(Provisional) Lecture 19: Combinatorics and more Reason
10:00 AM, Oct 18, 2019

Contents

1 Reason so far

Quick review of Reason,

- we have seen basic types: ints, floats, strings, bools, list(int), list('a), function types and more to come.
- "let" is used instead of (define...); each "let" opens a new environment.
- Arithmetic is "infix"; other functions are applied by writing f(x) or f(x, y)
- Type-ascription: (x:int) means "x is an expression whose value has type int"
- Lists: [], [1,2,3], [3, ... [4, 5]] "conses" 3 onto [4,5] to produce [3,4,5].
- Lists are monotype lists – all items must be the same time.
- List types are written "list(int)" rather than "int list".
- Tuples: (true, "my string") has type "(bool, string)" (which in Racket we would have written "bool * string")

2 Combinatorial Problems

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:

- They can be used to count or enumerate things. Like find all triples of ints between 0 and 10 whose sum is no more than 7.
- All increasing numbers between 0 and 20.
- The number of ways to partition a regular n-gon into triangles.

Today we are going to focus on one in specific: stars-n-stripes That finds all strings containing only "*" and ".", and which contain exactly n stars and n stripes.
3 Stars and Stripes

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

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

The order of the strings in the output list is unspecified. So on inputs 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 is a 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 : numb -> (string list)
;; Input: an integer, n
;; Output: a list of strings made up from exactly n asterisks and n dashes
(define (ss n)
  ...)

;; Test cases for stars-n-stripes
(check-expect (stars-n-stripes 0) empty)
(check-expect (stars-n-stripes 1) (list "*" "-"))
(check-expect (stars-n-stripes 2) (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. So, let’s write a recursive diagram for an input of 2:

– OI: 2
  * RI: 1
    * RO: * - *
  - : OO: **** *- *- *- *- *- *- *- *- *- *- *- *- *- *- *- *- *-
The idea for adding the star in is that you put a star in every possible ‘slot’: before, between, and after the stripes. However, this idea doesn’t work for adding in the second stripe: you’ll get duplicates.

One approach to this problem is to go ahead and produce duplicates, but filter them out later. However, this approach is incredibly slow and inefficient. Our recursive result doesn’t seem very helpful. When this happens a good strategy is to make the problem harder, so the recursive results can be more useful. Our more general problem is: Given \(n\), \(k\), create all strings containing \(n\) “*”s and \(k\) “-”s in any order.

- \((\text{sns} \ 2 \ 0) = \text{\{list "**"\}}\)
- \((\text{sns} \ 2 \ 1) = \text{\{list ".**" ".*." ".**."\}}\)

We can write recursive diagrams to this new problem.

\[\begin{align*}
\text{OI:} & \ 1 \ 2 \\
\text{RI:} & \ 0 \ 2 \\
\text{RO:} & \ - \ - \\
\text{isea:} & \text{ stick a star in every possible slot}
\end{align*}\]

\[\begin{align*}
\text{OO:} & \text{ ** -* -**}
\end{align*}\]

Another Diagram:

\[\begin{align*}
\text{OI:} & \ 0 \ 2 \\
\text{RI:} & \ 0 \ 1 \\
\text{RO:} & \ - \\
\text{isea:} & \text{ you can’t stick a stripe in every slot - duplicates!}
\end{align*}\]

\[\begin{align*}
\text{OO:} & \text{ --}
\end{align*}\]

Filtering out duplicates is very slow, let’s try something else...

### 3.1 Combinatorics Approach

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.)

If your broken-up sets are actually similar to your original set in some way, you can often repeat the process. Splitting things into 2 smaller problems is an application of divide and conquer in computer science. Let’s go back and see how this applies to \text{stars-n-stripes}.

### 3.2 Breaking up Stars and Stripes

There’s a really easy way to divide up \text{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 2-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 1 star 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 2-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!

Note: we can write this code in any language, This isn’t a racket problem. Can you write the base cases for this recursion?

It’s time to put this into code.

### 3.3 Writing Stars and Stripes

First off, let’s consider our cases:

```scheme
(define (stars-n-stripes num-stars num-stripes)
  (cond
    [(and (zero? num-stars) (zero? num-stripes)) ... ]
    [(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 ")")`. 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:
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.

And there we have \texttt{stars-n-stripes}.

## 4 Interacting with Reason

A REPL, or Read-Evaluate-Print-Loop, is a way to run code in a terminal. We are going to go back to reason and interact with it in an on-line REPL, Sketch.sh. There is a new syntax, \texttt{switch(x)} It is used for pattern matching.

If \texttt{x} is 1 then give 4, if \texttt{x} is 2 give 0, if it is anything else give -1.

You can use pattern matching with lists.
As a general rule, we suggest ending every line with ; EXCEPT for the following:
* multi-line lets, like from the example above.
* multi-line function definitions
  - For example:

```reason
let f (x) = switch (x) {
  | [] => ...
  | [hd, ... tl] => ...
};
```

* let ... in expressions

Now we are going to write the length procedure:

```reason
let rec len = lst =>
  switch(lst){
    | [] => 0
    | [hd, ... tl] => 1 + len(tl)
};
```

You have to write let rec to tell Reason the procedure is recursive. There is another way to write it because as you see lst is declared as an argument and then we switch on it and don’t use it again. The creators of Reason accounted for this. So we can write the length procedure as:

```reason
let rec len = fun
  | [] => 0
  | [hd, ... tl] => 1 + len(tl);
```

5 Summary

Ideas
* Recursive diagrams are wonderful, and we hope you use them a lot. (I know I do! - Your friendly neighborhood TA)

Skills
* A good approach for combinatorial or enumeration problems is to take the things you’re trying to count and divide them into two disjoint groups. Check out the stars and stripes function to see this in action!
* We can now write Reason code in a REPL
* We can now pattern match!
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/csci0170/feedback