cool! But it seems like `shift` did something we didn't want: discard the current continuation. It would be great if we could both store the continuation for future use, but also *not* interrupt the regular control flow, so that `shift` behaves like other kinds of Racket expressions. This turns out to be quite straightfoward, once we remember that we have access to the current continuation right in the body of the `shift`!

```
> (* 100
     (shift k
            ; Store the bound continuation in the global variable.
            (set! saved-cont k)
            ; Call the current continuation k on the desired argument
            (k (+ 4 5))))
900
```

`reset`: delimiting continuations

When a `shift` expression is used, it captures the **entire** continuation, or the entire rest of the computation to be performed after the expression. Sometimes, this is the intended behaviour.

```
> (+ 2 (shift k (* (k 3) (k 4))))
30
```

However, using `shift` by itself can produce results that are strange:

[4] This is similar to using a `return` deep inside a function body, or `raise`/`throw` for exceptions.

[5] You might have some fun thinking about how to approach this and the rest of the chapter without using any mutation.

```
> (define a (+ 2 (shift k (* (k 3) (k 4)))))
*: contract violation
  expected: number?
  given: #<void>
  argument position: 1st
  other arguments...:
```

What happened? The problem is that `shift` is *too powerful*. It is not only capturing the continuation `(+ 2 _)`, but also computation that needs to be done as part of the `define`! This is why `(k 3)` produced a `#<void>` value rather than a number.

We can fix the issue by placing a delimiter that `shift` is not allowed to go beyond. This delimiter is called `reset`. By placing the expression `(+ 2 (shift k (* (k 3) (k 4))))` inside a call to `reset`, we prevent computations related to the `define` from be captured in the continuation.

```
> (define a (reset (+ 2 (shift k (* (k 3) (k 4))))))
30
```

Applying continuations to -<

Recall that our motivation for learning about continuations was to implement a version of the amb operator `-<`. We would like this operator to store not only a stream of choices, but also apply the computational context around `-<`. We now have a name for that computational context: it's just the continuation of the `-<` expression!

Here is the first attempt at defining the operator `-<`. For reasons that will become clear later, we will make `-<` a *function*. This function will call a function `map-stream`, which creates a stream that applies the continuation `k` to every one of the argument to `-<`.

Here's a first attempt:

```
(define (-< . lst)
  (shift k (map-stream k lst)))

(define (map-stream k lst) ; create a stream that applies k to every item in lst
  (if (empty? lst)
      s-null
      (s-cons (k (first lst))              ; apply k to first item of lst
              (map-stream k (rest lst))))) ; apply k to the rest of the elements
```

Now, our call to `-<` captures the surrounding continuation (up to the nearest `reset`)!

```
> (define g (reset (+ 1 (-< 10 20)))) ; `k` is (+ 1 _)
> (next! g)   ; g is (s-cons (k 10) (map-stream k '(20)))
11
> (next! g)   ; g is (s-cons (k 20) (map-stream k '()))
21
> (next! g)
'DONE
```

It is rather amazing that such a small change unlocks a huge number of possibilities for our macro; this truly illustrates the power of continuations and macros in Racket.

Branching Choices

The syntax `-<` that we created in the previous section is interesting, but is not enough. The original intention behind the `-<` operator is that it should represent a number of possible *choices of values*, wherever a single value is expected. However, as soon as we have two choice points, we run into some issues:

```
> (define g (reset (+ (-< 10 20) (-< 2 3))))
> (next! g)
'(#<procedure> .  #<procedure>)
```

We would hope to see a single value here, but instead we see a stream. The reason is that each `-<` creates a stream, and so attempting to add them together creates some unintended behaviour. Here's a partial trace of what's happening:

```
    (reset (+ (-< 10 20) (-< 2 3)))
==> (shift k (map-stream k '(10 20))) ; where k = (+ _ (-< 2 3))
==> (s-cons ((+ _ (-< 2 3)) 10) (map-stream k '(20)))
```

The problem is that when `((+ _ (-< 2 3)) 10)` is evaluated, `(+ 10 (-< 2 3))` returns a stream. So, the first element of our stream is also going to be a stream!

The way around the issue is actually quite interesting, and maybe not all that intuitive, but actually quite useful. Instead of assuming that `(k x)` returns a single result, we will assume that `(k x)` returns a *stream of results*. So `map-stream` needs to *appends together a list of streams*! Or rather, we need a new function `s-append-map`, analogous to Racket's `append-map` function.

```
(define (-< . lst)
  (shift k (s-append-map k lst)))
```

Now, we have introduced two problems. First, we need to actually write the function `s-append-map`. Second, the simple example `(reset (+ 1 (-< 10 20)))` from earlier will no longer work, because `(k x)` will return a single value.

This second problem is actually not that difficult to solve, so let's start there.

Have `(k x)` return a stream

The way we can ensure that `(k x)` returns a stream is to *change the continuation* `k`. That is, we'll make sure that the *last* thing done in `k` is to wrap the result into a stream. We'll introduce a function `singleton` that makes a stream with a single element, and instead of defining a stream `g` to be `(reset (+ 1 (-< 10 20)))`, we will define `g` to be `(reset (singleton (+ 1 (-< 10 20))))`. That way, the continuation of the call to `-<` would be `(singleton (+ 1 _))`.

Here's how we can define `singleton`:

```
(define (singleton x) (make-stream x))
```

You might be tempted to use `make-stream` directly, instead of defining the function `singleton`. Unfortunately, because macro expansion happens *first*, using `make-stream` instead of `singleton` would result in the following expansion

```
    (reset (make-stream (+ 1 (-< 10 20)))) ; macro matches pattern (make-stream <first> <rest> ...)
==> (reset (s-cons (+ 1 (-< 10 20)) (make-stream))) ; the make-stream macro is expanded first
```

Wrapping `(reset (singleton _))` around expressions that use `-<` is a little tedious to remember, so let's write a macro called `do/-<` instead:

```
(define-syntax do/-<
  (syntax-rules ()
     [(do/-< <expr>) (reset (singleton <expr>))]))
```

Any expressions that use `-<` will need to be wrapped inside a `do/-<`, like this:

```
> (do/-< (+ (-< 10 20) (-< 2 3)))
```

Defining `s-append-map`

Now that `(k x)` returns a stream, we can move on to implementing `s-append-map`. This function will take two parameters: a continuation `k` and a list `lst`. It will apply `k` to each item in `lst`, and append the resulting streams together.

To keep the evaluation lazy, we won't actually perform the computation in that order. Instead, we will define this function recursively. We will also need a helper function that will append two streams at a time:

```
#|
(s-append-map k lst)
  k: A continuation
  lst: A list

  Returns a stream containing all the elements in each of the
  streams (k x), for every x in lst.
|#
(define (s-append-map k lst)    ; k is a function/continuation
  (if (empty? lst)
      s-null                    ; if lst is empty, return an empty stream
      (s-append (k (first lst)) ; otherwise, append together two streams
                (thunk (s-append-map k (rest lst))))))
```

The function `s-append` will help us append together two streams. Actually, the first parameter of `s-append` will be a stream, and the second parameter will be a *thunk* that evaluates to a stream. The reason we wrap the second parameter in a thunk is for lazy evaluation. If we don't do this, we won't have the error isolation behaviour from earlier.

Defining `s-append`

The function `s-append` take two arguments: the first stream `<s>` and a thunk containing a second stream. The implementation of `s-append` is mostly straightforward, and follows the recursive structure of the stream: If the first stream `s` is empty, then call the thunk `t`. Otherwise, create a new thunk with the first element of `s`, and the combined stream containing the rest of `s` along with the values in `t`.

```
(define (s-append s t)
  (cond
    [(s-null? s) (t)]
    [(pair? s)   (s-cons (s-first s)
                         (s-append (s-rest s) t))]))
```

Putting it together (without error handling)

Now, we can put all these definitions together, and revisit the examples from earlier.

```
; Every expression that uses `-<` should be wrapped with a `do/-<` as a delimiter
(define-syntax do/-<
  (syntax-rules ()
     [(do/-< <expr>) (reset (singleton <expr>))]))

; The ambiguous choice operator
(define (-< . lst)
  (shift k (s-append-map k lst)))

; Helper function to create a stream with a single element
(define (singleton x) (make-stream x))

#|
(s-append-map k lst)
  k: A continuation
  lst: A list

  Returns a stream containing all the elements in each of the
  streams (k x), for every x in lst.
```

```
|#
(define (s-append-map k lst)
  (if (empty? lst)
      s-null
      (s-append (k (first lst)) (thunk (s-append-map k (rest lst))))))

; Helper function to append a stream, and a thunk that will evaluate to a stream
(define (s-append s t)
  (cond
    [(s-null? s) (t)]
    [(pair? s)   (s-cons (s-first s) (s-append (s-rest s) t))]))
```

Trace through example with a single -<

Let's see whether the simple examples still work:

```
> (define g (do/-< (+ 10 (-< 2 3))))
> (next! g)
12
> (next! g)
13
> (next! g)
'DONE
```

It does! The trace through this code is actually not much more complex than before:

```
    (do/-< (+ 10 (-< 2 3)))
==> (thunk (reset (singleton (+ 10 (-< 2 3)))))
```

When called, the thunk evaluates to:

```
==> (s-append-map k '(2 3))                    ; where k = (singleton (+ 10 _)) throughout
==> (s-append (k 2) (thunk (s-append-map k '(3))))
==> (s-cons (s-first s) (s-append (s-rest s) (thunk (s-append-map k '(3)))))
       ; where s = (k 2) = (singleton (+ 10 2)) = (cons (thunk 12) (thunk s-null))
```

Calling `s-first` on the above stream will result in the value 12, and the stream will self-update to:

```
==> (thunk (s-append (s-rest s) (thunk (s-append-map k '(3))))) ; where k = (singleton (+ 10 _)) throughout
```

Calling that thunk will perform the computation:

```
==> (s-append-map k '(3))                                    ; since (s-rest s) is the empty stream s-nu
==> (s-append (k 3) (thunk (s-append-map k '())))
==> (s-cons (s-first s2) (s-append (s-rest s2) (thunk (s-append-map k '()))))
        ; where s2 = (k 3) = (singleton (+ 10 3)) = (cons (thunk 13) (thunk s-null))
```

Calling `s-first` on the above stream will result in the value 13. The stream will self-update to a thunk that returns s-null.

(You might have noticed one annoying behaviour in our implementation: the computation of the following element happens *before* the call to `next!`. We'll come back to this issue later.)

Trace through example with multiple -<

Now, let's try another example with multiple uses of `-<`:

```
> (define g (do/-< (+ (-< 10 20) (-< 2 3))))
> (next! g)
12
> (next! g)
13
> (next! g)
```

```
22
> (next! g)
23
> (next! g)
'DONE
```

Interestingly, using multiple `-<` also works! This may seem surprising, since all we did was to wrap elements into streams. Let's trace through this example so that we understand what's going on.

```
    (do/-< (+ (-< 10 20) (-< 2 3)))
==> (thunk (reset (singleton (+ (-< 10 20) (-< 2 3)))))
```

When this thunk is evaluated, the first thing that gets evaluated is the first `-<`. The second `-<` actually becomes a part of its continuation!

```
==> (s-append-map k '(10 20))   ; where k = (singleton (+ _ (-< 2 3))) throughout
==> (s-append (k 10) (thunk (s-append-mal k '(20))))
==> (s-cons (s-first s) (s-append (s-rest s) (thunk (s-append-map k '(20)))))
    ; where s = (k 10)
    ;         = (singleton (+ 10 (-< 2 3)))
```

In order to continue evaluating this expression, we need to first evaluate `(singleton (+ 10 (-< 2 3)))`. But we actually performed this evaluation earlier, and saw that this expression produced the stream containing 12 and 13! Let's not repeat the trace, and jump immediately to the value of `s`:

```
==> (s-cons (s-first s) (s-append (s-rest s) (thunk (s-append-map k '(20)))))
    ; where s = (cons (thunk 12) (thunk (s-append (s-rest s2) (thunk (s-append-map k2 '(3))))))
    ;             with k2 = (singleton (thunk (+ 10 _)))
    ;             and  s2 = (k2 2) = (cons (thunk (+ 10 2)) (thunk s-null))
    ;    in other words, s is the stream containing the elements 12 and 13
```

When we call `next!`, we will evaluate `((car s))`, and obtain:

```
==> ((car s))
==> ((car (cons (thunk (+ 10 2)) ...)))
==> ((thunk (+ 10 2))
==> (+ 10 2)
==> 12
```

. . . and the stream will self-update to:

```
==> ((cdr s))
==> (s-append (s-rest s) (thunk (s-append-map k '(20))))  ;s, s2, and k2 is the same value as above
==> (s-append (s-append (s-rest s2) (thunk (s-append-map k2 '(3)))) (thunk (s-append-map k '(20))))
==> (s-append (s-append s-null (thunk (s-append-map k2 '(3)))) (thunk (s-append-map k '(20))))
==> (s-append (s-append-map k2 '(3)) (thunk (s-append-map k '(20))))
==> (s-append (s-append (k2 3) (thunk (s-append-map k2 '()))) (thunk (s-append-map k '(20))))
==> (s-append (cons (thunk (+ 10 3)) ...) (thunk (s-append-map k '(20))))
==> (s-cons (s-first s3) (s-append (cdr s3) (thunk (s-append-map k '(20)))))
    ; where s3 = (cons (thunk (+ 10 3)) ... )
```

So, the expression `(do/-< (+ (-< 10 20) (-< 2 3)))` produces a stream containing the values 12, 13, 22 and 23.

Isolating Errors

One annoying behaviour in our implementation is that the computation of the next element of the stream happens *before* the call to `next!`. For example, if we create a stream with an invalid third element, the error is reported in the **second** call to `next!`:

```
> (define g (do/-< (/ 1 (-< 2 1 0 4))))
> (next! g)
1/2
```

```
> (next! g)        ; error is reported here
; /: division by zero [,bt for context]
1
> (next! g)        ; error should happen here, and we should be able to recover
; car: contract violation
;   expected: pair?
;   given: #<void>
; [,bt for context]
```

Additionally, we don't actually recover from the error! Additional calls to `(next! g)` produce the same `contract violation` error.

One way to work around the issue is to **wrap the entire computation in a thunk**. That is, we update `do/-<` to return a thunk instead of a stream.

```
; Every expression that uses `-<` should be wrapped with a `do/-<` as a delimiter
(define-syntax do/-<
  (syntax-rules ()
     [(do/-< <expr>) (thunk (reset (singleton <expr>)))])) ; wrap the result in a thunk
```

We also need to change the signature for `next!` so that `next!` takes as argument a thunk that evaluates to a stream.

```
(define-syntax next!
  (syntax-rules ()
    [(next! <g>)  ; in this version of next!, <g> is a thunk that evaluates to a stream
     (let* ([stream (<g>)]) ; first, evaluate <g>
       (if (s-null? stream)
           'DONE
           (begin
               (set! <g> (cdr stream)) ; use cdr rather than s-rest
               (s-first stream))))]))
```

Finally, we add an extra case to `s-append` to handle the case where `s` is an error.

```
(define (s-append s t)
  (cond
    [(s-null? s) (t)]
    [(pair? s)   (s-cons (s-first s) (s-append (s-rest s) t))]
    [else        (s-cons s (t))])) ; for error handling; s is a #<void> value
```

Now, errors are isolated as we expect

```
> (define g (do/-< (/ 1 (-< 2 1 0 4))))
> (next! g)
1/2
> (next! g)
1
> (next! g)
; /: division by zero [,bt for context]
> (next! g)
1/4
> (next! g)
'DONE
```

Our final set of definitions is as follows:

```
#|
(next! <g>)
  <g>: A stream

  Returns the first element of the stream. Then, mutate the stream
  value to the rest of the stream. (This is version 2 of next! from
```

```
  the posted notes)
|#
(define-syntax next!
  (syntax-rules ()
    [(next! <g>)  ; in this version of next!, <g> is a thunk that evaluates to a stream
     (let* ([stream (<g>)]) ; first, evaluate <g>
       (if (s-null? stream)
           'DONE
           (begin
               (set! <g> (cdr stream)) ; use cdr rather than s-rest
               (s-first stream))))]))

; Every expression that uses `-<` should be wrapped with a `do/-<` as a
; delimiter
(define-syntax do/-<
  (syntax-rules ()
     [(do/-< <expr>) (thunk (reset (singleton <expr>)))]))

; Helper function to create a stream with a single element
(define (singleton x) (make-stream x))

; The ambiguous choice operator
(define (-< . lst)
  (shift k (s-append-map k lst)))

; Append together the stream s, and a thunk t that produces a second stream
(define (s-append s t)
  (cond
    [(s-null? s) (t)]
    [(pair? s)   (s-cons (s-first s) (s-append (s-rest s) t))]
    [else        (s-cons s (t))]))

; Apply k to every item of lst, then append together the resulting streams
(define (s-append-map k lst)
  (if (empty? lst)
      s-null
      (s-append (k (first lst))
                (thunk (s-append-map k (rest lst))))))
```

Towards declarative programming (from Prof. David Liu's notes)

Even though we hope you find our work for the choice macro intellectually stimulating in its own right, you should know that this truly is a powerful mechanism that can change the way you write programs. Just as how in our discussion of streams we saw how thunks could be used to decouple the production and consumption of linear data, our choice library allows this decoupling to occur even when how we produce data is non-linear.

So far in this course, we have contrasted functional programming with imperative programming, with one of the main points of contrast being writing code that is more descriptive of the computation you want the computer to perform, rather than describing *how* to do it—that is, writing programs that are more *declarative* in nature

(In this course, we take the view of declarative as being a spectrum rather than a yes/no binary characteristic. So we won't say that a language or style *is* declarative, but rather that it's more declarative than what we're used to.)

For example, here is a simple expression that generates all possible binary strings of length 5, following the English expression "take five characters, each of which is a 0 or 1":i(Note that Racket uses the `#\` to represent a character literal.)

```
> (define g (do/-< (string (-< #\0 #\1)
```

```
                          (-< #\0 #\1)
                          (-< #\0 #\1)
                          (-< #\0 #\1)
                          (-< #\0 #\1))))
> (next g)
"00000"
> (next g)
"00001"
> (next g)
"00010"
> (next g)
"00011"
> (next g)
"00100"
```

We can do the same thing with an arbitrary list of strings. In fact, since `-<` is a function, we can use `apply` to provide a (Racket) list of options:

```
> (define options (list #\a #\b #\c))
> (define g (do/-< (string (apply -< options)
                           (apply -< options)
                           (apply -< options))))
> (next! g)
"aaa"
> (next! g)
"aab"
> (next! g)
"aac"
> (next! g)
"aba"
```

This easy composability is quite remarkable, and is arguably as close to a human English description as possible. Though we can achieve analogous behaviour with plain higher-order list functions (or nested loops), the simplicity of this approach is quite beautiful.

Acknowledgements

Portions of these notes are derived from the CSC324 Notes by Prof. David Liu. The portions that are directly duplicated are indicated. Other portions are still borrowed heavily from Prof. Liu's work. https://www.cs.toronto.edu/~david/