PROGRAMMING IN HASKELL

Chapter 15 – Lazy Evaluation

Introduction

Expressions in Haskell are evaluated using a simple technique called lazy evaluation, which:

- Avoids doing unnecessary evaluation;
- Ensures termination whenever possible;
- Supports programming with infinite lists;
- Allows programs to be more modular.

Evaluating Expressions

```
square n = n * n
```

Example:

```
square (1+2)
```

=

```
square 3
```

=

```
3 * 3
```

=

```
9
```

Apply + first.

Another evaluation order is also possible:

```
  square (1+2)
=
  (1+2) * (1+2)
=
  3 * (1+2)
=
  3 * 3
=
  9
```

Apply square first.

Any way of evaluating the same expression will give the same result, provided it terminates.

Evaluation Strategies

There are two main strategies for deciding which reducible expression (redex) to consider next:

- Choose a redex that is innermost, in the sense that does not contain another redex;

- Choose a redex that is outermost, in the sense that is not contained in another redex.

Termination

```
infinity = 1 + infinity
```

Example:

```
  fst (0, infinity)
=
  fst (0, 1 + infinity)
=
  fst (0, 1 + (1 + infinity))
=
  ⋮
```

Innermost evaluation.

```
    fst (0, infinity)
=
    0
```

Outermost evaluation.

Note:

- Outermost evaluation may give a result when innermost evaluation <u>fails to terminate</u>;

- If <u>any</u> evaluation sequence terminates, then so does outermost, with the same result.

Number of Reductions

Innermost:

```
  square (1+2)
=
  square 3
=
  3 * 3
=
  9
```

3 steps.

Outermost:

```
  square (1+2)
=
  (1+2) * (1+2)
=
  3 * (1+2)
=
  3 * 3
=
  9
```

4 steps.

Note:

- The outmost version is inefficient, because the argument 1+2 is duplicated when square is applied and is hence evaluated twice.

- Due to such duplication, outermost evaluation may require more steps than innermost.

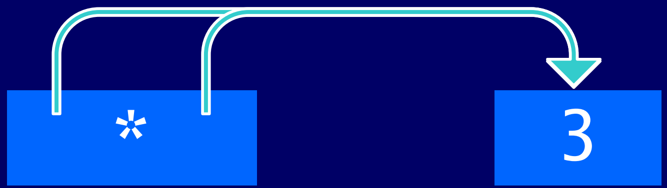
- This problem can easily be avoided by using pointers to indicate sharing of arguments.

Example:

`square (1+2)`

=

=



=

9

Shared argument evaluated once.

This gives a new evaluation strategy:

lazy evaluation	=	outermost evaluation + sharing of arguments

Note:

- Lazy evaluation ensures <u>termination</u> whenever possible, but <u>never</u> requires more steps than innermost evaluation and sometimes fewer.

Infinite Lists

```
ones = 1 : ones
```

Example:

```
   ones
= 1 : ones
= 1 : (1 : ones)
= 1 : (1 : (1 : ones))
= ⋮
```

An infinite list of ones.

What happens if we select the first element?

Innermost:

```
  head ones
=
  head (1:ones)
=
  head (1:(1:ones))
=
  ⋮
```

Does not terminate.

Lazy:

```
  head ones
=
  head (1:ones)
=
  1
```

Terminates in 2 steps!

Note:

- In the lazy case, only the <u>first</u> element of ones is produced, as the rest are not required.

- In general, with <u>lazy</u> evaluation expressions are only evaluated as <u>much as required</u> by the context in which they are used.

- Hence, ones is really a <u>potentially</u> infinite list.

Modular Programming

Lazy evaluation allows us to make programs more <u>modular</u> by separating control from data.

```
> take 5 ones
[1,1,1,1,1]
```

The data part ones is only evaluated as much as required by the control part take 5.

Without using lazy evaluation the control and data parts would need to be combined into one:

```
replicate :: Int → a → [a]
replicate 0 _ = []
replicate n x = x : replicate (n-1) x
```

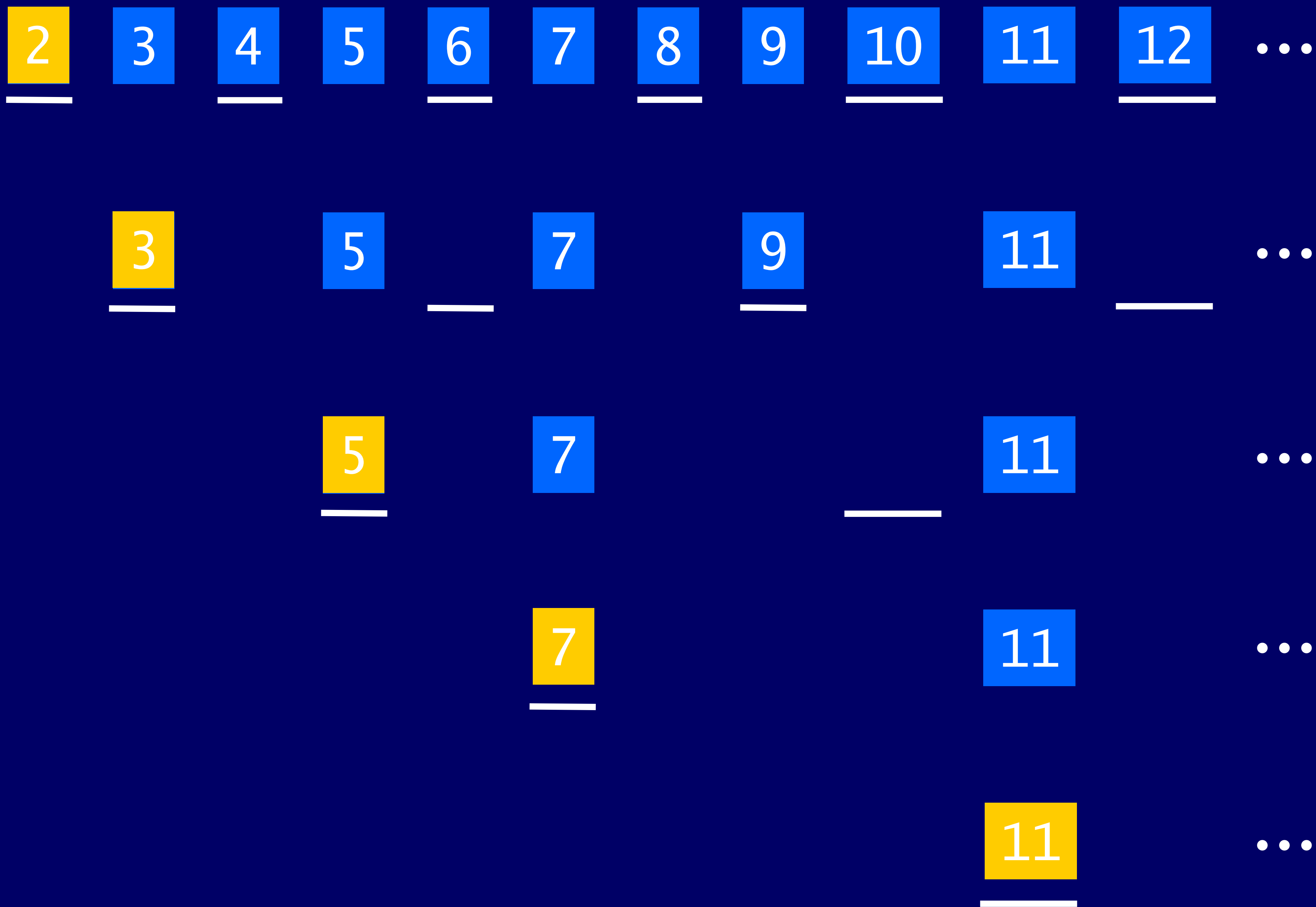
Example:

```
> replicate 5 1
[1,1,1,1,1]
```

Generating Primes

To generate the infinite sequence of primes:

1. Write down the infinite sequence 2, 3, 4, ...;
2. Mark the first number p as being prime;
3. Delete all multiples of p from the sequence;
4. Return to the second step.


2 3 4 5 6 7 8 9 10 11 12 …
3 5 7 9 11 …
5 7 11 …
7 11 …
11 …

This idea can be <u>directly</u> translated into a program that generates the infinite list of primes!

```
primes :: [Int]
primes = sieve [2..]
```

```
sieve :: [Int] → [Int]
sieve (p:xs) =
    p : sieve [x | x ← xs, mod x p /= 0]
```

Examples:

```
> primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,…
```

```
> take 10 primes
[2,3,5,7,11,13,17,19,23,29]
```

```
> takeWhile (< 10) primes
[2,3,5,7]
```

We can also use primes to generate an (infinite?) list of twin primes that differ by precisely two.

```
twin :: (Int,Int) → Bool
twin (x,y) = y == x+2
```

```
twins :: [(Int,Int)]
twins = filter twin (zip primes (tail primes))
```

```
> twins
[(3,5),(5,7),(11,13),(17,19),(29,31),…
```

Exercise

(1) The Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

starts with 0 and 1, with each further number being the sum of the previous two. Using a list comprehension, define an expression

```
fibs :: [Integer]
```

that generates this infinite sequence.