PLD Assignment 4

`sortraev` (Anders Lietzen Holst, wlc376)

April 9, 2021

1 A4.1

1.1 A4.1.a

- *Write a **SIS** program that starts with $a = n$, $b = 0$, and $c = 0$, and which terminates with $a = n$, $b = 0$ and with c equal to the sum of all numbers from 1 to n.*

I am looking for a program which computes:

$$\sum_{i=1}^{n} i$$

This sum can be rewritten as a nested sum in which the body of the inner summation is a constant 1, since the sum of i ones is precisely i:

$$\sum_{i=1}^{n} i = \sum_{i=1}^{n} \sum_{j=1}^{i} 1$$

This is helpful since we can accumulate constants using a finite sequence of variable increments - in this case, accumulating the number 1 corresponds to a single SIS variable increment. However, before translating the math to SIS code, we shift the bounds of the inner loop down by one - this is legal because the summation maintains the same number of iterations, and we do not have to modify the body because it is constant in the value of i (this shift will be convenient later):

$$\sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \sum_{i=1}^{n} \sum_{j=0}^{i-1} 1 \tag{1}$$

We, of course, cannot shift the outer loop since *its* body is *not* constant in i.

At this point I want to formulate the mathematical expression in the SIS language. I use `c` as accumulator for the result of the summation; then, for each 1 in the body of the summation corresponds I want to increment `c` by one.

The SIS code then becomes:

```
for i = b to a {
  i++;
  for j = b to i {
    c++;
    j++;
  };
}
```

As explained it was convenient to shift the inner summation bounds down by one - this was done so we could use `b = 0` for the lower bound. Notice then that `i` is incremented *before* the inner loop of index `j` rather than after; this is to achieve the outer loop going from 1 to and *including* n, rather than from 0 to and including $(n-1)$. As such we get $(k+1)$ iterations of the inner loop on the k'th iteration of the outer loop.

1.2 A4.1.b

- *Show that SIS is a reversible language.*

The course notes state that "*A programming language is reversible if every operation it performs is reversible*", and so I intend to show that SIS is a reversible language by showing how each of its operations is reversible. I do so by constructing reversal rules for each. First, here are my reversal rules:

$$\begin{aligned} R(\texttt{A++}) &= \texttt{A--} \\ R(\texttt{A--}) &= \texttt{A++} \\ R(\texttt{S1; S2}) &= R(\texttt{S2});\ R(\texttt{S1}) \\ R(\texttt{for A = B to C \{ S \}}) &= \texttt{for A = C to B } \{R(\texttt{S})\} \end{aligned}$$

where $R(\texttt{S})$ is the reversal of some statement `S`, and where `A`, `B`, and `C` are program variables.

The first two rules state that an increment is inversed by a decrement and vice versa.

The third rule states that to reverse a sequence of two statements, we must reverse each statement individually and then execute the resulting statements in reverse order. Since reversing the order of statements is an

associative operation, this reversal rule applies to arbitrary sequences of statements.

The fourth reversal rule is the interesting one. It states that to reverse a `for` loop, we must swap the upper and lower bounds of the loop and apply reversal to the loop body `S`. *Why?*

If a program p terminates without error, then any loop in p must have executed a finite number of iterations [1]. A loop whose body is `S` and which has terminated after i iterations will have executed `S` i times, and is thus equivalent to the unrolled sequence of statements:

$$\mathtt{S}_1 \ ; \ \mathtt{S}_2 \ ; \ \ldots \ ; \ \mathtt{S}_i$$

We can apply reversal to this sequence of statements using rule three - of course here, all statements are identical, so we don't technically need to reverse the order of the sequence, but we *do* recursively reverse each statement in the sequence.

Why swap the loop bounds? Consider a loop in some program p given by `for i = a to b { S }` which executes n times (where n is finite such that the loop terminates). If we unroll the executed statements of the loop into the sequence $\mathtt{S}_1 \ ; \ \ldots \ ; \ \mathtt{S}_n$, then `i` will have had value `a` before executing $\mathtt{S}_1$ and value `b` after executing $\mathtt{S}_n$.

In p', the loop is reversed to a sequence of statements $R(\mathtt{S}_n) \ ; \ \ldots \ ; \ R(\mathtt{S}_1)$. Here, the loop variable `i` will have value `b` before executing $R(\mathtt{S}_n)$, and we expect it to have value `a` after executing $R(\mathtt{S}_1)$ - hence we swap the loop bounds.

---

[1] A program does not terminate if it executes indefinitely!

- *Explain for the loop construct why (or why not) the restriction described in the assignment text is required for reversibility.*

Yes, the restriction *is* required for reversibility. To show why, let us *lift the restriction on loops* - but first, let's introduce some assumptions to ease understanding. Let:

- $p' = R(p) =$ `for i = b to a` $\{R(\texttt{S})\}$;
- $\hat{\texttt{x}}$ denote the value of some global variable `x` at the beginning of p and before p' terminates;
- $\bar{\texttt{x}}$ denote the value of global variable `x` before p terminates and at the beginning of p';
- `S'` $= R($`S`$)$, for brevity;
- $\texttt{S}_j$ and $\texttt{S'}_j$ be the j'th execution of `S` in p and `S'` in p', respectively - note that $\texttt{S}_j = R(\texttt{S'}_j)$;
- and $\texttt{v}_j$ be the value of some program variable `v` (global or loop variable) after executing $\texttt{S}_j$ in p; similarly, let $\texttt{v'}_j$ be the value of `v` after executing $\texttt{S'}_j$ in p'.

Using this, I reformulate the property given in the assignment text:

(1) if for any SIS program p which starts with $\texttt{x} = \hat{\texttt{x}}$ and ends with $\texttt{x} = \bar{\texttt{x}}$ for all global variables `x`, there exists a SIS program p' which starts with $\texttt{x} = \bar{\texttt{x}}$ and ends with $\texttt{x} = \hat{\texttt{x}}$ for all global variables `x`, then SIS is a reversible language.

Let now p be the program `for i = a to b { S }`, and assume that p terminates after n iterations of the loop. Since reversibility must hold for every subsequence of the program, I derive the following invariant for p:

(2) $\texttt{v}_j = \texttt{v'}_{j+1}$ for all $j < n$.

Where `v` is any SIS variable (global or loop variable). Notice that the case where $j = n$ is covered by property (1), since when $j = n$ the loop will have exited and only global variables are live.

Assume now that at some point in p we have $\texttt{i}_k = \texttt{a}_k$ after executing a k'th iteration of the loop (with $1 \leq k < n$). Since we have lifted the

restriction on loops, an error is *not* thrown; the loop continues on executing for the remaining $n-k$ iterations, at which point it terminates with $\texttt{i}_n = \hat{\texttt{b}}$, after which p terminates *without error* with $\hat{\texttt{x}}$ for all global variables `x`.

Now, we execute p'. Since $\texttt{i}_k = \texttt{i'}_{k+1}$ and $\texttt{a}_k = \texttt{a'}_{k+1}$ (by the invariant), and because we assumed $\texttt{i}_k = a_k$, we will after iteration $(k+1)$ of p' have $\texttt{i'}_{k+1} = \texttt{a'}_{k+1}$. At this point the loop terminates, meaning p' *without error* with $\texttt{x} = \texttt{x'}_{k+1}$ for all global variables `x`. However, we cannot assume $\texttt{x'}_{k+1} = \hat{\texttt{x}}$ is true for any `x` or k, and so invariant (2) is broken.

1.3 A4.1.c

- *Is the problem of whether a SIS program terminates (with or without error) regardless of input a trivial, decidable, or undecidable property?*

Unfortunately, I have not managed to work out a foolproof answer to this task - instead, I share some of my thoughts.

Neither the SIS grammar nor the language specification restricts the number of statements in a SIS program, but I shall restrict my view to those SIS programs which can be expressed finitely. Under this assumption, a SIS program is nonterminating only if it contains an infinite loop.

Because the computational range of SIS is limited to whatever can be expressed in three 32-bit variables, SIS is obviously *not* Turing-complete, so a cheap proof of undecidability by Rice's theorem is unfortunately out of the question ;)

On the contrary, because there is no limit to the number of nested loops in a SIS program, and thus no limit to the number of live loop variables at any given time, the amount of memory required to run a SIS program is unbounded - for this reason, it is not possible to simulate the program, checking for repeated memory states on a realizable machine with bounded memory.

Termination for a single loop with no nested loops

Termination is decidable for a program if it only contains singularly nested loops. This is because if a loop does not contain nested loops, then it can only contain increment and decrement statements, and because we assume SIS programs to be finitely expressed, there can only be a finite number of such statements in any loop body.

A finite sequence of statements can be counted in finite time, and hence the change in value of any variable between the start and end of the loop can be determined in finite time. As an example, for a loop with loop variable `i`, lower bound `a` and upper bound `b`, a loop terminates if there exists an integer $k \geq 0$ such that:

$$\mathtt{a}_0 + k\mathtt{i}_\Delta \equiv \mathtt{b}_0 + k\mathtt{b}_\Delta \pmod{2^{32}} \tag{2}$$

where $\mathtt{x}_0$ is the value of variable `x` *before* the loop and $\mathtt{x}_\Delta$ is the change in value of variable `x` across a single loop iteration. If such a k exists, then

k is also the number of iterations of the loop. Equation (2) can be solved in finite time. Note that the equation does not consider $\mathtt{a}_\Delta$; this is because under the restrictions on loops, we know that the loop will terminate if $\mathtt{i} = \mathtt{a}$ at the end of a loop iteration.

Under the assumption that the program contains only singularly nested lists, we can determine the values of $\mathtt{a}_0$ and $\mathtt{b}_0$ in similar fashion by counting the number of decrements and increments leading up to the beginning of the loop, and given k, we can compute the values of $\mathtt{a}_n$ and $\mathtt{b}_n$, where $\mathtt{x}_n$ is the value of some global variable $\mathtt{x}$ after a particular loop.

However, this method does not generalize to nested loops, since we cannot infer $\mathtt{x}_0$ nor $\mathtt{x}_n$ in a nest of loops; we might abstract these away in variables in a system of equations whilst keeping track of net change in value of any global variable at any point in the program, but these equations become messy and, by my intuition, unsolvable once we consider the fact that a nested loop can have a variable amount of iterations across iterations of its enclosing loop.

By this reasoning - however lackluster - I posit that SIS is undecidable.

---

2 A4.2

2.1 A4.2.a

- *Write the sequence of lines executed in the program given in the assignment text. Also, what is the value of* `n` *printed?*

Note: it is ambiguous whether "CBreak" *should be printed whenever the* `if`*-condition is evaluated* **or** *only when the* `break` *statement is actually reached.* For this reason, I answer for both cases.

First, assuming "CBreak" is printed each time the `if`-condition if evaluated:

Assuming "CBreak"

```
Init
Test
CBreak
IDecr
NIncr
Test
CBreak
IDecr
NIncr
Test
CBreak
Print
```

Second, assuming "CBreak" is only printed when the `break` statement is reached:

```
Init
Test
IDecr
NIncr
Test
IDecr
NIncr
Test
CBreak
Print
```

In any case, the value of `n` printed is 2, since `i` becomes 8 after two iterations and, and on the third iteration the loop is broken since `i % j == 8 % 4 == 0`.

2.2 A4.2.b

- *Eliminate the* `break` *statement in the while-loop without using* `goto` *or* `continue` *statements.*

Since the conditional `break` statement is nested directly inside the `while` loop, we can move the break condition up into the loop condition. This is because looping until `(i > j)` and then breaking the loop if `(i % j == 0)` is equivalent to simply looping until `(i > j && i % j != 0)`.

```
int i = 10, j = 4, n = 0;
while (i > j && i % j != 0) {
  i--;
  n++;
}
printf("%d\n", n);
```

2.3 A4.2.c

- *Describe a general method for eliminating* `break` *statements without introducing* `goto` *statements.*

Consider the below C-like pseudocode containing a while loop with a `break` statement at some arbitrary position in the loop body:

```
// arbitrary code preceding the loop.
while(loop_cond()) { // loop_cond() is any function and may have side effects.

  ... // arbitrary code preceding the break.
    ...
      ...

      // the break statement may be arbitrarily nested.
      break;

      ...
    ...
  ... // arbitrary code following the break.
}
```

We wish to eliminate the `break` statement but preserve semantics.

If we are disallowed from using gotos and break statements, then the only way to manipulate control flow out of a loop is for the loop condition to somehow become zero, causing the loop to end. This can be done like so:

- declare a new variable `foo` and initialize to a non-zero value before entering the loop;
- if the original loop condition is `loop_cond()`, then replace this with `foo && loop_cond()`. To preserve semantics, it is important that we place `foo` as the first operand to the logical `AND`, since `loop_cond()` might have side effects;
- finally, replace all occurrences of "`break`" in the loop body with: "`foo = 0; continue;`"

The resulting code looks like so:

```
// arbitrary code preceding the loop.
int foo = 1;
while(foo &&          // important that we short-circuit on foo
      loop_cond()) {  // in case loop_cond() has side effects!
  ...
    ... // arbitrary code preceding the break.
      ...

      // set foo to zero and jump to loop head.
      foo = 0;
      continue;

      ...
    ... // arbitrary code following the break.
  ...
}
```

In summary, whenever control reaches the point in the program where previously there was a `break` statement, the code will now set `foo = 0`; jump to the head of the loop; evaluate the logical `AND` expression and, because `foo` is zero, short-circuit and exit the loop without evaluating `loop_cond()` that extra time. Thus semantics of the loop is preserved.

3 A4.3

3.1 A4.3.a

- *Describe how the syntax of each expression in the Troll grammar is represented as embedded syntax in PLD LISP. Discuss nontrivial choices.*

3.1.1 The embedded syntax

In this subsubsection, I discuss how the Troll syntax is embedded into the PLD-LISP language. Please note that I do not explain the embedding of each and every single constructor, but rather only discuss *non-trivial* choices made - **where nothing else is stated, the embedded syntax can thus be assumed to resemble actual Troll syntax.**

On parsing in PLD-LISP

With limited pattern matching in PLD-LISP (we can only match the first n elements and the tail of a list, not a recursive pattern eg. with ML-like datatypes), my first thought was that it might at least be possible to implement an LL1-parser for Troll. However, the parsing table very quickly became very large and contrived on paper and so I gave up.

I instead pursue implementing a simple evaluator for fully parenthesized Troll expressions (with the exception that parentheses can be omitted around single-symbol expressions).

Embedding the Troll syntax

A Troll expression is represented as a single, nested list of PLD-LISP symbols and number constants. Troll operators and keywords (`d`, `accumulate`, `+`, `count`, etc.) as well as variables are simply represented with symbols. Aside from one restriction and one deviation to the Troll syntax (which will be discussed momentarily), *the only distinguishing factor is this enclosing PLD-LISP list.*

Restrictions to the embedded syntax wrt. whitespace and parenthesization

As mentioned above, expressions must be parenthesized if they contain more than one symbol - this for example means that the singleton collection containing the number 43 can be represented as both the singleton list `'(43)` and simply the constant "`43`" (and similarly for variables), whereas a Troll

expression such as `d x d3` must be parenthesized as `'(d(x d 3))`[2]. This is because evaluation of dice rolling is *largely* implemented as such:

```
(define eval
  (lambda (('d e))     (roll 1 (eval e))
          ((e1 'd e2)) (roll (eval e1) (eval e2))
          ...
```

(Note that this snippet is pseudocode and not actual implementation [3]*. More on this later.)*

Here, the `('d e)` pattern matches *precisely* one literal "`d`" followed by exactly one single value (a symbol or list or symbols), so we must parenthesize `2 d x` since this consists of three distinct symbols. We *could* rewrite the `d e` pattern to use a head/tail pattern as such:

```
(define eval
  (lambda (('d . es))    (roll 1 (eval es))
          ((e1 'd . es)) (roll (eval e1) (eval es))
          ...
```

Using these patterns enables us to evaluate lists such as `'(4 d d d 3 + 4)` - but this only works because the dice rolling operators are right-associative, and because the head of the head/tail pattern is matched *exactly*; unfortunately, this approach falls short for lists such as `'(3 + 4 d d 5)`, since the pattern `e1` cannot match the three symbols constituting `3 + 4`.

Different problems arise with other operators with multiple parameters, in particular the `accumulate` expression, which interleaves multiple keywords and expressions. Thus, to ease implementation, I give the restriction that any Troll operator which takes n parameters will in the embedded syntax take exactly n symbols or lists of symbols.

[2] Also, please note the explicit whitespace between the inner `d` and its second parameter `3`, which is required since otherwise the two would be recognized as the single symbol "`d3`", much like how Troll itself requires whitespace between `d` and `x`.

[3] The actual evaluator is also parameterized with the current environment, and will for example assert that `eval e` is a singleton collection containing one positive number.

All in all this makes for a great deal of redundant parenthesization in the embedded syntax. For a more contrived example, consider the Troll expression:

```
sum least 20 accumulate x := d42 d d d1337 while 888 < sum largest 19 x
```

In the embedded syntax, this is represented by the nested PLD-LISP list:

```
'(sum (least 20 (accumulate x := ((d 42) d (d (d 1337)))
                  while (800 < (sum (largest 19 x))))))
```

On the other hand one might argue that in some cases parenthesization benefits code legibility - because even if association and precedence is well-defiend for all Troll operators, it can sometimes be hard for a human programmer to decipher from a glance of the code, eg. as with the example given here.

3.1.2 Deviation in the embedded syntax wrt. union expressions

Aside from full parenthesization and some forced whitespace, my embedded syntax deviates from that of the given syntax in one other place: The union syntax, $\{\texttt{e}_1, \ldots, \texttt{e}_n\}$, which, in Troll, is given by zero or more comma separated Troll expressions enclosed in curly braces. In the embedded syntax, a union is simply a PLD-LISP list containing zero or more comma separated expressions. As an example, the Troll expression `{3 + 4, 4 d 19, choose {42, 43}}` is represented in the embedded syntax as:

```
'((3 + 4), (4 d 19), (choose (42 , 43)))
```

Notice the explicit whitespace surrounding the last comma; as per usual, whitespace is not required around commas in the presence of parentheses, but *is* required in all other cases since commas are merely PLD-LISP symbols, and other symbols are free to "absorb" those commas if there is no separating whitespace.

Question: *Why not use curly braces as in the source syntax?*

Answer: In order to use curly braces to denote unions in the embedded syntax, these would have to be represented as separate PLD-LISP symbols. This, I felt, overly complicated the embedded syntax, especially considering that to properly evaluate a union expression in the absence of a

parser, the curly braces would have to be treated as any other symbol in terms of whitespace and parenthesization. For example, the Troll expression `least 2 {1, 2, {3, 4}}` would necessarily be represented by the nested list:

```
'(least 2 ({ 1 , 2 , ({ 3 , 4 }) }))
```

with whitespace even between the opening brace and first expression, as well as between the last expression and closing brace (again, unless these are parenthesized), whereas by using PLD-LISP lists to implement the syntax, this expression is simply:

```
'(least 2 (1 , 2 , (3 , 4)))
```

To evaluate an empty union, simply give an empty PLD-LISP list ;)

3.2 A4.3.b

- *Implement the subset of Troll as presented in the assignment text. Discuss non-trivial choices.*

3.2.1 Implementation overview

I have (to the best of my knowledge) successfully implemented evaluation of every expression in the Troll subset as embedded syntax in the PLD-LISP interpreter. As explaiend ealier, I have not implemented any sort of parser, so the former only holds under the assumption of full parenthesization.

Aside from handling each constructor given in the grammar, my implementation handles:

- specifying a number of isolated rolls, which are then returned as a collection of collections;
- correct variable binding, including dynamic scoping and variable overshadowing;
- flattening of arbitrarily nested collections of collections (with the exception of specifying multiple rolls, where a single collection of flat collections is returned).

I claim neither efficiency nor exhaustive correctness (I will, however, make an effort of testing later), but my implementation *does* handle the following cases of run-time errors:

- non-singleton collection used where singleton collection expected
- (singleton collection containing a) non-positive value used where (singleton collection containing a) positive value expected
- unknown variable names

The implementation consists of one large evaluator function, `eval`, and a number of helper functions. In the following subsubsection, I go into detail with most of the features mentioned above, as well as other interesting features of the code.

The code can be viewed in its entirety in the attached `Troll.le` of the code hand-in directory.

3.2.2 Non-trivial implementation details

Randomness

My handling of randomness is entirely trivial, but I see it fit to mention that for the sampling and accumulation operators (which both evaluate some of their operand expressions multiple times, with different, random results each time), randomness is achieved by passing expressions *unevaluated*, such that they can be re-evaluated with new random results.

Variable environment

To implement dynamic scoping, I let my main evaluator function take an additional parameter `env` holding the local variable environment. Unfortunately, this parameter must be passed explicitly, which has lead to cluttered code and, during implementation, a great deal of errors.

Troll variable binding and lookup expressions are handled as such:

```
1  (define eval
2    (lambda ...
3
4      ; variable binding. v must be a symbol; e1 and e2 are arbitrary exps.
5      ((v ':= e1 '; e2) env) (if (symbol? v)
6                                 (eval e2 (bind v (eval e1 env) env)))
7      ...
8      ; scalar expression. symbols are assumed to be variables, while
9      ; singleton collections are created from scalar numeric values.
10     (x env)  (if (symbol? x) (lookup x env) (if (number? x) (list x)))
11   )
12 )
```

In line 6, the current `env` is extended with a binding of `v` to whichever value `e1` evaluates to, before `e2` is evaluated under this new environment. Line 10 shows how symbols are looked up as variables. Note that this is the bottom-most production of the `eval` function.

The actual variable environment `env` is implemented as a list of two-element PLD-LISP lists whose first element is a symbol and whose second element is a fully evaluated collection. I omit the code for binding and lookup as it is entirely trivial, except to say that variable overshadowing is implemented by always prepending variables to the head of the environment list.

Singleton collections

A number of Troll operators expect one or more singleton collections as operands; for example, the arithmetic operators and the multiple dice roll expects that *both* operands are singleton, while the sample operator expects its first operand to be. Furthermore, we also have operators which expect singleton collections containing positive values - examples of this are dice rolls and `least/largest`.

This is implemented using two wrappers for the main `eval` function, called `evalST` and `evalSTPos` (for "evaluate singleton [positive]"), which both evaluate its first argument expression and return empty lists (to signal failure) when the result is not a singleton collection or a singleton collection containing one positive value, respectively.

Flattening collections

The Troll manual specifies that nested collections of collections are always flattened. This is relevant for the sampling, union, and accumulate expressions, which produce multiple, isolated collections before flattening into a single collection.

Since collections are implemented as PLD-LISP lists, I write a generic function for the flattening of arbitrarily nested and arbitrarily irregular PLD-LISP lists (the function is called `flatten` in the code hand-in), which can then be used to wrap the result of a sampling, accumulation, or union expression.

Union expressions

An example of collection flattening is in the union expression $\{\mathtt{e}_1, \ldots, \mathtt{e}_n\}$.

As explianed earlier, I use simply PLD-LISP lists to represent Troll unions. To evaluate arbitrarily long union expressions, I give `eval` a head/tail pattern as seen below:

```
1  (define eval
2    (lambda
3      ...
4      ; {e1, ..., en}: this production assumes n > 0; to create an empty collection,
5      ; simply give an empty list. for example, '(count ()) evaluates to (0).
6      ((e ', . es) env)  (flatten (cons (eval e env) (eval es env)))
7      ...
```

Notice how the same `env` is used for each expression, and how the resulting union is created using `flatten`. The code comment explains the case of empty unions.

`least` and `largest` expressions

`least` and `largest` are both implemented using a simple (ie. not in-place) quicksort. To pick the `n` least elements of some collection `xs`, the collection is first fully evaluated and sorted before the first `min(n, count xs)` elements are extracted; for the `n` largest, the list is reversed after sorting.

`accumulate` expression

Below snippet shows how `accumulate` expressions are evaluated (interesting lines highlighted):

```
1  (define eval
2    (lambda
3      ...
4      (('accumulate v ':= e1 'while e2) env)
5        (if (symbol? v) (flatten (accumulate v e1 e2 env)))
6      ...
7    ...
8  ...
9
10 (define accumulate
11   (lambda
12     (v e1 e2 env) (if (define x (eval e1 env))
13                       (cons x (if (eval e2 (bind v x env))
14                                   (accumulate v e1 e2 env))))
15   )
16 )
```

Lines 4-5 show how `accumulate` expressions are matched in the main evaluator function; if `v` is a symbol, expressions `e1` and `e2` are passed *unevaluated* to a helper function for accumulation expressions.

Given a symbol `v`, two *unevaluated* expressions, and the current environment `env`, the `accumulate` helper function works as such:

- evaluate `e1` under `env` and bind result to `x` (PLD-LISP variable);
- evaluate `e2` under the environment `env[v -> x]`;
- accumulate `x`, and, if `e2` evaluates to a non-empty collection; recurse with `e1` and `e2` again unevaluated, and using the original `env` (ie. without the binding of `v -> x`).

Note the `if`-expression beginning on line 12 of the snippet. This is to sequence the binding of `eval e1 env` to the PLD-LISP variable `x` before recursing, such that `e1` is only evaluated once. This works because `define` is always successful (which it is because `x` is not a reserved name in PLD-LISP).

3.3 A4.3.c

- *Write a short guide to usage of your implementation. Show example rolls by generating at least 10 rolls of each example Troll expression given in the assignment text.*

3.3.1 User guide

To boot up my embedded Troll interpreter implementation, use `make run` from within my code hand-in directory; alternatively, load the module manually by evaluating `(load Troll)` from the PLD-LISP interpreter.

The API of my evaluator is simply the function `troll`. To evaluate some Troll expression `e` (where `e` is given in the embedded syntax), use `(troll e)`. To roll `e` multiple times - say, `n` times, each with different random results; use instead `(troll n e)`.

3.3.2 Example rolls

In below snippet, I evaluate the example rolls given in assignment text, followed by 7 rolls of the the example Troll expression discussed earlier:

```
> ; 10 rolls of `d12+d8`
  (troll 10 '((d 12) + (d 8)))
> = ((10) (13) (13) (8) (14) (12) (2) (9) (17) (7))

> ; 10 rolls of `min 2d20`
  (troll '(min (2 d 20)))
> = ((5) (3) (7) (13) (9) (12) (3) (4) (13) (8))

> ; 10 rolls of `max 2d20`
  (troll 10 (max (2 d 20)))
> = ((11) (20) (17) (8) (10) (20) (17) (20) (19) (18))

> ; 10 rolls of `sum largest 3 4d6`
  (troll 10 '(sum (largest 3 (4 d 6))))
> = ((12) (14) (14) (9) (4) (15) (11) (10) (13) (8))

> ; 10 rolls of `count 7 < 10d10`
  (troll 10 '(count (7 < (10 d 10))))
> = ((3) (5) (3) (3) (1) (3) (3) (3) (1) (4))

> ; 10 rolls of: `choose {1, 3, 5}`
  (troll 10 '(choose (1 , 3 , 5)))
> = ((5) (5) (3) (5) (3) (5) (3) (1) (3) (1))

> ; 10 rolls of `accumulate x := d6 while x > 2`
  (troll 10 '(accumulate x := (d 6) while (x > 2)))
> = ((3 4 4 1) (2) (6 4 5 3 2) (4 1) (6 4 4 1)
```

```
      (5 6 5 5 4 4 5 3 6 6 3 3 3 6 3 1) (3 6 4 1) (4 2)
      (3 5 2) (4 3 3 4 6 4 5 6 2))

> ; 10 rolls of `x := 3d6 ; min x + max x`
  (troll 10 '(x := (3 d 6) ; ((min x) + (max x))))
> = ((3) (11) (9) (3) (6) (9) (8) (6) (7) (5))

> ; 7 rolls of the example discussed in the report wrt. parenthesization
  (troll 7 '(sum (least (19 + 1) (accumulate x := ((d 42) d (d (d 1337)))
  while (800 < (sum (largest 19 x)))))))
= ((782) (1250) (175) (1212) (418) (170) (277))
```

To reproduce these example rolls, use `make example` from within the code hand-in. The output will not include the same annotations as above snippet.

3.3.3 Testing

In addition to the example rolls, I want to test my program. Ideally, I would write automated and reproducible unit tests, but this is only feasible for Troll expressions with no randomness involved; for dice rolling (ie. `d` e and e_1 `d` e_2) and the `choose` operator, manual testing will have to suffice [4]. I also perform manual testing of the `accumulate` expression, since if there is no randomness involved then we can only have one accumulation *or* infinite looping.

Manual testing of random operators

Below are the test cases used in my manual testing of random operators. I hope that the prompt print-out speaks for itself.

```
> (troll '(d 400))     ; single die roll.
= (276)
> (troll '(d 1))       ; single one-sided die roll.
= (1)
> (troll '(d -12))     ; invalid single die roll.
= ()

> (troll '(23 d 400)) ; multi dice roll.
= (197 341 318 240 79 387 121 318 121 113 287 86 131 211 12 216 116 53 166 120
   202 11 112)
> (troll '(1 d 19))    ; multi dice roll, but just one die.
= (8)
> (troll '(0 d 19))    ; no dice!
= ()

> (troll '(choose (1 , 2 , 3 , 4 , 5))) ; simple choose expression.
= (2)
> (troll '(choose (1843 d 5000000)))    ; choose among many.
= (4533089)
> (troll '(choose (d 1)))               ; choose from singleton collection.
= (1)
> (troll '(choose (50 < (4 d 19))))     ; choose from empty collection.
= ()

> (troll '(accumulate x := (65 < (10 d 100))  ; simple accumulate expression
                      while (1 < (count x)))) ; which runs for multiple iterations.
= (91 76 81 98 82 66 100 75 94 78 99 96 75 91 100 77 87 90 83 80 97 79 87 87 86
   66 77 97 95 83 76 80 82 96 74 88 93 83 97 100 90 81 66 76 99 98 95 66 73 82
   97 97 74 90 93 76 88)
```

The results seem reasonable - all values are within expected ranges, and

[4] In a different setting or scope we might have had eg. quickcheck testing.

the sizes of all produced collections are as expected. To reproduce, use `make manual_tests` from within the code hand-in.

Unit testing

I write small and simple unit tests for the rest of the constructors in the subset of Troll. I attempt to also test various features of the Troll language, such as correct flattening of collections in collections, correct variable scope, and variable overshadowing, as well as a small number of edge case use cases, such as empty collections, illegal dice rolls, and unknown variables.

I do *not* write any tests of syntactically invalid programs.

All my test cases run successfully. Test cases can be viewed in the `testing.le` file of my code hand-in, and test results can be reproduced with `make unit_tests` from inside the code hand-in directory.

To reproduce all tests, use instead `make test`.

3.4 A4.3.d

- *If there is anything missing, discuss how this might be resolved.*

I am fairly certain that my implementation correctly handles all expressions in the subset of Troll given in the assignment text. However, I for one cannot guarantee that my implementation is bugfree.

More importantly, my implementation only works under the assumption of full parenthesization of the embedded syntax. This is quite a huge assumption! To remedy this, I would look into implementing an actual parser for the language. In this regard, I again argue that PLD-LISP's head/tail pattern matching lends itself best to a LL1-parser with stack-based AST generation. Given more time, I would have liked to have given a better attempt at this - in hindsight, I would have done better to have used an automatic parser table generator rather than attempting (and quickly failing) at doing this by hand.

---

4 A4.4

4.1 A4.4.a

- *Define a subtype ordering $\sqsubseteq$ on interval types and prove that it is a partial order.*

Let `A` and `B` be interval types given by:

$$\begin{aligned}&\text{type } \texttt{A} = a\,..\,b\\&\text{type } \texttt{B} = c\,..\,d.\end{aligned}$$

Then the subtype ordering $\sqsubseteq$ on interval types is given by:

$$\texttt{A} \sqsubseteq \texttt{B} \iff a \geq c \wedge b \leq d\,. \tag{3}$$

We put no restriction on the relationship between `a` and `b` or between `c` and `d`. *Why*? Because if `a` $>$ `b` then `A` is an empty interval type, and then `A` is trivially a subtype of any other interval type, and if `c` $>$ `d`, then we have:

$$\begin{aligned}c > d \ \wedge\ \texttt{A} \sqsubseteq \texttt{B} \ &\implies\ a > b\\&\implies\ \texttt{A} \text{ and } \texttt{B} \text{ are both empty interval types.}\end{aligned}$$

4.1.1 Showing $\sqsubseteq$ is a partial order

For the interval type ordering to be a partial order, it must be a reflexive, antisymmetric, and transitive relation.

Reflexivity

Let's start with reflexivity. Our subtype ordering is reflexive if an interval type `A` is a subtype of itself, ie. that:

$$\texttt{A} \sqsubseteq \texttt{A} \iff a \geq a \ \wedge\ b \leq b \tag{4}$$

The RHS of expression (4) is true by reflexivity of the "less/greater than or equal to" relations on numbers. Our subtype ordering is reflexive - so far, so good.

Antisymmetry

Next, we want to show that $\sqsubseteq$ is antisymmetric. This is the case if:

$$\mathtt{A} \sqsubseteq \mathtt{B} \ \wedge \ \mathtt{B} \sqsubseteq \mathtt{A} \quad \Longrightarrow \quad \mathtt{A} = \mathtt{B}$$

In other words, our subtype ordering is antisymmetric if two interval types can only be subtypes of each other if they are equal. I intend to show antisymmetry by contradiction.

Assume first that $\mathtt{A} \sqsubseteq \mathtt{B}$ and $\mathtt{B} \sqsubseteq \mathtt{A}$. Then we have:

$$\mathtt{A} \sqsubseteq \mathtt{B} \ \wedge \ \mathtt{B} \sqsubseteq \mathtt{A} \quad \Longleftrightarrow \quad \underbrace{(a \geq c \ \wedge \ b \leq d)}_{\text{by } \mathtt{A} \sqsubseteq \mathtt{B}} \wedge \underbrace{(a \leq c \ \wedge \ b \geq d)}_{\text{by } \mathtt{B} \sqsubseteq \mathtt{A}}$$

From the RHS of above expression, we derive the following:

$$a \leq c \ \wedge \ c \leq a \iff a = c,$$
$$b \leq d \ \wedge \ d \leq b \iff b = d$$

Assume now, for contradiction, that $\mathtt{A}$ and $\mathtt{B}$ are *not* equal:

$$\mathtt{A} \neq \mathtt{B} \quad \Longleftrightarrow \quad a \neq c \ \vee \ b \neq d \tag{5}$$

However, this immediately contradicts our previous assumption, from which we derived $a = c$ and $b = d$, since $\mathtt{A} \neq \mathtt{B}$ requires at least one of these to be false; hence two types cannot be subtypes of each other if they are also not equal, and our subtype ordering is shown to be antisymmetric.

Transitivity

Assume $\mathtt{A} \sqsubseteq \mathtt{B}$ and $\mathtt{B} \sqsubseteq \mathtt{C}$ for some third interval type $\mathtt{C}$ given by:

$$\text{type } \mathtt{C} = e \mathinner{..} f.$$

To show transitivity, we must show that this implies $\mathtt{A} \sqsubseteq \mathtt{C}$. Starting from our assumptions:

$$\mathtt{A} \sqsubseteq \mathtt{B} \quad \Longleftrightarrow \quad a \geq c \ \wedge \ b \leq d,$$
$$\mathtt{B} \sqsubseteq \mathtt{C} \quad \Longleftrightarrow \quad c \geq e \ \wedge \ d \leq f$$

If we reorder the right-hand sides of above expressions and use transitivity of the "less/greater than or equal to" relations on numbers, we have:

$$\begin{aligned} a \geq c \geq e \quad &\implies \quad a \geq e, \text{ and} \\ b \leq d \leq f \quad &\implies \quad b \leq f. \end{aligned}$$

As such `A` is a subtype of `C`. In addition to reflexivity and antisymmetry, our subtype ordering has now also been shown to be transitive, and with that it is also shown to be a partial order.

4.2 A4.4.b

- *Should the* `Association` *constructor be covariant or contravariant in its first type parameter wrt. the subtype ordering* $\sqsubseteq$ *(as it was defined in the previous task)? And what about its second parameter?*

For brevity, let `A` and `B` be arbitrary instances of the association type given by:

$$\texttt{A} = \texttt{Association(T1, T2)},$$

$$\texttt{B} = \texttt{Association(T3, T4)}.$$

Where `T1`, `T2`, `T3`, and `T4` are interval types.

A value of type `A` is a list of tuples which map values of type `T1` to values of type `T2`, and because all first elements of the list of pairs are required to be distinct, we might even be so bold as to call it a *mapping* from `T1` to `T2` ;)

Since a mapping is essentially a function, it is reasonable to assume that the `Association` type constructor should be *contravariant in its first parameter* and *covariant in its second parameter*, and then we would have:

$$\texttt{A} \sqsubseteq \texttt{B} \quad \iff \quad \texttt{T1} \sqsupseteq \texttt{T3} \ \wedge \ \texttt{T2} \sqsubseteq \texttt{T4}, \tag{6}$$

but we would do well to justify this assumption.

"Proof"

I will not pursue a formal proof as to why it must be so, but rather show the intuition.

Assume $\texttt{A} \sqsubseteq \texttt{B}$. For our association subtype ordering as defined in (6) to be correct, we must be able to swap in a value of type `A` wherever we would otherwise have a value of type `B`.

Assume, for contradiction, that our association type constructor is instead *covariant* in its *first parameter*. Then `T1` is at best *as expressive* in the range of values as `T3`, but in the most cases it would be more restrictive and thus we cannot assume that the value of type `A` is sufficient. Thus our association subtype ordering must be contravariant in the first parameter.

Assume then, once again for contradiction, that our association type constructor is *contravariant* in its *second parameter*. Then `T2` is as expressive

or more expressive in the range of possible values than `T4` is. If this is the case then we have a problem whenever we lookup an association and expect a value in `T4`, but get back a value that is in `T2` but not in `T4`. We must restrict the interval type `T2` to describe a *sub*set of what is described by `T4`, and hence we make our association type constructor *covariant* in its second parameter.

---

5 A4.5

- *Formulate big-step semantic rules for the new* `for` *loop construct.*

5.1 Disambiguating the specifications

The assignment text is ambiguous in a number of ways. First off, it states that "*[...]the loop body s is executed, if* $n_2 > n_1$". However, I argue that it is more correct to say that the loop body is executed if $\texttt{x} < \texttt{n}_2$. [5]

Second, it is not entirely clear whether $\texttt{n}_1$ and $\texttt{n}_2$ are constants or variables (ie. modifiable in `s`), or whether `x` is modifiable in `s` (not counting the increment immediately following `s`).

I resolve the ambiguities by making the following assumptions:

1. the loop condition is $\texttt{x} < \texttt{n}_2$, not $\texttt{n}_1 < \texttt{n}_2$;
2. $\texttt{n}_1$ and $\texttt{n}_2$ are constants and are thus never modified in `s`;
3. `x` is a variable as any other, and can thus be modified in `s` (in addition to the increment immediately following `s`).

I also make the following assumption on the loop construct, since nothing was stated explicitly in the text:

4. the assignment $\texttt{x := n}_1$ is made *exactly* once - right before the first evaluation of the loop condition, and the state σ is modified to reflect this assignment even if `s` is executed zero times;

5.2 Desugaring the `for` loop syntax

From the lectures on loops and control flow, we know that `for` loops are (typically) syntactic sugar for equivalent `while` loops. Using assumptions 4 and 1 as given above, a `for` loop construct can be desugared into a semantically equivalent `while` loop as such:

$$\texttt{for x := n}_1\texttt{ to n}_2\texttt{ do s} \quad \equiv \quad \texttt{x := n}_1\texttt{; while (x < n}_2\texttt{) \{s; x++;\}},$$

where $\equiv$ is semantic equivalence of statements. *But why desugar in the first place?* Answer: In devising the new semantic rule(s), I suspect that I

[5] Hans agrees on this, as per his reply to this post on the course's Absalon forum: `https://absalon.ku.dk/courses/47084/discussion_topics/351077`

will need to be able to distinguish between the first iteration of a `for` loop, which involves initialization of `x`, and subsequent iterations, which do not. I suspect this is better expressed by desugaring.

5.3 New semantic rules

I formulate the following three semantic rules for the new `for` loop construct:

$$\frac{\sigma[\mathtt{x} \mapsto \mathtt{n}_1] \vdash \texttt{while (x < n}_2\texttt{) do \{ s; x++; \}} \rightsquigarrow \sigma'}{\sigma \vdash \texttt{for x := to n}_1\texttt{ to n}_2\texttt{ do s} \rightsquigarrow \sigma'} \tag{7}$$

$$\frac{\sigma(\mathtt{x}) < \mathtt{n} \qquad \sigma \vdash \mathtt{s} \rightsquigarrow \sigma' \qquad \sigma' \vdash \mathtt{x{+}{+}} \rightsquigarrow \sigma'[\mathtt{x} \mapsto \sigma(\mathtt{x}) + 1] = \sigma''}{\sigma \vdash \texttt{while (x < n) do \{ s; x++; \}} \rightsquigarrow \sigma''} \tag{8}$$

$$\frac{\sigma(\mathtt{x}) \geq \mathtt{n}}{\sigma \vdash \texttt{while (x < n) do \{ s; x++; \}} \rightsquigarrow \sigma} \tag{9}$$

Rule (7) is the desugaring rule, states that the change in the store of a `for` loop is equivalent to that of the change in the store of an equivalent `while` loop under σ extended with the initialized loop variable. *One could then define semantic rules for generic* ***`while`*** *loops and use these to derive* ***`for`*** *loop semantics - but for the sake of cementing the new* ***`for`*** *loop, I define rules specifically for the desugared loop.*

Rule (8) describes the case where the loop condition is true, and states that:

- if `x < n` under σ, and
- if under σ, executing the `for` loop body `s` provokes a change in state to some modified state σ', and
- if under σ' following an execution of `s`, the final increment of `x` further modifies the state from σ' to some final state $\sigma'' = \sigma'[\mathtt{x} \mapsto \sigma'(\mathtt{x}) + 1]$,
- then the semantics of the `while` loop is a change in state from σ to σ''.

Finally, rule (9) describes how, if the loop condition is false, then the desugared `while`-loop leaves the state unmodified (ie. `s` is never executed)

Notice that:

- while not immediately obvious from rule (7), the store σ is always modified by executing a `for` loop, since even if the loop body `s` is never executed (ie. if $\mathtt{n}_1 \geq \mathtt{n}_2$), then the semantics is a chagne in state from σ to $\sigma[\mathtt{x} \mapsto \mathtt{n}_1]$.
- rule (8) explicitly separates the loop body `s` from the following increment of `x` as to make the distinction that `x` can also be modified in `s`.

---

6 A3.6

6.1 A3.6.a and A3.6.b

- *Explain by means of the query resolution algorithm why the query* `knows(a, d)` *fails to terminate.*

When `knows(a, d)` is queried, the resolution algorithm will match the first `knows/2` rule: and try to unify `X` with `a`, and `Y` with `d`. This is successful, since neither `X` nor `Y` is bound yet so we now have the list of substitutions $\Theta = \{$`X01 = a`, `Y01 = d`$\}$, where `X01` and `Y01` are temporary variables.

To solve this goal under Θ, the algorithm will then attempt to solve `knows(a, Z)`, which again matches the first `knows/2` rule - the algorithm will unify `Y` with `Z`, which is successful, and the list of substitutions is now $\Theta = \{$`X01 = a`, `Y01 = d`, `Y02 = Z01`$\}$. The algorithm recurses once again on solving `knows(A, Z01)` (where `Z01` is a newly bound temporary variable), and in fact goes into an infinite loop here.

The infinite recursion happens because the resolution algorithm is always free to instantiate a new temporary variable for `Z` and attempt solving the first `knows/2` rule. The resolution tree looks like this:

```
knows(a, d)
   |
   | X = a
   |
knows(a, Z01)
   |
   |
knows(a, Z02)
   |
   |
knows(a, Z03)
   |
  ...
```

- *Modify the program such that* `knows(a, d)` *terminates successfully and explain why the modification works.*

To solve the problem, we need to eliminate left recursion while still encoding the transitive closure of `knows/2`. This is done by replacing the culprit left recursing `knows(X, Z)` call with a call to `friend(X, Z)`, like in the below snippet:

```
friend(a, b).
friend(b, c).
friend(c, d).

knows(X, Y) :- friend(X, Z), knows(Z, Y).
knows(X, Y) :- friend(X, Y).
```

Now, in solving the goal `knows(X, Y)`, the resolution algorithm is forced to satisfy the goal `friend(X, Z)`, and since our program has no recursive rules for `friend/2` (in fact, it only has facts), we can guarantee progression towards solving the supergoal.

Now, the resolution tree for `knows(a, d)` looks like this (the algorithm makes a pre-order traversal):