Lab: Abstraction with Records
CSC/MAT 208: Discrete structures
Department of Computer Science
Department of Mathematics and Statistics
Grinnell College

Pairs

Here are the programmer pairs for today’s lab. Drivers are on the left.

- David Chang and Madeleine Goldman
- Thu Nguyen and Adam Wesely
- Linda Oyolu and Anita DeWitt
- Jong Hoon Bae and Dennis Chan
- Corey Simmonds and Jacob Ekstrand
- Julia Fay and Adam Hudson
- Danielle Williams and Matt Jasaitis
- Amanda Hinchman-Dominguez and Devin Dooley
- Bazil Mupisiri and Tyler Williams
- Ella Nicolson and Elizabeth Zak
- Kathryn Yetter and Kevin Connors
- Samee Zahid and Lindsey Byrne
- Muhammad Hamza and Marcel Champagne

Michael Owusu will replace anyone who is absent.

Nimbers

There is a two-player children’s game called nim, played with an arbitrarily large number of identical pieces such as checkers or marbles. The pieces are grouped into three or more piles, and then the players take turns, each one selecting a pile and removing as many pieces as she wants from that pile. The player who takes the last piece wins.

One reason that nim remains a children’s game is that the complete strategy of the game is known and too many adults can either figure it out or learn it from someone else. It takes a pretty bright child to get the hang of it, though.
The idea behind the strategy is to think of the number of pieces in each pile as a sum of powers of two. If, at the beginning of a turn, every power of two in every pile can be matched up with the same power of two in another pile, then it’s a losing position for whichever player has to go next.

Since that player has to take some number of counters from a single pile, he will always have to leave some unmatched powers of two in the position that results from his play, no matter how many or how few counters he takes. This “unbalances” the resulting position. The other player always has at least one play that restores the balance by adjusting some other pile so as to provide a match for every power of two again.

For instance, if there are four piles, and they have 3, 7, 5, and 12 pieces in them, the position is unbalanced:

\[
\begin{align*}
3 &= 1 + 2 \\
7 &= 1 + 2 + 4 \\
5 &= 1 + 4 \\
12 &= 4 + 8
\end{align*}
\]

The 1s in the 3 and 7 piles match, but that leaves the 1 in the 5 pile unmatched. The 2s in the 3 and 7 piles match. The 4s in the 7 and 5 piles match, but that leaves the 4 in the 12 pile unmatched. And the 8 in the 12 pile is also unmatched.

To restore balance, we have to remove at least five pieces from the 12 pile so that fewer than eight will be left, in order to get rid of the unmatched 8 in the current position. Then we have to remove at least four more, to eliminate the unmatched 4. The 2s already match, so we don’t want to create an unmatched 2, which means that we have to remove two more. But we need to leave the last piece in the pile behind to create a match for the unmatched 1 in the 5 pile.

So the only winning move is to take eleven pieces from the 12 pile, leaving just one:

\[
\begin{align*}
3 &= 1 + 2 \\
7 &= 1 + 2 + 4 \\
5 &= 1 + 4 \\
1 &= 1
\end{align*}
\]
Now all the powers of two can be paired off and the position is balanced. Any play that the other player makes will unbalance it again, and the first player will just rebalance in the same way.

**Exercise 0:** Play out the rest of the game just shown, to demonstrate that the player who just took eleven pieces can always get the last piece by rebalancing the position at each play.

**Exercise 1:** In the position with piles of 3, 7, 5, and 12, suppose that the player on move blunders by taking five pieces from the 7 pile. Show that this leaves an unbalanced position and demonstrate that the second player can now win by confronting the blunderer with a balanced position.

In figuring out the correct move in a complicated nim position, it is useful to be able to carry out an arithmetic operation called “nim-addition” on the sizes of the piles. The “nim-sum” of two pile sizes is the sum of the unmatched powers of two in those piles. So, for example, let’s compute the nim-sum of 3 and 6:

\[
\begin{align*}
3 & = 1 + 2 \\
6 & = 2 + 4 
\end{align*}
\]

The 2s match, but the 1 and 4 don’t, so the nim-sum is 1 + 4, which is 5.

**Exercise 2:** Compute the nim-sum of 9 and 14.

Nim-addition is commutative and associative, so that it makes sense to speak of the nim-sum of all of the sizes of piles in a position — just add each pile size into a running total using nim-addition.

By definition, then, a position is balanced if, and only if, the nim-sum of the sizes of the piles is 0. Note that the nim-sum of any number and itself is 0. That means that, in nim-addition, every number is its own “negative” (additive inverse), and “nim-subtraction” is exactly the same operation as nim-addition.

The sizes of piles in these computations are of course always natural numbers, but in the context of analyzing nim positions it is conventional to refer to them as “nimbers” and to speak of “taking the nim-sum of two (or more) “nimbers.”

It’s not difficult to compute the nim-sum of two nimbers in Scheme using recursion over the bits in their binary representations, from least significant to most significant. Here’s how I do it:
(define nim-sum
  (lambda (left right)
    (cond ((zero? left) right)
          ((zero? right) left)
          (else (let ((smaller-sum (nim-sum (div left 2) (div right 2)))
                     (last-bit (if (eq? (even? left) (even? right)) 0 1)))
                     (+ (* smaller-sum 2) last-bit)))))

**Exercise 3:** Walk through this definition, explaining how it works and why it gives the correct answer.

**Implementing nimbers in Scheme**

Let’s use Scheme’s facility for defining new types to create a `nimber` data type and define a couple of useful procedures for them, collecting our definitions in a new library called *(discrete nimbers)*.

**Exercise 4:** In what directory will you have to create the `nimbers.sls` file in order to enable Racket to find the new library?

The standard library that supports record type definitions is called *(rnrs records syntactic)*, so we’ll need to import that library along with *(rnrs base)* when defining our library.

The only field that we’ll need in our `nimber` records is the field that contains the natural number indicating the size of a pile.

**Exercise 5:** Write the `define-record-type` form that declares a new record type called `nimber` with one field called `size`. Should we make this field mutable? What are the pros and cons of that choice?

**Exercise 6:** What are the names of the constructor, type predicate, and accessor(s) that are implicitly defined by the record type definition? Start an export list for the *(discrete nimbers)* library and add those names to it.

We’ll define and export two more operations in the *(discrete nimbers)* library: `nimber+` and `mex`.

4
• The `nimber+` procedure will take two arguments, both of which must be nimbers, and returns their nim-sum, also as a nimber. (This can be a “husk” procedure that simply extracts the pile sizes from the given nimbers, invokes the “kernel” procedure `nim-sum` to calculate the numerical result, and “packages” that number as a nimber before returning it.)

• The `mex` procedure takes one argument, a list of nimbers, and returns the least nimber that is not an element of the given list. (The name “mex” is short for “minimum excluded.”) For example, if the elements of the list are the nimbers with sizes 4, 7, 6, 0, 9, and 1, the result of applying `mex` to the list will be the nimber with size 2. This procedure isn’t used in the strategy for playing nim, but it’s handy in the analysis of some other turn-taking games that use nimbers.

Exercise 7: Complete the `(discrete nimbers)` library by writing the definitions of `nimber+` and `mex` and adding those names to the export list. Don’t export `nim-sum` — from the point of view of the `(discrete nimbers)` library, that’s a mere helper procedure, only for local use.

Exercise 8: Use your `(discrete nimbers)` library to write a top-level program that (a) defines a procedure `nimber-list-sum` that takes a list of nimbers and uses `nimber+` repeatedly get a single total, expressed as a nimber; (b) defines a predicate `balanced?` that takes a list of nimbers and determines whether it represents a balanced position in nim; (c) uses that predicate to figure out whether a nim position with five piles of sizes 31, 41, 59, 26, and 35 is balanced, displaying the string "first player wins" if it is balanced and the string "second player wins" otherwise.