Homework #3: Interpreter for The Set Operation Language

Out: Tuesday, May 12, 2021, Due: May 23, 2021, 23:55

Administrative

The language for this homework is:

```
#lang pl
```

The homework is twofold. The first part is basically an implementation of a simple language, similar to the WAE language that we have seen in class. The implementation itself is relatively straightforward, and it will be easy to follow the simple steps described below to complete it. The second part, which is separated from the first one, deals with the ability to point out free occurrences of identifiers – already in the syntactic analysis part (parsing).
Important: the grading process requires certain bound names to be present. These names need to be global definitions. The grading process *cannot* see names of locally defined functions.
This homework is for work and submission in pairs (individual submissions are also fine).

In case you choose to work in pairs:

1. **Make sure to specify the ID number of both partners within the file-name for your assignment. E.g., 022222222_44466767_3.rkt.**
2. **Submit the assignment only once.**
3. **Make sure to describe within your comments the role of each partner in each solution.**

Integrity: Please do not cheat. You may consult your friends regarding the solution for the assignment. However, you must do the actual programming and commenting on your own (as pairs, if you so choose)!! 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.

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. **A solution without proper comments may be graded 0.** In general, comments should appear above the definition of each procedure (to keep the code readable).

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. Note that your tests should **not only** cover the code, but also all end-cases and possible pitfalls.

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.

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 code for all the following questions should appear in a single .rkt file named <your IDs>_3 (e.g., 022222222_44466767_3.rkt for a pair of students with ID numbers 022222222 and 44466767, and 333333333_3 for a single-submission of a student whose ID number is 333333333).

PART A:

Implementing the SOL Language

In this part, you are going to implement the Set Operation Language (SOL). We will think of a Set as a sequence of numbers, containing no repetitions. The SOL language will come with three primitive operations: Union, Intersection, and

Scalar Multiplication (where each element of the set is multiplied by the same scalar). The language will also support identifiers – and will have 'with' expressions, similarly to the WAE language that we have seen in class.

You are requested to fill in all missing parts in the following incomplete interpreter (note that each <-- fill in --> may stand for more than a single phrase).

In each of the following parts you are asked to add comments and tests as explained above.

Note that S must be added to the relevant constructors and functions at the end of the name, such as: IdS, parseS, etc.

1. Writing the BNF for the SOL language

The first step would be to complete the BNF grammar for the SOL language in the partial code provided to you as a skeleton (in the above link to the incomplete interpreter). In writing your BNF, you can use <num> and <id> the same way that they were used in the WAE language.

The following are valid expressions:

```
"{1 3  4 1 4  4 2 3 4 1 2 3}"
"{}"
"{union {1 2 3} {4 2 3}}"
"{intersect {union {1 2 3} {7 7 7}} {4 2 3}}"
"{with {S {intersect {1 2 1 3 7 3} {union {1 2 3} {4 2 3}}}}
      {union S S}}"
"{ scalar-mult 3/2  {with {S {intersect {1 2 1 3 7 3} {union {1 2 3} {4 2 3}}}}
                            {union S S}}}"
```

The following are invalid expressions:

```
"{1 3  4 1 4  4 2 3 4 1 2 3} {}"     ;; there should appear a single expression
"{x y z}"                            ;; sets cannot contain identifiers
"{union {1 2 3} {4 2 3} {}}"          ;; union and intersect are binary operations
"{intersect {union {1 2 3} {7 7 7}}}" ;; union and intersect are binary operations
"{with S {intersect {1 2 1 3 7 3} {union {1 2 3} {4 2 3}}}
      {union S S}}"                   ;;    bad with syntax
"{ scalar-mult {3 2} {4 5}}"          ;; first operand should be a nymber
```

2. Parsing

Complete the parsing section within the above code. Having understood the syntax, this should be quite straightforward. See the following tests:

```
(test (parseS "{1 3  4 1 4  4 2 3 4 1 2 3}") => (Set '(1 3 4 1 4 4 2 3 4 1 2 3)))
(test (parseS "{union {1 2 3} {4 2 3}}") => (Union (Set '(1 2 3)) (Set '(4 2 3))))
(test (parseS "{intersect {1 2 3} {4 2 3}}") => (Inter (Set '(1 2 3)) (Set '(4 2 3))))
(test (parseS "{with S {intersect {1 2 3} {4 2 3}}
               {union S S}}")
      =error> "parse-sexprS: bad `with' syntax in")
(test (parseS "{}") => (Set '()))
```

3. Set Operations

Complete the missing code for the set operations appearing in the incomplete code provided to you. Specifically, complete the code for the following procedures, using the tests that follow below (all already appear in the attached file):

```
(: ismember? : Number SET  -> Boolean)
(: remove-duplicates : SET  -> SET)
(: create-sorted-set : SET -> SET)
(: set-union : SET SET -> SET)
(: set-intersection : SET SET -> SET)
(: set-smult : Number (Listof Number) -> SET)

(test (ismember? 1 '(3 4 5)) => #f)
(test (ismember? 1 '()) => #f)
(test (ismember? 1 '(1)) => #t)

(test (remove-duplicates '(3 4 5 1 3 4)) => '(5 1 3 4))
(test (remove-duplicates '(1)) => '(1))
(test (remove-duplicates '()) => '())

(test (create-sorted-set '(3 4 5)) => '(3 4 5))
(test (create-sorted-set '( 3 2 3 5 6)) => '(2 3 5 6))
(test (create-sorted-set '()) => '())

(test (set-union '(3 4 5) '(3 4 5)) => '(3 4 5))
(test (set-union '(3 4 5) '()) => '(3 4 5))
```

(test (set-union '(3 4 5) '(1)) => '(1 3 4 5))

(test (set-intersection '(3 4 5) '(3 4 5)) => '(3 4 5))
(test (set-intersection '(3 4 5) '(3)) => '(3))
(test (set-intersection '(3 4 5) '(1)) => '())

(test (set-smult 3 '(3 4 5)) => '(9 12 15))
(test (set-smult 2 '()) => '())

4. Substitutions

Using the following formal specifications (you can also use our WAE interpreter as reference), write the subst procedure

```
(: substS : SOL Symbol SOL -> SOL)
  ;; substitutes the second argument with the third argument in the
  ;; first argument, as per the rules of substitution; the resulting
  ;; expression contains no free instances of the second argument
  ….
```

```
  #| Formal specs for `subst':

     Formal specs for `subst':

   (`set' is a <NumList>, E, E1, E2 are <SOL>s, `x' is some
    <id>, `y' is a *different* <id>)

      set[v/x]                = set
      {smult n E}[v/x]        = {smult n E[v/x]}
      {inter E1 E2}[v/x]      = {inter E1[v/x] E2[v/x]}
      {union E1 E2}[v/x]      = {union E1[v/x] E2[v/x]}
      y[v/x]                  = y
      x[v/x]                  = v
      {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}
      {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
|#
```

5. Evaluation

Using the following formal specifications (you can also use our WAE interpreter as reference), and using the set operations you implemented above – write the eval procedure:

(: eval : SOL -> SET)
;; evaluates SOL expressions by reducing them to set values

```
#| Formal specs for `eval':
    Evaluation rules:

  eval({ N1 N2 ... Nl }) = sort( create-set({ N1 N2 ... Nl }))
        ;; where create-set removes all duplications from the
           sequence (list) and sort is a sorting procedure.
  eval({scalar-mult K E}) =  { K*N1 K*N2 ... K*Nl }
        ;; where eval(E)={ N1 N2 ... Nl }
  eval({intersect E1 E2})= sort (create-set (set-intersection
                                     (eval(E1), eval(E2))))
  eval({union E1 E2}) = sort (create-set (set-union
                                     (eval(E1), eval(E2))))
  eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
  eval(id)               = error!
```

---

6. Interface

The run procedure wrapping it all up is provided to you. Use it to write tests for your code.

PART B:

Detecting free instances in a WAE program

A code that contains free instances of an identifier is a bad code, and should not be evaluated into anything but an error message. In this section, we go back to the realm of the WAE language to consider the task of identifying free instances in a program.

Identifying free instances before evaluation

To check whether or not there are free instances of an identifier (any identifier) – it is enough to perform a syntactic analysis (that is, no need to evaluate the code).
In our WAE interpreter we indeed issue an error message in such a case, however, we do so during the evaluation process. Write a function
(: freeInstanceList : WAE -> (Listof Symbol))

that consumes an abstract syntax tree (WAE) and returns null if there are no free instance, and a list of all the free instances otherwise – each appearing exactly once in the list (i.e., multiple free occurrences of the same identifier that are free should all be represented by a single occurrence of it in the list). The order in which symbols appear within the list is not important.
Here are some tests that should assist you:

```
(test (freeInstanceList (parse "w")) => '(w))

(test (freeInstanceList (parse "{with {xxx 2} {with {yyy 3} {+
{- xx y} z}}}")) => '(xx y z))

(test (freeInstanceList (With 'x (Num 2) (Add (Id 'x) (Num
3)))) => '())

(test (freeInstanceList (parse "{+ z {+ x z}}")) => '(z x))
```

HINT: use the current structure of eval (and subst) in the WAE interpreter we have seen in class to do so. Your resulting program should never evaluate the code (WAE) it takes as input. In fact, the function eval itself should not appear in your code.