A4.3 Embedding Troll DSL in PLD LISP (30%)

I have implemented a subset of Troll and its random roll semantics as an embedded DSL in PLD LISP using the provided simple linear congruential random number generator. I have implemented all the operations from the Troll subset: n, x, $d\ e$, $e_1\ d\ e_2$, $e_1\ \#\ e_2$, $e_1 + e_2$, $e_1 - e_2$, $e_1 < e_2$, $e_1 > e_2$, *sum* e, *count* e, $\{e_1, \ldots, e_n\}$, *min* e, *max* e, *least* $e_1 e_2$, *largest* $e_1\ e_2$, *choose* e, $x := e_1; e_2$, *accumulate* $x := e_1$ *while* e_2, where n is a number, x is a variable, and e_i are Troll expressions.

a) Describe how the syntax of each of the above is represented as embedded syntax in PLD LISP. Discuss nontrivial choices.

I decide on a representation of the syntax associated with Troll expressions and operations in PLD LISP that is familiar to the syntax in Troll while being easy to implement in the PLD LISP host language with its own constructs. Recall, Lisp is a homoiconic language, meaning its programs can be manipulated as data. Thus, the internal syntax representation can be deduced by reading the program itself and code it analogous to data. In our case, PLD Lisp proves useful, since we can extend it with new constructs that have similar syntax to Lisp but with a new semantics that is otherwise unavailable to Lisp. Thus, I take this to be the way to implement Troll as an embedded DSL in PLD Lisp. Namely, I stick with the Lisp philosophy of representing troll expressions in my DSL as a list structure so that I avoid needing a separate parser. This means my embedded Troll DSL exploits the built-in parsing of symbolic expression (s-expressions) in PLD Lisp so my DSL can be parsed into syntax with no effort from my implementation.

In this regard, I have made the following design choices. Firstly, I prefer a functional style, since the host language, PLD Lisp, is based on that. That means writing Troll operators as functions that manipulate Troll expressions as data. Secondly, I avoid infix operators and keep everything prefix so that I can reuse certain common operators in PLD Lisp and stay true to the prefix notation used in PLD Lisp. Personally, I find it confusing to switch between different notations and I do not want to implement my own parsing of infix expressions when I can get prefix notation for free. In order to worry less about precedence and associativity, I strive to fully parenthesize Troll expressions. Finally, I use closures in PLD Lisp to delay the evaluation of Troll expressions, which enables me to reuse a Troll expression in my embedded syntax to generate different results. According to the Troll manual, all die-roll results in either a single integer value or an unordered collection of such values. Thus, a die-roll definition is an expression that use numbers and operators to create die-rolls. Consequently, we can represent values with symbolic expressions in LISP. More specifically, we represent Troll values as lists of integers where a single result is represented by a singleton list with one element and a collection of results is represented by a list of integers in PLD Lisp. I use the built-in representation of integers and variables in PLD Lisp with the assumption that variables only contain letters and no digits as in the Troll manual specification. I could use the built-in arithmetic operators in PLD Lisp. However, since I chose to represent all Troll expressions by lists, I have written my own addition and subtraction functions to add and subtract Troll expressions represented as singleton lists.

The embedded syntax of Troll expressions and operators in PLD Lisp are listed in the table below. Notice, equivalent syntax signify use of same syntax built into PLD Lisp.

Troll Syntax	**PLD Lisp Embedded Troll Syntax**	**Meaning**	**Example**
n	*n*	number	2
x	*x*	variable	(define x 2)
d *e*	(d *n*)	roll one n-sided die labelled 1-n where n = e	(d 2)
e_1 d e_2	(mdn e_1 e_2)	roll m d n where m = e_1 and n = e_2	(mdn 2 3)
e_1 # $e2$	(# *m* (lambda () *e*))	m samples of e where m = e_1 and e = e_2	(# 2 (lambda () (d 2)))
e_1 + e_2	(plus e_1 e_2)	arithmetic addition on single values	(plus (d 2) (d 3))
e_1 - e_2	(minus e_1 e_2)	arithmetic subtraction on single values	(minus (d 2) (d 3))
e_1 < e_2	(filter e_1 < e_2)	filter: keep values from e_2 less than e_1	(filter 7 < (mdn 10 10))
e_1 > e_2	(filter e_1 > e_2)	filter: keep values from e_2 greater than e_1	(filter 7 > (mdn 10 10))
sum *e*	(sum e)	add up values in collection *e*	(sum (mdn 5 6))
count *e*	(count e)	count values in collection *e*	(count (filter 4 < (mdn 10 6)))
{e_1, ..., e_n}	(union e_1 (union e_2 (union ... (union e_n))))	union of e_1, ..., e_n = e_1 *U...U* e_n	(union 2 (union (mdn 3 6) (d 2)))
min *e*	(min e)	minimum value in collection *e*, same as (least 1*e*)	(min (mdn 2 3))
max *e*	(max e)	minimum value in collection *e*, same as (largest 1*e*)	(max (mdn 2 3))
least e_1e_2	(least *n e*)	*n* least values in collection *e* where *n* = e_1 and *e* = e_2	(least 3 '(3 2 10 2 8)) = (2 2 3)
largest e_1e_2	(largest *n e*)	*n* largest values in collection *e* where *n* = e_1 and *e* = e_2	(largest 3 '(1 2 10 10)) = (2 10 10)
choose *e*	(choose e)	choose value from collection of size n, uniformly distributed between 1 and n	(choose '(1 2 3))
x := e_1; e_2	(let e_1 (lambda (x) e_2)) or ((lambda (x) e2) e1)	bind *x* to value of e_1 in e_2	(let (mdn 10 8) (lambda (x) (+ (min x) (sum x)))) or ((lambda (x) (+ (min x) (sum x))) (mdn 10 8))
accumulate *x* := e_1 *while* e_2	(accumulate (lambda () e1) (lambda (x) e2))	repeatedly evaluate $e1$ while e_2 becomes true (non-empty). Return union of all values in up to *n* = 12 iterations. e_2 on the form (filter *n operator e*), not *e* operator *n*	(accumulate (lambda () (d 10)) (lambda (x) (filter 10 > x)))
Number of rolls	(repeat (lambda () e) n)	repeats random results from evaluating *e*, *n* times	(repeat (lambda () (mdn 1 1)) 5)

Table 1: Troll DSL Syntax

One key lesson I have learned from building an embedded Troll DSL in PLD Lisp is that it gives you the opportunity to exploit the syntax of the host language, although this also means your expressiveness is limited by the constructs of the chosen host language. On one hand, I found the ability to treat Troll expressions as symbolic list expressions (similar to collections) in PLD Lisp to be very powerful. On the other hand, having chosen to express Troll operations as functions in PLD Lisp, I sometimes found the syntax a bit awkward to read and work with when delaying the evaluation of Troll expressions by using closures. However, this compromise allowed me to easily implement all of the Troll expressions and methods with a very Lisp-like syntax. Particularly, I liked the productivity that arose from treating Troll expressions as a list (collection) data structure. Nonetheless, I think my solution is limited by the syntax of the implementation language, which is less readable when given the combined use of prefix notation and closures for delayed evaluation of random die roll results in Troll expressions. Thus, one might also consider implementing the Troll DSL as a standalone language to fully express the domain-specific notation, although this comes at the cost of implementing a full compiler or interpreter. To sum up, I chose to represent the syntax of the embedded Troll DSL with a list datatype, since it was already available as a symbolic expression in PLD Lisp, which meant no parser was required.

Terminal session showing how to compile and run PLD Lisp and load Troll DSL:

```
> source compile.sh
> mono lisp.exe
PLD LISP version 2.1
> (load troll)
```

b) Implement the random roll semantics for the above subset of Troll. You can assume that Troll programs are well formed. You should be able to specify how many random results you want. Discuss nontrivial aspects of the implementation. Your implementation should be included as a file Troll.le with your report.

The following *repeat (e n)* function allows you to repeat obtaining a random result from the evaluation of a Troll expression *e*, *n* times. This is equivalent to "Number of rolls".

```
(define repeat (lambda
                    (e 0) ()
                    (e n) (cons (e) (repeat e (- n 1)))
))
(repeat (lambda () (mdn 1 1)) 3) ; e.g. repeat: n=3 1d1 = ((1) (1) (1))
```

The following snippet lists all helper functions in the Troll DSL. Notice, I have shortened the comments from *troll.le* and replaced the PLD Lisp comment symbol '\' with ';' for better syntax highlightning in Latex with the Minted package for Lisp.

```
; Fold HoF to accumulate results when traversing die roll results in list collection
(define foldl (lambda
                (f acc ()) acc
                (f acc (x . xs)) (foldl f (f acc x) xs)
))

; helper function uses to simulate random die rolls
(define dieRoll (lambda (n) (list(+ (rnd n) 1))))

; Remove helper function for least and largest
(define remove (lambda
                    (n ()) ()
                    (n list) (if (= n (car list)) (cdr list)
                             (cons (car list) (remove n (cdr list))))
))

; helper function for least and largest
(define lHelper (lambda
                    (0 e opr) () ; n elements taken (assume n >= 0)
                    (n () opr) () ; no more elements to take
                    (n e opr) ((lambda (m) (cons m (lHelper (- n 1) (remove m e) opr)))
                               (car (opr e)))
))

; helper function for choose
(define nth (lambda
               (n list) (if (= n 1) (car list) (nth (- n 1) (cdr list)))))
```

Here is the list of operators in the embedded Troll DSL:

```
(define d (lambda (e2) (dieRoll e2)))

(define mdn (lambda
                  (0 e2) ()
                  (e1 e2) (append (d e2) (mdn (- e1 1) e2))))

(define \# (lambda
             (0 e) ()
             (n e) (append (e) (\# (- n 1) e))))

(define filter (lambda
                 (e1 operator ()) ()
                 (e1 operator list) (if (operator e1 (car list))
                                        (cons (car list) (filter e1 operator (cdr list)))
                                        (filter e1 operator (cdr list)) )))

(define sum (lambda (e) (list (foldl + 0 e))))

(define count (lambda (e) (list (foldl (lambda (acc x) (+ acc 1)) 0 e))))

(define union (lambda (e1 e2) (append (if (number? e1) (list e1) e1)
                                      (if (number? e2) (list e2) e2))))

(define min (lambda
                 (e) (list (foldl (lambda (acc x) (if (< x acc) x acc)) (car e) (cdr e)))))

(define max (lambda
                 (e) (list (foldl (lambda (acc x) (if (> x acc) x acc)) (car e) (cdr e)))))

(define least (lambda (n e) (lHelper n e min)))

(define largest (lambda (n e) (reverse (lHelper n e max))))

(define choose (lambda (e) ((lambda (n) (list (nth n e)))
                            (car (dieRoll (car (count e)))))))

(define let (lambda (e1 e2) ((lambda (x) (e2 x)) e1)))

(define accumulate (lambda (e cond) (accHelper e cond 12)
                           (e cond n) (accHelper e cond n)))
(define accHelper
(lambda (e cond 1) ((lambda (res) (if (cond res) (accHelper e cond 1) res) ) (e))
(e cond n) ((lambda (res) (if (cond res) (append res (accHelper e cond (- n 1))) res))(e))))
```

Implementation

My implementation of the Troll DSL is based on the assumption that all die roll results are either represented as singleton values (i.e. list with single integer result value) or collections (i.e. list with multiple integer result values), which may be manipulated as list symbolic expressions in PLD Lisp. Further, I use functions to represent Troll expressions and operations. The random roll semantics are executed by exploiting pattern matching on list collections that represent die roll results. Closures are used to delay the evaluation of random die results so that I can reuse Troll expressions to generate multiple different results. All my operators follow prefix notation to stay consistent with the prefix notation of PLD Lisp. I sometimes use higher order functions like fold to accumulate die roll results in collections in a variety of ways. Otherwise, I define small helper functions that help me break down the problems that arise when trying to aggregate die roll results in many different ways. Essentially, a lot of the functionality in the Troll DSL boils down to functions that manipulate list data that represents by Troll collections of die roll results. The most obvious improvement to my implementation would be to get rid of the need to pass on closures as arguments to my functions, since it is cumbersome for the client user to represent Troll expressions wrapped in inside closures, when wanting to reuse it to generate multiple results as part of a random die roll experiment in the Troll DSL.

c) Write a short guide of how to use your implementation to generate random results. Show examples of use. This should test all constructions and at least include equivalents of the following Troll expressions: $d12 + d8$, $min\ 2d20$, $max\ 2d20$, $sum\ largest\ 3 4d6$, $count\ 7 < 10d10$, $choose\ \{1, 3, 5\}$, $accumulate\ x := d6\ while\ x > 2$, and $x := 3d6; (min\ x) + (max\ x)$. Generate 10 different random results for each of these and show the PLD session in your report.

Please see table 1 for a full overview of how to use the Troll DSL embedded.

The table shows all the required Troll constructs and their equivalents in the Troll DSL embedded in PLD Lisp, along with explanations and examples ready to be executed in the PLD Lisp interpreter.

I have listed the PLD session on the next page containing random results repeated 10 times with the mentioned Troll expressions. Notice, the implementation should work with other valid Troll expressions too if you follow the table guide with valid inputs.

d) If you did not manage to get everything to work, describe what is missing and what thoughts you have about implementing the missing parts.

To the best of my knowledge, I believe my implementation covers the requirements.

Session: 10 random results for Troll expressions

```
% mono lisp.exe
PLD LISP version 2.1
> (load troll)
> > = car
> = cdr
> = caar
> = cadr
> = cdar
> = cddr
> = list
> = length
> = append
> = reverse
> = equal
> = ()
> > = seed
> = rnd
> = ()
> = repeat
> = foldl
> = dieRoll
> = remove
> = plus
> = minus
> = d
> = mdn
> = #
> = filter
> = filter2
> = sum
> = count
> = union
> = min
> = max
> = least
> = largest
> = lHelper
> = choose
> = nth
> = let
> = accumulate
> = accumulateHelper
> = ()
```

```
# NB: I use the repeat function to get N=10 random die roll results as specified.
# Format: repeat (lambda () e) 10) where e = Troll expression and N = 10.

# d12 + d8 = (plus (d 12) (d 8))
> (repeat (lambda () (plus (d 12) (d 8))) 10)
= ((7) (9) (8) (3) (3) (14) (8) (15) (15) (4))

# min 2d20 = (min (mdn 2 20))
> (repeat (lambda () (min (mdn 2 20))) 10)
= ((2) (6) (4) (15) (9) (2) (6) (9) (10) (5))

# max 2d20 = (max (mdn 2 20))
> (repeat (lambda () (max (mdn 2 20))) 10)
= ((17) (14) (14) (15) (17) (17) (10) (17) (18) (9))

# sum largest 3 4d6 = (sum (largest 3 (mdn 4 6)))
> (repeat (lambda () (sum (largest 3 (mdn 4 6)))) 10)
= ((11) (6) (9) (12) (13) (9) (10) (13) (13) (14))

# count 7 < 10d10 = (count (filter 7 < (mdn 10 10)))
> (repeat (lambda () (count (filter 7 < (mdn 10 10)))) 10)
= ((0) (4) (7) (3) (4) (3) (1) (4) (1) (2))

# choose (1, 3, 5) =  (choose '(1 3 5)) or (choose (union 1 (union 2 3)))
> (repeat (lambda () (choose (union 1 (union 2 3)))) 10)
= ((1) (5) (1) (1) (3) (5) (1) (1) (3) (5))

# accumulate x:=d6 while x>2 = (accumulate (lambda () (d 6)) (lambda (x) (filter 2 < x)))
> (repeat (lambda () (accumulate (lambda () (d 6)) (lambda (x) (filter 2 < x)))) 10)
= ((5 2) (6 5 3 2) (4 3 5 2) (4 6 3 1) (6 6 1) (3 2) (3 3 6 4 1) (1) (5 6 5 5 4 2)
(4 5 5 6 5 1))

# x := 3d6; (min x) + (max x) = (let (mdn 3 6) (lambda (x) (plus (min x) (max x))))
> (repeat (lambda () (let (mdn 3 6) (lambda (x) (plus (min x) (max x))))) 10)
= ((9) (7) (7) (8) (9) (11) (7) (8) (7) (9))
```