Programmeringssprog: Assignment 3

Rasmus Dalsgaard (201605295)

April 2017

Introduction

In this assignment, we're tasked to construct some data structures, ranging from abstract-syntax trees to proof trees. The constructs must obey the following BNF

```
<regexp> ::= (empty)
           | (atom <atom>)
           | (any)
           | (seq <regexp> <regexp>)
           | (disj <regexp> <regexp>)
           | (star <regexp>)
           | (plus <regexp>)

<atom>   ::= ...any Scheme integer...

<name>   ::= ...any Scheme identifier...
```

Exercise 1

In this exercise we're asked to derive and draw abstract-syntax trees, AST, for the following regular expressions:

Expression 1

```
(seq (atom 1) (seq (atom 2) (seq (atom 3) (seq (atom 4) (empty)))))
```

Now, for deriving the AST, we're approaching the regular expression in a left-to-right fashion

```
<regexp> ->
(seq <regexp> <regexp>) ->
(seq (atom <atom>) <regexp>) ->
(seq (atom 1) <regexp>) ->
```

```
(seq (atom 1) (seq <regexp> <regexp>)) ->
(seq (atom 1) (seq (atom <atom>) <regexp>)) ->
(seq (atom 1) (seq (atom 2) <regexp>)) ->
(seq (atom 1) (seq (atom 2) (seq <regexp> <regexp>))) ->
(seq (atom 1) (seq (atom 2) (seq (atom <atom>) <regexp>))) ->
(seq (atom 1) (seq (atom 2) (seq (atom 3) <regexp>))) ->
(seq (atom 1) (seq (atom 2) (seq (atom 3) (seq <regexp> <regexp>)))) ->
(seq (atom 1) (seq (atom 2) (seq (atom 3) (seq (atom <atom>) <regexp>)))) ->
(seq (atom 1) (seq (atom 2) (seq (atom 3) (seq (atom 4) <regexp>)))) ->
(seq (atom 1) (seq (atom 2) (seq (atom 3) (seq (atom 4) (empty)))))
```

Drawing the derived AST is a straight forward ASCII art competition. Here we go!

```
<regexp>
   |
   |
(seq <regexp> <regexp>)
        |         \
        |          \
   (atom <atom>) (seq <regexp> <regexp>)
        |                |         \
        |                |          \
        1        (atom <atom>) (seq <regexp> <regexp>)
                         |             |         \
                         |             |          \
                         2        (atom <atom> (seq <regexp> <regexp>)
                                       |               |         \
                                       |               |          \
                                       3          (atom <atom>)  (empty)
                                                       |
                                                       |
                                                       4
```

This tree shows that we can move from the *root* <regexp> and construct the introduced regular expression by following all the paths per layer and manipulating the leaf nodes' values into a parenthesized list and keeping the operational vectors

$$(seq\ (atom\ 1)\ (seq\ (atom\ 2)\ (seq\ (atom\ 3)\ (seq\ (atom\ 4)\ (empty)))))$$

Expression 2

```
(seq (atom 1) (seq (atom 2) (seq (atom 3) (atom 4))))
```

Now, for deriving the AST, we're approaching the regular expression in a left-to-right fashion

```
<regexp> ->
(seq <regexp> <regexp>) ->
(seq (atom <atom>) <regexp>) ->
(seq (atom 1) <regexp>) ->
(seq (atom 1) (seq <regexp> <regexp>)) ->
(seq (atom 1) (seq (atom <atom>) <regexp>)) ->
(seq (atom 1) (seq (atom 2) <regexp>)) ->
(seq (atom 1) (seq (atom 2) (seq <regexp> <regexp>))) ->
(seq (atom 1) (seq (atom 2) (seq (atom <atom>) <regexp>))) ->
(seq (atom 1) (seq (atom 2) (seq (atom 3) <regexp>))) ->
(seq (atom 1) (seq (atom 2) (seq (atom 3) (atom <atom>)))) ->
(seq (atom 1) (seq (atom 2) (seq (atom 3) (atom 4))))
```

Whoo, time for another round of Draw That Tree!

```
<regexp>
   |
   |
(seq <regexp> <regexp>)
        |         \
        |          \
   (atom <atom>) (seq <regexp> <regexp>)
        |                |         \
        |                |          \
        1        (atom <atom>) (seq <regexp> <regexp>)
                         |             |         \
                         |             |          \
                         2        (atom <atom> (atom <atom>)
                                       |            |
                                       |            |
                                       3            4
```

By applying the same procedural approach as the previous expression, we start at <regexp>, move along all the paths and end up with the valid list of Scheme operations

$$(\mathit{seq}\ (\mathit{atom}\ 1)\ (\mathit{seq}\ (\mathit{atom}\ 2)\ (\mathit{seq}\ (\mathit{atom}\ 3)\ (\mathit{atom}\ 4))))$$

Expression 3

```
(seq (seq (empty) (seq (atom 1) (atom 2))) (seq (empty) (seq (atom 3) (atom 4))))
```

```
<regexp> ->
(seq <regexp> <regexp>) ->
(seq (seq <regexp> <regexp>) <regexp>) ->
(seq (seq (empty) <regexp>) <regexp>) ->
(seq (seq (empty) (seq <regexp> <regexp>)) <regexp>) ->
(seq (seq (empty) (seq (atom <atom>) <regexp>)) <regexp>) ->
```

```
(seq (seq (empty) (seq (atom 1) <regexp>)) <regexp>) ->
(seq (seq (empty) (seq (atom 1) (atom <atom>))) <regexp>) ->
(seq (seq (empty) (seq (atom 1) (atom 2))) <regexp>) ->
(seq (seq (empty) (seq (atom 1) (atom 2))) (seq <regexp> <regexp>)) ->
(seq (seq (empty) (seq (atom 1) (atom 2))) (seq (empty) <regexp>)) ->
(seq (seq (empty) (seq (atom 1) (atom 2))) (seq (empty) (seq <regexp> <regexp>))) ->
(seq (seq (empty) (seq (atom 1) (atom 2))) (seq (empty) (seq (atom <atom>) <regexp>))) ->
(seq (seq (empty) (seq (atom 1) (atom 2))) (seq (empty) (seq (atom 3) <regexp>))) ->
(seq (seq (empty) (seq (atom 1) (atom 2))) (seq (empty) (seq (atom 3) (atom <atom>)))) ->
(seq (seq (empty) (seq (atom 1) (atom 2))) (seq (empty) (seq (atom 3) (atom 4))))
```

Ready. Get set. Draw!

```
                <regexp>
                   |
                   |
           (seq <regexp> <regexp>)
                   /            \
                  /              \
     (seq <regexp> <regexp>)     (seq <regexp> <regexp>)
            /        |                   |         \
           /         |                   |          \
      (empty) (seq <regexp> <regexp>) (empty) (seq <regexp> <regexp>)
                      |        \                     /          \
                      |         \                   /            \
                 (atom <atom>) (atom <atom>)  (atom <atom>)  (atom <atom>)
                      |             |              |              |
                      |             |              |              |
                      1             2              3              4
```

Same procedure as last year, Miss Sophie?

(*seq* (*seq* (*empty*) (*seq* (*atom*1) (*atom* 2))) (*seq* (*empty*) (*seq* (*atom* 3) (*atom* 4))))

Expression 4

```
(seq (seq (seq (atom 1) (atom 2)) (atom 3)) (atom 4))
```

A bunch of nested *seq*, how nice. This is really reminiscent of SKIX procedures and the whole Λ-calculus way of thinking. As a curiousity, aren't these procedures known as currying?
Oh well, here comes the derivation of this subproblem's regular expression

```
<regexp> ->
(seq <regexp> <regexp>) ->
(seq <regexp> (atom <atom>)) ->
(seq <regexp> (atom 4)) ->
(seq (seq <regexp> <regexp>) (atom 4)) ->
```

```
(seq (seq <regexp> (atom <atom>)) (atom 4)) ->
(seq (seq <regexp> (atom 3)) (atom 4)) ->
(seq (seq (seq <regexp> <regexp>) (atom 3)) (atom 4)) ->
(seq (seq (seq (atom <atom>) <regexp>) (atom 3)) (atom 4)) ->
(seq (seq (seq (atom 1) <regexp>) (atom 3)) (atom 4)) ->
(seq (seq (seq (atom 1) (atom <atom>)) (atom 3)) (atom 4)) ->
(seq (seq (seq (atom 1) (atom 2)) (atom 3)) (atom 4))
```

Building a tree over the derived regular expression:

```
                     <regexp>
                        |
                        |
           (seq <regexp> <regexp>)
                   /           \
                  /             \
     (seq <regexp> <regexp>)    (atom <atom>)
            /           \            \
           /             \            \
(seq <regexp> <regexp>)  (atom <atom>)  4
        |         \            \
        |          \            \
 (atom <atom>) (atom <atom>)     3
        |           |
        |           |
        1           2
```

Same procedure as every year, James

(*seq* (*seq* (*seq* (*atom* 1) (*atom* 2)) (*atom* 3)) (*atom* 4))

Expression 5

```
(seq (seq (seq (seq (empty) (atom 1)) (atom 2)) (atom 3)) (atom 4))
```

```
<regexp> ->
(seq <regexp> <regexp>) ->
(seq <regexp> (atom <atom>)) ->
(seq <regexp> (atom 4)) ->
(seq (seq <regexp> <regexp>) (atom 4)) ->
(seq (seq <regexp> (atom <atom>)) (atom 4)) ->
(seq (seq <regexp> (atom 3)) (atom 4)) ->
(seq (seq (seq <regexp> <regexp>) (atom 3)) (atom 4)) ->
(seq (seq (seq <regexp> (atom <atom>)) (atom 3)) (atom 4)) ->
(seq (seq (seq <regexp> (atom 2)) (atom 3)) (atom 4)) ->
(seq (seq (seq (seq <regexp> <regexp>) (atom 2)) (atom 3)) (atom 4)) ->
(seq (seq (seq (seq (empty) <regexp>) (atom 2)) (atom 3)) (atom 4)) ->
(seq (seq (seq (seq (empty) (atom <atom>)) (atom 2)) (atom 3)) (atom 4)) ->
(seq (seq (seq (seq (empty) (atom 1)) (atom 2)) (atom 3)) (atom 4))
```

Abstract-syntax tree over the valid internal procedures obeying the given Backus-Naur Form:

```
                                 <regexp>
                                     |
                                     |
                            (seq <regexp> <regexp>)
                                     /        \
                                    /          \
                     (seq <regexp> <regexp>) (atom <atom>)
                             /          \          \
                            /            \          \
            (seq <regexp> <regexp>)  (atom <atom>) 4
               /              |            \
              /               |             \
(seq <regexp> <regexp>)  (atom <atom>)       3
        /        |            |
       /         |            |
   (empty)  (atom <atom>)     2
                 |
                 |
                 1
```

Following the layers of the tree for every layer, path, node and leaf yields

$$(\mathit{seq}\ (\mathit{seq}\ (\mathit{seq}\ (\mathit{seq}\ (\mathit{empty})\ (\mathit{atom}\ 1))\ (\mathit{atom}2))\ (\mathit{atom}\ 3))\ (\mathit{atom}\ 4))$$

Exercise 3

In this exercise, we're asked to construct proof trees for abstract-syntax trees crafted in the previous exercise. These new proof trees must follow the rules of the table below, such that e is a valid regular expression whenever the judgment (%regexp e) holds.

$$\text{EMPTY}\ \frac{}{\text{(\%regexp (empty))}}$$

$$\text{ATOM}\ \frac{\text{n is a Scheme integer}}{\text{(\%regexp (atom n))}}$$

$$\text{ANY}\ \frac{}{\text{(\%regexp (any))}}$$

$$\text{SEQ}\ \frac{\text{(\%regexp e1) (\%regexp e2)}}{\text{(\%regexp (seq e1 e2))}}$$

$$\text{DISJ}\ \frac{\text{(\%regexp e1)(\%regexp e2)}}{\text{(\%regexp (disj e1 e2))}}$$

$$\text{STAR}\ \frac{\text{(\%regexp e)}}{\text{(\%regexp (star e))}}$$

$$\text{PLUS}\ \frac{\text{(\%regexp e)}}{\text{(\%regexp (plus e))}}$$

$$\text{VAR}\ \frac{\text{x is a Scheme identifier}}{\text{(\%regexp (var e))}}$$

Expression 1

The given expression for this particular tree is

```
(seq (atom 1) (seq (atom 2) (seq (atom 3) (seq (atom 4) (empty)))))
```

Constructing a proof tree from the very first expression, we'll go bottom-up and see if things pan out. Expression 1 was confirmed valid in previous exercise, so we should see a similar result this time around, too.

$$\text{SEQ}\ \frac{\text{ATOM}\ \frac{\text{1 is a Scheme integer}}{\text{(\%regexp(atom 1))}} \quad \text{SEQ}\ \frac{\text{ATOM}\ \frac{\text{2 is a Scheme integer}}{\text{(\%regexp(atom 2))}} \quad \text{SEQ}\ \frac{\text{ATOM}\ \frac{\text{3 is a Scheme integer}}{\text{(\%regexp(atom 3))}} \quad \text{ATOM}\ \frac{\text{4 is a Scheme integer}}{\text{(\%regexp(atom 4))}}}{\text{(\%regexp(seq (atom 3) (seq (atom 4) (empty))))}}}{\text{(\%regexp(seq (atom 2) (seq (atom 3) (seq (atom 4) (empty)))))}}}{\text{(\%regexp (seq (atom 1) (seq (atom 2) (seq (atom 3) (seq (atom 4) (empty))))))}}$$

This procedure takes way too long to execute on the remaining trees in a digital fashion; I have, however, completed then on paper for training's sake.

Exercise 4

Given the following unit test

```
(define test-plus-and-times
  (lambda (candidate)
    (and (equal? (candidate 0 0)
                 0)
         (equal? (candidate 2 2)
                 4)
         ;;; don't add more tests here
         )))
```

Will + and $*$ pass the test?

Granted how functional programming environments use reverse polish notation, for example $(+\ 1\ 2) = 3$, these tests will pass for reasons explained below. Given a procedure as *candidate*, that procedure will return whatever resulting integer is computed, and check if it's equal to 0 or 4 respectively, in this task.

Logically, given (*candidate* 0 0) and checking if it's equal to 0, whether it is addition or multiplication then 0 handled precisely 0 times will yield 0. This means that (*equal*? (*candidate* 0 0) 0) will return #*t* if *candidate* is equal to the *plus* or *times* procedure.

An identical, mathematical reasoning goes for (*equal*? (*candidate* 2 2) 4); if *candidate* is equal to our *plus* procedure, then (*plus* 2 2) = 4. Checking 4 = 4 returns #*t*. The same goes for (*times* 2 2) = 4.
If tested against any other integers, the test will fail.

```
> (test-plus-and-times plus)
#t
> (test-plus-and-times times)
#t
```

Exercise 5

5.1

Define a traced version of *times* called *times_traced*, apply this procedure to 3 and 2 and make sense of the output.

```
(define times_traced
  (trace-lambda times (n1 n2)
    (if (zero? n1)
        0
        (plus n2 (times_traced (- n1 1) n2)))))
```

Applying this traced version to 3 and 2 results in the following:

```
> (times_traced 3 2)
|(times 3 2)
| (times 2 2)
| |(times 1 2)
| | (times 0 2)
| | 0
| |2
| 4
|6
6
```

Attempting to make sense of this, first let's look at the definition of the procedure. Our point of termination is (if (zero? $n1$)), that is, if our first input, in this case 3, reaches 0, then we return 0 and stop. As long as $n1$ is not 0, we dive deeper into the recursive rabbit hole with (- $n1$ 1). When it reaches 0, we're at a point where we can actually start doing integer computations, and from there on out we add the second parameter onto our 0 for a total number of n1 times, resulting in the integer represented by $n2$ a number of $n1$ times. $n2 \cdot n1$.

Using this fact, we can conclude that multiplying with this procedure, minimizing the recursion layers and optimizing the practical speed of multiplication is done by keeping $n2 \geq n1$.

5.2

Define a traced version of *times* that uses *plus_traced* instead of *plus*. Applied to 3 and 2, make sense of the output.

```
(define times_traced_twice
  (trace-lambda times (n1 n2)
    (if (zero? n1)
        0
        (plus_traced n2 (times_traced (- n1 1) n2)))))
```

Applying this traced version to 3 and 2 results in the following:

```
> (times_traced_twice 3 2)
|(times 3 2)
| (times 2 2)
| |(times 1 2)
| | (times 0 2)
| | 0
| |2
| 4
|(plus 2 4)
| (plus 1 4)
| |(plus 0 4)
| |4
| 5
|6
6
```

Expanding a bit on the previous explanation, we may now see more clearly how the additions are done per layer of recursion.

Exercise 18

Given some new rules of mathematics, implement predicates about whether a given natural number is ternary, pre-ternary or post-ternary.

The new rules specify that $\pi = 3$, and a *ternary* number is defined as $\forall n \bmod 3 = 0$. The direct successor of a ternary number is called *post-ternary*, that is any ternary number +1; the opposite is a *pre-ternary* number that is any ternary number -1.

Theoretically, constructing a procedure that tests if any input n is divisible by π is best written using the modulo operater, checking the remainder, such that if $n \bmod \pi = 0$ it is a pure divisor of π. This is valid for $\{x\pi \mid x \in \mathbb{Z}\}$.

With ternary numbers defined, let's handle pre-ternary numbers. The same reasoning goes for this as for ternary numbers, yet with one subtle difference: pre-ternary numbers are the direct predecessor of ternaries. This means that numbers like $1\pi - 1,\ 2\pi - 1,\ 3\pi - 1,\ ..., n\pi - 1$ are pre-ternary, $\forall n > 0$, and we will have a remainder of 2 for each modulo if this is the case, because $2 \bmod \pi = 2$, $5 \bmod \pi = 2$, etc. We may now conclude that using $(x \bmod \pi) - 2 = 0 \Rightarrow$ pre-ternary numbers. This is valid for $\{x\pi - 1 \mid x \in \mathbb{Z}\}$.

The last possibility, post-ternary numbers, are $1\pi - 2,\ 2\pi - 2,\ 3\pi - 2,\ ..., n\pi - 2$, and this goes for $\forall n > 0$. Using the usual arguments, we arrive at a set of numbers described as post-ternary $= \{x\pi - 2 \mid x \in \mathbb{Z}\}$

Implementing a simple yet functional way of testing which category the input integer is in, the straight-forward way would be to simlpy compare the remainder of $x \bmod \pi$. This, in Scheme, is (remainder x 3), due to our new rule of $\pi = 3$. Another rule is required at this stage to start writing Scheme code: each of the predicates in Version 0 should use *remainder*. First, let's set up a few unit-tests based on our observations.

```
(define test-ternary
  (lambda (candidate)
    (and (equal? (candidate 1) #f)
         (equal? (candidate 2) #f)
         (equal? (candidate 3) #t)
         (equal? (candidate 4) #f)
         (equal? (candidate 5) #f)
         (equal? (candidate 6) #t))))

(define test-preternary
  (lambda (candidate)
    (and (equal? (candidate 1) #f)
         (equal? (candidate 2) #t)
         (equal? (candidate 3) #f)
         (equal? (candidate 4) #f)
         (equal? (candidate 5) #t)
         (equal? (candidate 6) #f))))
```

```
(define test-postternary
  (lambda (candidate)
    (and (equal? (candidate 1) #t)
         (equal? (candidate 2) #f)
         (equal? (candidate 3) #f)
         (equal? (candidate 4) #t)
         (equal? (candidate 5) #f)
         (equal? (candidate 6) #f))))
```

I tried to have a little fun with version0 and version1. The true three-way predicate is set up in version2. A sample code may look like:

Version 0

```
(define is_ternary_v0
  (lambda (x)
    (if (zero? (remainder x 3))
      (display "Input is ternary")
      (if (equal? (remainder x 3) 1)
        (display "Input is post-ternary")
          (if (equal? (remainder x 3) 2)
            (display "Input is pre-ternary"))))))
```

Running the code, we get the output

```
> (is_ternary_v0 2)
Input is pre-ternary
> (is_ternary_v0 3)
Input is ternary
> (is_ternary_v0 4)
Input is post-ternary
```

Version 1

For the next version, Version 1, each of the predicates should be (self-)recursive and each recursive call should decrement the argument by 3. A sample code may look like this:

```
(define is_ternary_v1
  (lambda (x)
    (if (zero? x)
      (display "Input is ternary")
      (if (equal? x 1)
        (display "Input is post-ternary")
          (if (equal? x 2)
            (display "Input is pre-ternary")
              (is_ternary_v1 (- x 3)))))))
```

Yielding the following output:

```
> (is_ternary_v1 20)
Input is pre-ternary
> (is_ternary_v1 21)
Input is ternary
> (is_ternary_v1 22)
Input is post-ternary
```

Version 2

Another rule is added: each of the predicates in Version 2 should be mutually recursive and each recursive call should decrement the argument by 1. A sample code may look like this:

```
(define is_ternary?
  (lambda (x)
    (if (= x 0)
      #t
      (is_pre-ternary? (- x 1)))))
(define is_pre-ternary?
  (lambda (x)
    (if (= x 0)
      #f
      (is_post-ternary? (- x 1)))))
(define is_post-ternary?
  (lambda (x)
    (if (= x 0)
      #f
      (is_ternary? (- x 1)))))
      }
```

Running this code, asking if (is-ternary 3), it'll do several recursive bounces, going from is-ternary into is-pre-ternary then into is-post-ternary and back up into is-ternary with an x value of 0, returning $\#t$. A few outputs shows the working code.

```
> (is_ternary? 11)
#f
> (is_ternary? 12)
#t
> (is_ternary? 13)
#f
>
```