Homework #2: BNFs, Higher-Order Functions, Typed Racket

Out: Monday, April 12, 2017, Due: Sunday, April 30, 23:55

Administrative

This is another introductory homework, and it is for **work and submission in pairs (single submissions will also be accepted)**. In this homework you will be introduced to the course language and some of the additional class extensions as well as BNF.

In this homework (and in all future homeworks) you should be working in the "Module" language, and use the appropriate language using a `#lang` line. You should also click the "Show Details" button in the language selection dialog, and check the "Syntactic test suite coverage" option to see parts of your code that are not covered by tests: after you click "run", parts of the code that were covered will be colored in green, parts that were not covered will be colored in red, and if you have complete coverage, then the colors will stay the same. Note that you can also set the default language that is inserted into new programs to `#lang pl`, to make things more convenient. There are some variants for the `pl` language for various purposes — in particular, `#lang pl/untyped` will ignore all type declarations, and will essentially run your code in an untyped Racket.

The language for this homework is:

```
#lang pl 02
```

As in previous assignment, you need to use the special form for tests: `test`.

Reminders (this is more or less the same as the administrative instructions for the previous assignment):

This homework is for work and submission in pairs.

Integrity: Please do not cheat. You may consult your friend regarding the solution for the assignment. However, you must do the actual programming and commenting on your own!! This includes roommates, marital couples, best friends, etc… I will be very strict in any case of suspicion of plagiary. Among other thing, students may be asked to verbally present their assignment.

Important – If you consult any person (other than your partner) or any other source – You must indicate it within your personal comments. Also the contribution of each of you to the solution must be detailed.

Comments: Submitted code for each question should include at least two lines of comments with your personal description of the solution, the function and its type. In addition, you should comment on the process of solving this question – what were the main difficulties, how you solved them, how much time did you invest in solving it, did you need to consult others. Specifically, describe the portion of the work that each of the partners did.

VERY IMPORTANT NOTE:

A solution without proper and elaborate PERSONAL comments describing your work process may be graded 0.

Tests: For each question, you should have enough test cases for complete coverage (DrRacket indicates covered expressions with colors for covered and uncovered source code, unless your code is completely covered). See below on the way to create tests.

Important: Your tests should cover your whole code; otherwise the server will heavily penalize your submission. You should not have any uncovered expressions after you hit "Run" — it should stay at the same color, indicating complete coverage. Furthermore, the server will run its own tests over your code, which means that you will not be able to submit code that does not work. Reminder: this means that most of the focus of this homework is put on the contract and purpose statements, good style (indentation, comments, etc), and good tests.

General note: Code quality will be graded. Write clean and tidy code. Consult the Style Guide, and if something is unclear, ask questions on the course forum.

The test form can be used to test that an expression is true, that an expression evaluates to some given value, or that an expressions raises an error with some expected text message. For example, the three kinds of tests are used in this example:

```
#lang pl
(: smallest : (Listof Number) -> Number)
(define (smallest l)
  (match l
    [(list)       (error 'smallest "got an empty list")]
    [(list n)     n]
    [(cons n ns)  (min n (smallest ns))]))
(test (smallest '(5 7 6 4 8 9)) => 4)
(test (zero? (smallest '(0 1 2 3 4))))
(test (smallest '()) =error> "got an empty list")
```

In case of an expected error, the string specifies a pattern to match against the error message. (Most text stands for itself, "`?`" matches a single character and "`*`" matches any sequence of characters.)
Note that the `=error>` facility checks only errors that *your* code throws, not Racket errors. For example, the following test will not succeed:

```
(test (/ 4 0) =error> "division by zero")
```

The code for all the following questions should appear in a single .rkt file named <your ID>_1 (e.g., 222222222_333333333_1 for two students whose ID numbers are 222222222 and 333333333).

1. BNF (LE)

a. In class we have seen the grammar for AE – a simple language for "Arithmetic Expressions".

   Write a BNF for "**LE**": a similarly simple language of "List Expressions". Valid 'programs' (i.e., word in the **LE** language) in this language should correspond to Racket expressions that evaluate to S-expressions holding numbers and symbols. The valid operators that can be used in these expressions are **`cons`**, **`list`**, and **`append`**, and **`null`** is a valid expression too. Note that the grammar should allow only **`cons`** with an expression that represents a list (no numbers or symbols as the second part of a cons expression), **`list`** can have any number of arguments (including zero arguments), and **`append`** can have any number of *list* arguments. Use **curly brackets** as we do in the AE language in class, i.e., complex expressions should start with a '{' and end with a '}'.

Plain values are either numbers or quoted symbols — no quoted lists, and only the quote character can be used. You can use ⟨`num`⟩ and ⟨`sym`⟩ as known numbers and symbols terminals. (Note that we assume that ⟨`sym`⟩ corresponds to a Racket symbol, and that *does not* include the quote (`'`) character, i.e., the quote (`'`) character should appear in your BNF.)
For example, some **valid** expressions in this language are:

```
Null

12

'boo

{cons 1 {cons 'two null}}

{list 1 2 3}

{list {list {list {list 1 2 3}}}}

{append {list 1 2} {list 3 4} {list 5 6}}

{list}

{append}

{cons 1 {cons {append {cons 'x null} {list 'y 'z}}
null}}

{cons 1 null}

{list 'a}
```

but the following are **invalid** expressions:

```
{cons 1 2}

{list {3 4}}

{quote boo}

{append 1 {list 2 3} 4}

{cons 1 2 null}

{cons 1 {2}}

{cons 1 '{}}

'{1 2 3}

{cons '1 null}

{list ''a}

{car {list 1 2}}
```

Remark: Note that some of these expressions become valid Racket S-expressions after applying the `string->sexpr` procedure to them — but this question specifies a *restricted* set of valid Racket S-expressions.
NOTE: The use of ellipsis (`'...'`) or '*' is ***not*** allowed here (find ways within the BNF framework to specify zero-or-more occurrences of a previous piece of syntax). Use λ to specify the empty string.

Important remark: Your solution should only be a BNF and not a code in Racket (or in any other language). You cannot test your code!!! Indeed, your answer should appear inside a comment block (write the grammar in a #|---|# comment).

b. Add to your BNF a derivation process for 3 different **LE** expressions, in which all plain values are sub-words (of length 2 or 3) of either your name or ID number. E.g., if your name is Baba Ganush and your ID number is 123456789, then you might want to show how you derive the word **(cons 345 (cons (append (cons 'Bab null) (list 'Gan 'sh)) null))** from your BNF (make sure you use the full power of your BNF). You may either provide a derivation tree or a series of replacements starting with <LE> and ending with your string. Mark each derivation rule by an index (use "$\overset{(i)}{=>}$" to state that in a certain step, you have used rule number i of your BNF).

2. Higher Order Functions

Write a function called **sequence** that receives a function `f`, and two values `first` and `last`. The function should return a list of values that starts with `first` and ends with `last`, and for each consecutive pair of values `a1`, `a2` in the list, the application (`f a1`) results in `a2` --- that is, `(equal? (f a1) a2)` should be `#t`. You may assume that the arguments in any application of **sequence** are such that after a finite number of applications of `f` to `a1,` the result is `a2`, i.e., any test should result in a finite sequence (list).

A few test cases will clarify further how this function works:

```
(test (sequence add1 1 1) => (list 1))

(test (sequence add1 1 5) => (list 1 2 3 4 5))

(test (sequence sub1 5 1) => (list 5 4 3 2 1))

(test (sequence sqrt 65536 2) =>
                  (list 65536 256 16 4 2))
(test (sequence not #f #t) => (list #f #t))
```

You can also use these tests in your code (as well as add more tests).
Note that Typed Racket cannot properly infer types for uses of polymorphic higher order functions, so if you want to try this function with a list function, you need to use `inst` to ``instantiate'' a polymorphic function with a specific type. For example:

```
(test (sequence (inst rest Number) (list 1 2 3) null)
           => (list (list 1 2 3) (list 2 3) (list 3) null))
```

(It is also possible by providing a redundant function just for it's different type, for example

```
(: num-rest : (Listof Number) -> (Listof Number))
            (define (num-rest l) (rest l))
```

but `inst` is obviously more convenient.)
Pay attention to the type you declare for `sequence` --- if you use something like `(Number -> Number)` for the first argument you will reduce the utility of this function, since it can only work with numeric functions.
Your solution must also be usable with other types too, as shown in the above tests. Note, however, that Typed Racket has subtypes --- and this can lead to a different extreme where the type declaration is too general:

```
(: sequence : Any Any Any -> Any)
```

This is also not a good type declaration, because it provides very little information about the function. Specifically, it will not allow you to use even apply the first argument, since it is not known that it is a function. A proper type will need to be a ***polymorphic function***: the type of `sequence` should refer to ``some type A'' in the following way:

```
(: sequence : (All (A) << -- fill in -- >>))
```

3. Defining and Using a New Type (INTSET)

In this question you will implement a type that is often useful: a possibly sparse set of integers. For the purpose of this question, we represent such sets as one of:

- A single integer;
- An inclusive range of integers;
- A combination of two integer sets.

Note that for simplicity there are no empty sets.

Begin by a type definition. Fill in the following template:

```
(define-type INTSET
    [Int        << -- fill in -- >>]
    [Range      << -- fill in -- >>]
    [2Sets      << -- fill in -- >>])
```

(2Sets is a valid Racket identifier.)

An integer set is said to be in **normal form** if the following holds:

1. no numbers are repeated (singleton numbers and ranges have no overlap),
2. all ranges are formed from two numbers, where the first is smaller than the second,
3. when 2Sets unions are also in order (the set on the left side is completely below the set on the right), and
4. there **is** some gap between the two subsets in a 2Sets value (it cannot be that the left side one contains a number n and the right side one contains $n + 1$).

In other words, a normalized integer set is one that is completely ordered and has the shortest number of elements. Implement an **intset-normalized?** predicate (should return a Boolean value) that checks this.

It is relatively easy to define this function given two utility functions, intset-min and **intset-max**, that return the smallest and biggest members of an INTSET respectively. But these two functions are quite similar, so implement a third function, **intset-min/max**, that can return either result, based on a comparator function. Your **intset-min/max** should be implemented such that the other two utility functions are defined as follows:

```
(: intset-min : INTSET -> Integer)
 ;; Finds the minimal member of the given set.
(define (intset-min set) (intset-min/max set <))

(: intset-max : INTSET -> Integer)
 ;; Finds the maximal member of the given set.
(define (intset-max set) (intset-min/max set >))
```

Note that using this is not going to be too efficient; specifically, each side of a 2Sets construct is scanned twice (which makes the run-time asymptotically worse than a single scan --- not just double!). Finding an implementation that performs a single scan is more difficult, however, so it is not necessary. But this does *not* mean that you should neglect common sense --- for example, don't do more scans than needed.

3.2. BNF for INTSET

The INTSET type is useful. One example is for representing a range of pages to print. We will use <int> as a "known terminal" that specifies any integer. A single page will be denoted by a single Integer. An interval will be denoted by two Integers and a dash in between them and rapped with curly brackets (the three inner parts must be separated by at least one space between every two of them). Finally, a combination of two sets will be denoted as two expressions rapped with curly brackets.

For example, some **valid** expressions in this language are:

```
23
{1 - 99}
{ 1 {33 - 44}}
{{22 - 33} {1 2}}
{{1 2} {1 - 4}}
```

However, the following are **invalid** expressions:

```
{2}
{1 2 {3 4}}
1 - 2
{1-4}
```

Write a BNF with starting non-terminal <IntSet> for such specification. Write the grammar in a #|...|# comment. Make sure that your BNF is unambiguous.