(Provisional) Lecture 19: Combinatorics and more Ocaml
10:00 AM, Oct 20, 2017

Contents

1 Stars and Stripes 1
   1.1 Combinatorial Problems: a generic approach for the simple ones 2
   1.2 Breaking up stars and stripes 3
   1.3 Analysis of Stars and Stripes 4

2 Bignum 4

3 Procedure Definitions 5

4 Multi-parameter Procedures 7

5 Dictionaries 10

1 Stars and Stripes

Let’s write a procedure, stars-n-stripes, which consumes two natural numbers, num-stars and num-stripes, and produces a set of strings represented as a list of strings, the elements of which constitute all possible permutations of strings of length num-stars + num-stripes consisting of num-stars "*"s and num-stripes ".-"s.

(stars-n-stripes 2 1)
=> (list "*-" "*-" "*-"")

(stars-n-stripes 2 2)
=> (list "*- -" "*- -" "*- -" "*- -" "*- -" "*- -" "*- -" "*- -"")

The order of the strings in the output list is unspecified. So on inputs 1 1, the output (list "-""-""") is as good as (list "-""-""").

As always, let’s follow the design recipe in writing this procedure.

;; Data Def: The inputs to the stars-n-stripes procedure are two natural
;; numbers, and the output is a list of strings. Here are some examples:
;; Examples of natural numbers:
;; 0, 1, 2, 3
;; Examples of (string list):
;; (list "*")
;; (list "*-" "*-")
;; stars-n-stripes : number * number → (string list)
;; Input: two natural numbers, num-stars and num-stripes
;; Output: a list of strings of all possible arrangements of num-stars "*"s
;; and num-stripes "-"s

(define (stars-n-stripes num-stars num-stripes)
  ...)

;; Test cases for stars-n-stripes
(check-expect (stars-n-stripes 0 0) empty)
(check-expect (stars-n-stripes 1 0) (list "*"))
(check-expect (stars-n-stripes 0 1) (list "-"))
(check-expect (stars-n-stripes 1 1) (list "*-", "-*"))
(check-expect (stars-n-stripes 2 3)
  (list "**--", "*---", "*--*", "-*--", "*-*-", "-*-*", "--**", "--*-", "---*", "---")
)

We know stars-n-stripes is a recursive procedure, partly because we’re in CS17 and everything we’ve written so far has been recursive, mostly because that’s what makes the most sense for this procedure.

What are our cases?

(define (stars-n-stripes num-stars num-stripes)
  (cond
    [(and (zero? num-stars) (zero? num-stripes)) (list "") ... ]
    [(and (zero? num-stars) (succ? num-stripes)) ... ]
    [(and (succ? num-stars) (zero? num-stripes)) ... ]
    [(and (succ? num-stars) (succ? num-stripes)) ... ]))

What’s our base case? Well we have two options: empty or (list ""), our specs say we produce a list of strings with strings of length num-stars + num-stripes so our base case should be (list "").

(During class, I deliberately wrote the base case wrong, and then went back to fix it, to emphasize the importance of doing these things right!)

What about our second and third cases? Well, if you have n num-stripes and zero num-stars, then we’re going to have a single string of n "-"s. The opposite is true if we have n num-stars and zero num-stripes. Great! We have our first three cases:

(define (stars-n-stripes num-stars num-stripes)
  (cond
    [(and (zero? num-stars) (zero? num-stripes)) (list "") ]
    [(and (zero? num-stars) (succ? num-stripes))
      (map (lambda (x) (string-append x "-"))
        (stars-n-stripes 0 (sub1 num-stripes)))]
    [(and (succ? num-stars) (zero? num-stripes))
      (map (lambda (x) (string-append x "*"))
        (stars-n-stripes (sub1 num-stars) 0))]
    [(and (succ? num-stars) (succ? num-stripes)) ... ]))
Okay, now let’s look at our final case. For simplicity, let’s label num-stars as \( n \) and num-stripes as \( k \).

We know we want to make all patterns of \( n \) stars and \( k \) stripes, and that somehow we’re going to build those up from simpler cases.

During last class, we drew several recursive diagrams and found that in the “ideation space” for each one, we got different ideas, none of which would work for the general case.

Let’s pause briefly to discuss a general approach that can be useful for these kinds of problems.

### 1.1 Combinatorial Problems: a generic approach for the simple ones

Problems that involve writing out all examples of some pattern, or counting how many examples of something there are, or similar activities, are called *combinatorial problems*. Example combinatorial problems:

- How many binary trees with exactly \( n \) nodes are there, as a function of \( n \)?
- Given an arithmetic expression like \( 3 + 4 \times 5/7 + 8 \), how many distinct ways are there to parenthesize it? (Here \((4)\) and 4 are not counted as distinct, for example.)
- How many ways of shuffling a deck of cards have all the aces together somewhere in the deck?

For simple combinatorial counting or enumeration problems, there’s a trick that often works: take the things you’re trying to count or enumerate and divide them into two groups that are *disjoint*, i.e., that share no elements. (You can also divide them into overlapping groups, but then you have to remove duplicates, and that makes both enumeration and counting more difficult, so we try to avoid it).

If counting the items in these two smaller groups is then an easier problem, you’re doing well. (Of course, if one group has no elements and the other is your whole original group, you’re probably not getting anywhere. So dividing the students in our class into “those shorter than 8 feet tall” and “those 8 feet tall or taller” wouldn’t be a very useful division.)

As an example: For the “counting aces” problem, you might break things into the cases where the top card is an ace and the case where it is not.

If your broken-up sets are actually similar to your original set in some way, you can often repeat the process. Let’s go back and see how this applies to stars-n-stripes.

### 1.2 Breaking up stars and stripes

There’s a really easy way to divide up stars-n-stripes patterns into two piles: consider all patterns that start with a star, and all that start with a stripe. These are pretty clearly disjoint!

If we’re looking at all 3-star, 2-stripe combinations, we get:

- ***--
- **--*
- ***--
But is counting each of these piles any easier than counting the whole thing? Well, strip off that initial star from every item in the first list, and you have

**--
--**
-**-
-*-
*--
...

...which is exactly all combinations of 2 stars and 2 stripes! That’s a smaller instance of the same problem!

In the same way, if we strip off the initial stripe from each item in the second group, we have all 3-star, 1-stripe combinations. This just begs for recursion!

Notice, though, that we’ll have to make two recursive calls, each of which will return a list of answers, and then we’ll have to slightly modify the items in each list - adding a stripe or star - and then append these lists. That means that the size of the answers we’re getting, as we rise up in the recursion, is rapidly getting bigger and bigger — very different from most of our recursive procedures, which have taken in a list of length \( n \) and produced a bool, or an int, or perhaps a length-\( n \) list. That, too, is typical of combinatorics: the size of the things you’re counting or enumerating tends to be pretty large!

It’s time to put this into code.

For the two complicated base cases, I’ve simplified the recursion because in each of those cases there’s nothing in one of the two sets in our decomposition.

```scheme
(define (stars-n-stripes num-stars num-stripes)
  (cond
    [(and (zero? num-stars) (zero? num-stripes)) (list "")]
    [(and (zero? num-stars) (succ? num-stripes))
      (map (lambda (x) (string-append "-" x))
          (stars-n-stripes 0 (sub1 num-stripes)))]
    [(and (succ? num-stars) (zero? num-stripes))
      (map (lambda (x) (string-append "*" x))
          (stars-n-stripes (sub1 num-stars) 0))]
    [(and (succ? num-stars) (succ? num-stripes))
      (append (map (lambda (x) (string-append "*" x))
                    (stars-n-stripes (sub1 num-stars) num-stripes))
              (map (lambda (x) (string-append "-" x))
                    (stars-n-stripes num-stars (sub1 num-stripes)))))])
)```
1.3 Analysis of Stars and Stripes

Let’s try to analyze this procedure. How long does it take to run? The recursive call recurs either `num-stars` or `num-stripes`, but not both. Neither the maximum nor one of the arguments will measure the size of the problem. We will have to go with the sum of arguments.

Let \( S(u) \) be the maximum number of operations in computing \( \text{stars-n-stripes num-stars num-stripes} \) for any pair of natural numbers \( n \), the number of stars, and \( k \), the the number of stripes, with \( u = n + k \).

We will get a recurrence like \( S(u) \leq A(v) + B + 2S(u - 1) \). \( A(v) \) is the cost of `append` and \( v \) is the size of the list returned by the recursive call. \( B \) is the constant work that we must do. And, since we make two recursive calls, we add \( 2S(u - 1) \).

2 Bignum

Spike presented in class a solution to bignum in Ocaml.

3 Procedure Definitions

In OCaml, there are two kinds of procedures: primitive, which are often written using infix notation, and user-defined, which are always written using prefix notation.

Note: A prefix procedure in OCaml can be applied to one argument only!

In particular, all user-defined procedures can be applied to one argument only. This may seem limiting, but it is not, because of currying, as you will learn shortly.

Procedure Definitions The shape of a (one-argument) procedure definition is:

\[
\text{let } \text{id} <\arg> = \text{body} \]

For example,

\[
\text{let add17 num = num + 17} \]

When applied to an argument (an `int`, specifically), this procedure evaluates as expected:

\[
\text{add17 1} \\
\Rightarrow 1 + 17 \\
\Rightarrow 18
\]

The analog of this code in Racket is:

\[
\text{(define (add17 num) (+ num 17)) (add17 1) => (+ 1 17) => 18}
\]
Anonymous Procedures  We all think that \texttt{lambda} is fun, don’t we? With that in mind, the shape of an anonymous procedure in OCaml is quite reasonable:

\begin{verbatim}
fun <arg> -> <body>
\end{verbatim}

In fact, \texttt{fun} is an abbreviation for \texttt{function}. But why bother typing \texttt{function} when it is so much fun to type \texttt{fun}.

Here is an example:

\begin{verbatim}
fun num -> num + 17
\end{verbatim}

Anonymous procedure application works just as it does in Racket (except that it is even more fun):

\begin{verbatim}
(fun num -> num + 17) 1
\end{verbatim}

\begin{verbatim}
=> 1 + 17
=> 18
\end{verbatim}

The analog of this procedure application in Racket is:

\begin{verbatim}
((\texttt{lambda} (num) (+ num 17)) 1) \Rightarrow (+ 1 17) \Rightarrow 18
\end{verbatim}

Syntactic Sugar  Now, can you guess what was really going on when we defined the \texttt{add17} procedure? If you guessed that the name \texttt{add17} was associated with the anonymous procedure \texttt{(fun num -> num + 17)}, then you guessed correctly.

In other words,

\begin{verbatim}
let add17 num = num + 17
\end{verbatim}

is actually syntactic sugar for

\begin{verbatim}
let add17 = fun num -> num + 17
\end{verbatim}

So, in the above evaluation of \texttt{add17 1}, we glossed over what was going on under the hood. Here is the whole story, with nothing swept under the rug:

\begin{verbatim}
add17 1
\end{verbatim}

\begin{verbatim}
=> (fun num -> num + 17) 1
=> 1 + 17
=> 18
\end{verbatim}

The analog of this derivation in Racket is:

\begin{verbatim}
(add17 1)
\end{verbatim}

\begin{verbatim}
=> ((\texttt{lambda} (num) (+ num 17)) 1)
=> (+ 1 17)
=> 18
\end{verbatim}

More generally, just like in Racket
let <id> <arg> = <body>

is syntactic sugar for

let <id> = fun <arg> -> <body>

**Type Annotations**  If a procedure takes an argument of type \(⟨\text{arg-type}⟩\) and returns a value of type \(⟨\text{val-type}⟩\), then its type signature is written as \(⟨\text{arg-type}⟩ - ⟨\text{val-type}⟩\). For example, both sample OCaml procedures above (add17 and the anonymous procedure) are of type \(\text{int} - \text{int}\).

When defining procedures, it is good practice to annotate the procedures’ inputs with their types. For example,

```ocaml
let add17 (num : int) = num + 17

fun (num : int) -> num + 17
```

You should also type annotate a procedure’s output, like this:

```ocaml
let add17 (num : int) : int = num + 17

(fun (num : int) -> num + 17 : int -> int)
```

**Question:** How do you type in the arrows?

**Answer:** You type a “minus” sign followed by a “greater than” sign.

That completes the story for one-argument procedures, and modulo syntax, things work as they did in Racket. When we come to two argument procedures, however, we will find that OCaml differs from Racket, because OCaml does not technically support multi-argument procedures.

### 4 Multi-parameter Procedures

We’ve written two procedures (one anonymous and one user-defined), each of which adds 17 to its parameter. Similarly, we could write a procedure (or two) that adds 18 to its parameter. Or smelling abstraction, we could write a more general procedure that consumes two parameters and produces their sum. Opting for anonymity, here’s how that would look:

```ocaml
fun x y -> x + y
```

Now for the tricky question. What is the type of this anonymous procedure? This question is tricky because this procedure appears to consume two parameters, but OCaml procedures can only consume one parameter!
**Question:** Is the type of this anonymous procedure `int * int -> int`?

**Answer:** I’m afraid not. That would be the type of a procedure that consumes a 2-element tuple as its sole input:

```
fun (x, y) -> x + y
```

What’s going on here is that

```
fun x y -> x + y
```

is actually syntactic sugar for

```
fun x -> (fun y -> x + y)
```

As you can see, we have one procedure that consumes an `int` and produces a procedure. The produced procedure in turn consumes an `int` and produces an `int`.

This means of defining multi-parameter procedures is called currying.\(^1\)

With it, we nest one-parameter procedures, one for each parameter we need. That way, all are bound within the body of the innermost procedure just as they would be bound in the body of a multi-parameter procedure in Racket.

The type of a curried procedure follows from its definition. For example, the type of our anonymous procedure is:

```
int -> (int -> int)
```

As the definition of `add` consists of two nested `fun` expressions, the type of `add` consists of two nested `->` type-expressions. This correspondence is the key to understanding the types of curried procedures, which is why it is important to remember the way multi-parameter `fun` expressions desugar.

Let’s see how function application works:

```
(((fun x -> (fun y -> (fun z -> x + y + z))) 1) 2) 3
=> ((fun y -> (fun z -> 1 + y + z)) 2) 3
=> (fun z -> 1 + 2 + z) 3
=> 1 + 2 + 3
=> 6
```

Here is our same adding procedure from before, applied to three arguments. As you can see, each application eliminates one of our nested procedures until we just have `1 + 2 + 3` left, which evaluates normally.

\(^1\)Currying is named for the logician Haskell Curry, who studied under David Hilbert. Curry is one of very few mathematicians to be remembered by naming something for both his first (the functional programming language, Haskell) and last name.
Writing all those parentheses seems a bit verbose, right? Even in Racket you don’t need to introduce a new pair of parentheses for each and every argument. Well fortunately OCaml associates some things so that we need far less parentheses in practice.

The -> is right associative: i.e., it binds most tightly to the right, so that

```
fun x -> fun y -> x + y
```

means

```
fun y -> (fun x -> x + y)
```

and

```
int -> int -> int
```

means

```
int -> (int -> int)
```

On the other hand, procedure application is left associative: i.e., it binds most tightly to the left, so that

```
add 17 1
```

means

```
((add 17) 1)
```

We can now rewrite our example from above as:

```
(fun x ->
  fun y ->
    fun z ->
      x + y + z)

1 2 3
```

```
=> (fun y ->
    fun z ->
      1 + y + z)

2 3
```

```
=> 1 + 2 + 3
```

```
=> 6
```

Much less cluttered!

It turns out that just as OCaml will curry our procedure expression for us, it also will curry our procedure definitions. How convenient! For example, `add` defined to be our procedure from before,

```
let add (x : int) (y : int) : int = x + y
```

is syntactic sugar for

```
let add : int -> int -> int =
  fun x ->
    fun y ->
      x + y
```

Here are two more examples, where we drop the parentheses for readability.

This definition:

```
let foo (x : int) (y : int) (z : int) : int = (x + y) * z
```

is syntactic sugar for this one:

```
let foo : int -> int -> int -> int =
  fun x ->
    fun y ->
      fun z ->
        (x + y) * z
```

And this definition:

```
let bar (a : float) (b : float) (c : float) (d : float) : float =
  (a -. (b /. c)) ** d
```

is syntactic sugar for this one:
let bar : float -> float -> float -> float -> float =
  fun a -> fun b -> fun c -> fun d -> (a -. (b /. c)) ** d

Since OCaml has no notion of a multi-parameter procedure, all multi-parameter procedures are
either curried, or (more rarely) written with a single-parameter procedure that consumes a tuple.
Currying gives OCaml a certain theoretical elegance. Whereas other languages have two different
type errors: applying the wrong number of arguments to a function, and applying arguments to
a non-function, Currying means the former becomes the latter. More generally, currying allows
OCaml to be a smaller and simpler language than it would be otherwise.

Infix Tricks For every arithmetic infix procedure application in OCaml, there is corresponding
prefix procedure application. In particular, for an infix procedure \( \langle \text{op} \rangle \), the corresponding prefix
procedure is spelled \( \langle \langle \text{op} \rangle \rangle \). For example, the following two OCaml expressions are equivalent:

\[
\begin{align*}
1 + 17 & \\
=> 18 & \\
( + ) 1 17 & \\
=> 18
\end{align*}
\]

If an infix operator consumes two arguments, one of type \( \langle \text{left-type} \rangle \) and the other of type \( \langle \text{right-type} \rangle \),
and returns a value of type \( \langle \text{val-type} \rangle \), then the type of that operator is not written as \( \langle \text{left-type} \rangle * \langle \text{right-type} \rangle -> \langle \text{val-type} \rangle \); rather, it is written in its curried form as \( \langle \text{left-type} \rangle -> \langle \text{right-type} \rangle -> \langle \text{val-type} \rangle \). For example, the type of \( ( + ) \) is \( \text{int} -> \text{int} -> \text{int} \).

Note: You must enter \( ( + ) \) if you want OCaml to return the type of +.
The infix form is shorter to type and more closely resembles standard mathematical notation, so it
is used more frequently. The prefix form is useful when you want to apply one procedure to another
(i.e., you want the latter to be an actual argument of the former)—for example, if you wanted to
fold addition across a list of integers you would apply the fold procedure to \( ( + ) \).

Question: Can you use this trick to make :: into a prefix operator?
Answer: No, I’m afraid not. But you can do this: let cons x y = x :: y.

5 Dictionaries

Dictionaries are a special type of data structure used in many different languages. You can think of
a dictionary as a function, call it \( f \). This function takes in a key and returns a value associated
that key. Thus, you can also think of dictionaries as a set of key-value pairs.

You’ve encountered things resembling dictionaries before! A phone book, for example, can be
thought of as a dictionary where names are the keys and phone numbers are the values associated
with a certain key. An English-French dictionary has English words as its keys, and French words
as the associated values. Even environments can be thought of as dictionaries from identifiers to
values bound to those identifiers.

There are some typical operations associated with dictionaries:
• Is this key in the dictionary
• What value is associated with this key
• Add a key-value pair
• Replace a key-value pair
• Delete a key-value pair

Let’s attempt to develop a lookup procedure that takes in a key and a dictionary, and returns the value associated with that key.

```ocaml
(* my_dict is an English to French dictionary *)
my_dict = [("I", "je"); ("you", "tu"); ("dog", "chien"); ...] ;;

(* Input: an English word to look up in a dictionary, key
 a dictionary from English to French words, dict *)
(* Output: the French words associated with key *)
let rec lookup (key : string) (dict : (string * string) list) : string =
  match dict
  | [] -> ???
  | (k, v) :: tl -> if (key == k) then v else (lookup key tl) ;;
```

But what happens if the key we are looking for is not in our dictionary? We’ll need a way to tell the user that our lookup procedure did not find a value in our dictionary associated with the key they entered. Can we do better than outputting an error message in this case?

**Options**  Options are a built-in type for handling success/failure cases!

```ocaml
(* The following defines a type 'a option *)
type 'a option = Some of 'a | None

The intended use of an option is to indicate that either what you were looking for was found, or not. If an answer was found, a procedure returning an option will return Some(x), where x is the answer you were looking for. If an answer was not found, the procedure will return None.

Having a procedure return an option instead of a value is helpful for dictionaries, where we aren’t sure if a lookup procedure will have a non-existent key as an input. We can use options to remedy this issue! Keep in mind, though, that if options are used, a new type signature will be required for most functions. Notice how the type signature changes for our `lookup` procedure.

```ocaml
...
let rec lookup (key : string) (dict : (string * string) list) :
  string option =
  ...
```
Writing `lookup` using options is left as a task in Lab 7.

Here is an example of a procedure that makes use of options. Note the use of parentheses to wrap the value associated with a `Some` option. This is a good habit to get into.

```ocaml
let rec sum_but_ones = function
| [] -> None
| l :: tl -> sum_but_ones tl
| hd :: tl -> (match sum_but_ones tl with
             | None -> Some (hd)
             | Some (s) -> Some (hd + s)) ;;
```

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)