Programming Assignment #2 CS671

1st submission due 6/21/2021
2nd submission due 6/28/2021

This assignment focuses on functional programming using lists (and list-like structures). The first part implements recursive functions on a list-like representation of polynomials. The second part searches the problem space of instances of the finding-change problem using lists.

Representing polynomials

The first part of the assignment explores a representation of single-variable polynomials with integer coefficients. It illustrates simple recursive programming: The representation is linear (list-like) and most operations on it (e.g., multiplication or exponentiation) can be defined recursively.

Classes `S`, `X` and `C`

The chosen way to represent polynomials is inspired by Horner's algorithm. It may seem counterintuitive at first, but it results in simpler and faster code than using lists of coefficients.

Polynomial $P_0 = 5x^2 - 3x + 7$ can be written as: $7 + x(-3 + x(5))$. Notice how the polynomial is written as a number plus x times another polynomial, and so on recursively. This recursive structure—a sort of "*comb*", or linear tree—can be implemented in terms of three classes:

- Class `S` represents a sum of a number and x times a polynomial:
  Equation (1): $\texttt{S}(n, p) = n + x \cdot p$
- Class `X` is used to handle the special case where $n = 0$ and represents the multiplication of x by a polynomial:
  Equation (2): $\texttt{X}(p) = x \cdot p$
  Accordingly, class `S` is never used when $n = 0$ (class X is used instead).
- Finally, class `C` is used to represent a constant. It corresponds to the case where $p = 0$ in `S` (and, accordingly, class `S` is never used when $p = 0$):
  Equation (3): $\texttt{C}(n) = n$

Together, these three classes can be used to build a unique representation of any polynomial:

$$P_0 = 5x^2 - 3x + 7 = \texttt{S}(7, \texttt{S}(-3, \texttt{C}(5)))$$
$$P_1 = 2x^5 + 3x^2 - x + 5 = \texttt{S}(5, \texttt{S}(-1, \texttt{S}(3, \texttt{X}(\texttt{X}(\texttt{C}(2))))))$$
$$P_2 = x^3 - 4x^2 + x = \texttt{X}(\texttt{S}(1, \texttt{S}(-4, \texttt{C}(1))))$$

Note how the representation of P_1 uses successive `X` to account for the absence of x^3 and x^4 terms in the polynomial, and the representation of P_2 uses `X` as its root, not `S`, because the polynomial has no constant term. It is important to keep in mind that in $\texttt{S}(n, p)$, n and p cannot be zero:

- $\texttt{S}(n, \texttt{C}(0))$ is modeled as $\texttt{C}(n)$ and represents constant n;
- if p is not $\texttt{C}(0)$, $\texttt{S}(0, p)$ is modeled as $\texttt{X}(p)$ and represents the polynomial $x \cdot p$.

The Scala implementation of the three classes `S`, `X` and `C` is sketched in Listing 1 (next page). Together, they define a recursive data structure (classes `S` and `X` take polynomials as arguments in their constructors). Note how class `S` is similar to the class :: that defines lists in Scala (think of n as the head and p as the tail). Case classes are used to enable pattern-matching.

```
package cs671.polynomials

sealed abstract class Polynomial {
  ...
  def degree: Int
  def eval(x: Double): Double
  def +(that: Polynomial): Polynomial
  def +(n: Int): Polynomial
  def -(that: Polynomial): Polynomial
  def -(n: Int): Polynomial
  def *(that: Polynomial): Polynomial
  def *(n: Int): Polynomial
  def unary_- : Polynomial
  def @@(n: Int): Polynomial
  def derivative: Polynomial
}

case class C(c: Int) extends Polynomial {...}

case class X(p: Polynomial) extends Polynomial {...}

case class S(n: Int, p: Polynomial) extends Polynomial {...}
```

Listing 1: `Polynomial` class

Polynomial algorithms

Given this representation, most polynomial algorithms rely on decomposition, according to equations (1), (2) and (3), and are naturally recursive.

- `degree`: the degree of the polynomial, i.e., the highest degree of its terms: the degree of P_1 is 5. The degree of instances of `C` is zero; the degree of instances of S and X is one more than the degree of the polynomial they contain.
- `eval`: polynomial evaluation: $P_1(3) = 515$.
  The value of a constant is itself; the value of `S`(n, p) at x is n plus x times the value of p; the value of `X`(p) is x times the value of p.
- `+`: polynomial addition: $P_1 + P_2$ is $2x^5 + x^3 - x^2 + 5$. The name + is overloaded to allow a number to be added to a polynomial. For instance, $P_1 + 1 = 2x^5 + 3x^2 - x + 6$.
  These methods can be implemented recursively using equations (1), (2) and (3). However, one needs to watch out for malformed expressions. For instance, `S`(n, p) + `X`(q) is $n + x \cdot p + x \cdot q$, which is equal to $n + x \cdot (p + q)$. If $p + q \neq 0$, this is the term `S`($n, p + q$). However, if $p + q = 0$, it is the term `C`(n) instead, since the arguments of `S` should never be zero.
- `unary_-`: this method defines “unary minus”. $-P_2 = -x^3 + 4x^2 - x$.
  The negation of `C`(n), `X`(p) and `S`(n, p) is `C`($-n$), `X`($-p$) and `S`($-n,-p$), respectively.
- $-$: polynomial subtraction: $P_1 - P_2 = 2x^5 - x^3 + 7x^2 - 2x + 5$. The $-$ name is overloaded to allow $P_1 - 5 = 2x5 + 3x2 - x$. Note that $P_1 - P_2 = P_1 + (-P_2)$, making it possible to implement $-$ using `+` and `unary_-`.
- `*`: polynomial multiplication: $P_1 * P_2 = 2x^8 -8x^7 +2x^6 +3x^5 -13x^4 +12x^3 -21x^2 +5x$. The name `*` is overloaded to allow a polynomial to be multiplied by a number (scalar multiplication). For instance, $P_2 * 10 = 10x^3 - 40x^2 + 10x$.
  These methods can be implemented recursively, following the same strategy as in methods `+` and facing the same difficulties (malformed terms). For instance, `S`(n, p) * k (where k is an

integer) is S($n \cdot k$, $p * k$), unless $k = 0$. Similarly, S(n, p) * q (where q is a polynomial) can be calculated as $q * n$ + X($p * q$), unless $p * q$ is zero.

- `@@`: polynomial exponentiation: $P_2^3 = x^9 - 12x^8 + 51x^7 - 88x^6 + 51x^5 - 12x^4 + x^3$.
  It is defined recursively as:
  1. P `@@` $n = P$ * (P `@@` ($n - 1$)) for $n > 0$, and
  2. P `@@` 0 = 1, for any P (including C(0)).
- `derivative`: polynomial derivative. The derivative of P is sometimes written as P':
  $P_1' = 10x^4+6x-1$.
  The derivative of constants is zero; the derivative of S(n, p) and X(p) is the same: $p + x \cdot p'$, which is p + X(p.`derivative`), unless p.`derivative` is zero.

Implicit conversions and building syntax

The Polynomial companion object defines a method apply to create constant polynomials. It also defines a value `x` that represents the polynomial x. Importing `Polynomial.x` in programs enables the use of expressions of the form x `@@` 2 for x^2 or p1 `*` x for $P_1 \cdot x$.

Since `+`, `−` and `*` are overloaded, one can already write `x - 1` or `p1 * 10`. An implicit conversion is added to the `polynomials` package to enable expressions of the form `1 - x`, `2 + p1` and `3 * p2`. With these mechanisms in place, polynomials can be written in a more natural form:

```
val p1 = 2 * x @@ 5 + 3 * x @@ 2 - x + 5
val p2 = x @@ 3 - 4 * x @@ 2 + x
```

Implicit conversions are an advanced feature of Scala that will be discussed later. The implicit conversion code for this assignment is given and requires no modification.

Changer

Problem Definition

In this part of the assignment, we consider the problem of giving change using coins and bills from a cash register. Coins and bills are modeled as positive integers. The cash register is modeled as a multiset, represented as a list, possibly with duplicates; the order of elements in the list is unimportant. The problem becomes one of finding a sublist of integers that add up to a given target.[1] For example, there are two different ways to make 221 from [50 10 1 69 51 9 73 89 30]:

$$9 + 50 + 73 + 89 \qquad \text{or} \qquad 1 + 30 + 50 + 51 + 89$$

But to make 225 with the same set of numbers is impossible.

Algorithm

The simplest algorithm to solve this problem works recursively. Given a list of integers L and a target t:

i. If the target t is zero, the problem is trivially solved with an empty list.
ii. Otherwise, if the list L is empty, the problem has no solution and the algorithm fails.
iii. Otherwise, pick a number x from L and let L' be the list of remaining numbers from L after x is removed. If x is used to build the sum, this will leave a target of $t - x$ to reach with the remaining numbers. Therefore, try to solve the smaller problem of changing $t - x$ with L' using a recursive call.

[1] This problem is a classic of theoretical computer science, known as *subset-sum*.

- If this succeeds, we have a list of integers from *L′* that add up to *t* – *x*. Add *x* to the list to get a list of integers from *L* that add up to *x*, which is a solution to the original problem.
- Otherwise, if the algorithm fails to make *t* – *x* with *L′*, it means that *x* shouldn't be used. Discard it and try to make *t* with *L′* using a recursive call.

In this algorithm, the recursion has to choose between two branches: to use *x* or not. Which branch leads to a solution is unknown a priori. This algorithm first tries with *x* and, if this fails, tries without. Figure 1 shows how the algorithm is applied to the problem of reaching 5 with the list [1 2 3 5] (black arrows are recursive calls, blue arrows represent successful solutions and red arrows represent failures).

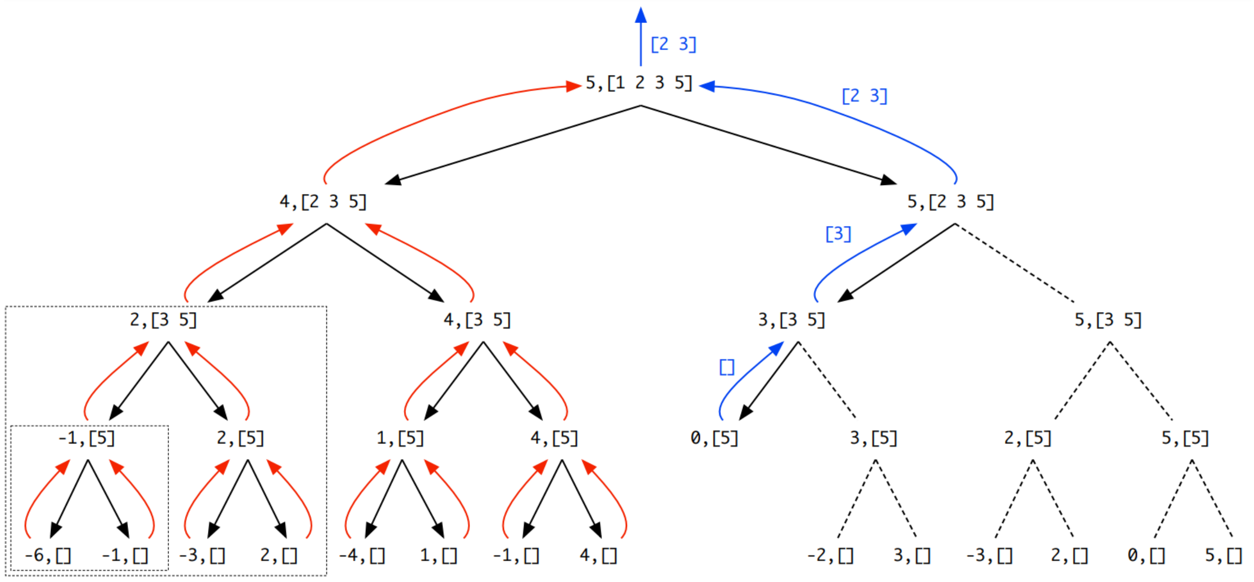

Figure 1: Applying the “change” algorithm.

Two minor optimizations can be added to the algorithm (it is not required to do so for the assignment):

- If *x* > *t*, the new target *t* – *x* would be negative and certainly cannot be reached using positive numbers. So the “use x” branch can be skipped in this case.
- Furthermore, if the list of numbers is sorted in non-decreasing order, all the remaining values in the list are at least as large as *x* and hence larger than *t*. The problem has therefore no solution and the second branch can be skipped as well.

In Figure 1, the tree `-1, [5]` is not explored under the first optimization (negative targets); if the second optimization is used, the tree `2, [3 5]` is skipped altogether (target smaller than all numbers). These optimizations do not fundamentally change the algorithm, which remains exponential in complexity.

Implementing backtracking

If a branch fails, the program needs to backtrack and try the other branch. Two features of Scala come in handy to implement this backtracking:

- *Persistent (immutable) data structures*. Because lists are immutable, the computation that takes place in a branch cannot modify the list of numbers, which is ready when trying the other branch if needed.

- *Options*. Options can be used to represent success (`Some(list)`) or failure (`None`).[2]

Scala Implementation

Object Changer implements three variants of the algorithm:

- `change` implements the algorithm as described above. It is guaranteed to find a solution if there is one and returns `None` otherwise.
- `changeBest` is a variation that returns a solution of minimal length, if there is one. The algorithm is modified to explore the branch “don't use *x*” even if the branch “use *x*” produces a solution, since there is no way a priori to know if the shortest solution uses *x* or not. After it has explored both branches, the algorithm returns the solution of smallest length, or None if both branches fail. In the scenario of Figure 1, this algorithm would continue after the tree `3,[3 5]` produces a solution and would explore the tree `5,[3 5]`, which would lead to a shorted solution (up from the leaf `0,[]`). Applied to the problem of reaching 221 with [50 10 1 69 51 9 73 89 30], this method produces the solution 9 + 50 + 73 + 89.
- `changeApprox` finds the closest under-approximation to a solution if the problem cannot be solved. It does this by exploring both branches and by returning the solution that is closest to (but below) the target. Note that, as an optimization, there is no need to explore the second branch if the first branch produces an exact solution. Applied to the problem of reaching 225 with [50 10 1 69 51 9 73 89 30], this method produces a list that adds up to 224, with a difference of 1.
  It is required that this method never produces a list that adds up to more than the target. For instance, given the problem of reaching 12 with [3 5 5 8 10]—which has no solution—the method produces [3 8] with a difference of 1, *not* [3 10] with a difference of –1.

Notes

- There is no requirement to rely on *tail recursion* anywhere in the polynomials implementation.
- Like Scala lists, polynomials are immutable. Operations do not modify polynomials and always return a new polynomial instead.
- `Polynomial` objects must always be in proper (canonical) form: no zero values in `S` or `X`. In the reference implementation, constructors do not reject illegal values so methods can create temporary malformed terms, e.g., `X(p + q)` even is *p*+*q* is zero. These objects are then checked and, when needed, replaced by well-formed objects before being returned to the user.
- `Polynomial` code can be placed in the class `Polynomial` itself (and use pattern-matching on `S`, `X` and `C`), or it can be specialized in classes `S`, `X` and `C`. For instance, a method `isZero` could be implemented entirely in class Polynomial (no mention of the method in classes `S`, `X` or `C`):

```
sealed abstract class Polynomial {
  def isZero: Boolean = this match {
    case C(0) => true
    case _ => false
  }
  ...
}
```

  Alternatively, the method could be left abstract in `Polynomial` and implemented in classes `S`, `X` and `C`:

[2] Another common approach is to use exceptions, but options are more convenient here.

```
sealed abstract class Polynomial {
  def isZero: Boolean // abstract
  ...
}
case class C(c: Int) extends Polynomial {
  def isZero = c == 0
  ...
}
case class X(p: Polynomial) extends Polynomial {
  def isZero = false
  ...
}
case class S(c: Int, p: Polynomial) extends Polynomial {
  def isZero = false
  ...
}
```

A third approach could use a default implementation in class `Polynomial`, overridden in class `C` only:

```
sealed abstract class Polynomial {
  def isZero: Boolean = false // default
  ...
}
case class C(c: Int) extends Polynomial {
  override def isZero = c == 0
  ...
}
```

Which approach is best depends on the method to implement. The starting code is given with all methods abstract in class `Polynomial`, but this can be changed. The API documentation generated by Scaladoc reflects the choices made in the reference implementation. You can make a different choice as to which methods are abstract and which are concrete (in other words, ignore the fact that the API documentation is showing some methods as being abstract/concrete).

- If additional methods are added to classes `S`, `X` and `C`, they should not be public. They can be `private` or `private[polynomials]`.
- The algorithms used in `Changer` are inherently inefficient (they take time exponential with the length of the list). Although some techniques can be used to improve them (e.g., memorization or dynamic programming), there is no simple "trick" that would allow to dispense exploring both branches in the worst case. Do not try to use "simpler" algorithms (e.g., sort the list from high to low and explore only one branch in changeBest). They won't work.
- The `Changer` object contains a simple command-line application, already implemented, that applies the three change strategies to the same problem:

```
> runMain change.Changer 50 10 1 69 51 9 73 89 30  221
change: Some(List(1, 30, 50, 51, 89)) (0.2 millis)
changeBest: Some(List(9, 50, 73, 89)) (0.4 millis)
changeApprox: (0,List(1, 30, 50, 51, 89)) (0.2 millis)

> runMain change.Changer 50 10 1 69 51 9 73 89 30  225
change: None (0.4 millis)
changeBest: None (0.6 millis)
changeApprox: (1,List(1, 30, 51, 69, 73)) (0.5 millis)
```