Homework #4: Interpreter for The Set Operation Language – with Environments (static and dynamic)

Out: Tuesday, June 06, 2021, Due: June 18, 2021, 12:55

Administrative

The language for this homework is:

```
#lang pl
```

The homework is a continuation of the previous one, where you will write an implementation of the simple SOL language in the environment model (extended to also allow dynamic scoping).
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_4.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>_4 (e.g., 022222222_44466767_4.rkt for a pair of students with ID numbers 022222222 and 44466767, and 333333333_4 for a single-submission of a student whose ID number is 333333333).

Introduction

In this work, you are going to augment the Set Operation Language (SOL) that you've seen in the previous HW in three ways:

1. Allowing Boolean expressions and conditionals.
2. Turning `with' expressions into a syntactic sugar for call+fun expressions.
3. Implementing the evaluation using environments (allowing both static and dynamic calls to functions).

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, and even include more non-equal numbers of left/right parenthesys).

PART A:

Augmenting the syntax for the SOL Language with Boolean expressions and call-static/call-dynamic

In this part, you are going to augment the Set Operation Language (SOL) that you've seen in the previous HW to allow Boolean expressions and two distinct call expressions. In this part you will complete adapting the parser.

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.

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.

Note that both 'fun' and 'with' expressions will have exactly two papmeters!!

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 3} {4 2 3}} c {}}
            {call-dynamic {fun {x y} {union x S}}
                          {if {equal? S {scalar-mult 3 S}}
                              then S
                              else {4 5 7 6 9 8 8 8}}
                          {1 2 66}}}"
"{with {S {intersect {1 2 3} {4 2 3}} c {}}
            {call-dynamic {fun {x y} {union x S}}
                          {scalar-mult 3 S}
                          {4 5 7 6 9 8 8 8}}}"
"{with {S {intersect {1 2 3} {4 2 3}}
              S1 {}}
           {call-static {fun {x y} {union x y}}
                          {scalar-mult 3 S}
                          {4 5 7 6 9 8 8 8}}}"
"{if {equal? {1 2 3} {1 2}} then {1 2 3} else {1 2}}"
"False"
```

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 number
"false"
"#f"
"{if {1 2 3} {45 67} {}}"
"{call-static f y}"
```

2. Parsing

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

Note that with expressions should be syntactic sugar for calling a function – i.e., you should not define a special constructor for 'with' expressions!!

PART B:

Adapting the implementation of eval for the SOL Language with Boolean expressions and call-static/call-dynamic

In this part, you are going to augment the Set Operation Language (SOL) that you have seen in the previous HW to allow Boolean expressions and two distinct call expressions. In this part you will complete adapting the parser.

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).

3. Formal evaluation rules

Complete the missing evaluation rules. Specifically, complete the following incomplete rules (all already appear in the attached file). Provide explanations on all your choices.

```
eval({call-static E-op E1 E2}, env)
                    = eval(Ef,extend(x2,eval(E2, env) ... <-- fill in --> )
                                if eval(E-op, env) = <{fun {x1 x2} Ef}, envf>
                    = error!          otherwise
eval({call-dynamic E-op E1 E2},env)
                    = <-- fill in -->
                                if <-- fill in -->
                    = error!          otherwise

eval(False, env)      = <-- fill in -->
eval({if E1 then E2 else E3}, env)
                    = eval(E3, env)      if eval(E1,env) = false
```

= <-- fill in --> otherwise

Remarks: a. if statement should work similarly to Racket (i.e., only one expression other the condition-expression should be evaluated).
b. A static call to a function should evaluate a function in the (extended) environment in which it is defined. A dynamic call to a function should evaluate a function in the (extended) environment in which it is called.

4. Before eval

Like we did in class, the returned type of eval is going to be **VAL**. To support the eval function (which you will do in the next section), you need to complete the following missing parts:

The definition of the **VAL** type.

a. The definition of **smult-set** and of **set-op** (use the procedure **SetV->set**, provided to you).

Explain your choices!

5. Evaluation

Using the formal specifications that you previously completed, and the tests provided to you within the incomplete code – write the eval procedure.

6. Creating a non-empty global environment

In class, we used the empty environment as the global environment. Here, you are requested to create a non-empty global environment. Specifically, this global environment should contain the procedures: cons (creating a pair), first (returning the first element in a pair), and second (returning the second element in a pair). For example. the following test should work:

```
(test (run "{with {p {call-static cons {1 2 3} {4 2 3}}
                  S1 {}}
            {with {S {intersect {call-static first p {}}
                                {call-static second p {}}}
                  S1 {}}
              {call-static {fun {x y} {union x S}}
                           {scalar-mult 3 S}
                           {4 5 7 6 9 8 8 8}}}}")
      =>  '(2 3 6 9))
```

Guidance:

a. The idea is to implement a pair as a procedure that remembers the values with which it was initialized. This procedure is the returned value of the **cons** procedure. Specifically, when the procedure **first** and **second** are applied with the created pair (which is a procedure), they apply this pair on a selector function. Thus. Someone, who knows the interior of our implementation, can run the following test:

```
(test (run "{with {p {call-static cons {1 2 3} {4 2 3}} c{}}
             {with {foo {fun {x y} {intersect x y}} c {}}
               {call-static p foo {}}}}")
      =>  '(2 3))
```

b. To start implementing you could do as follows. Complete the following three rows (which should not appear as part of the final interpreter's code, but should be executed if you complete all other parts of the missing code).

```
(run "{with {cons {fun {f s} < -- fill in -- >}}
        cons}")
(run "{with {first {fun {p spare-param}
                     {call-< -- fill in -- >
                                          }}
       first}")
(run "{with {second < -- fill in -- >
              {< -- fill in -- >
                        {fun {a b} b}
                        {}}}
        spare-param {}}
       second}")
```

Note that, since we fixed the number of parameters for function/with expressions, you would sometimes need to use an extra (useless) parameter, e.g., {} (this happens if you only need a single parameter, rather than two).

Now use the returned values you got to complete the following code:

```
(: createGlobalEnv : -> ENV)
 (define (createGlobalEnv)
   (Extend 'second <-- fill in -->
         (Extend <-- fill in -->
               (Extend <-- fill in -->
                     (EmptyEnv)))))
```

7. Interface

The run procedure wrapping it all up is provided to you. Use it to write tests for your code. Note that we allow three different types as the returned value of **run**.

At first you should use the empty environment as the global one, but after you complete section 6, you should use the non-empty one.

PART C:

Open questions

1. What are the types that we have now (after you are done) in the SOL language?
2. Explain where in the solution of section 2 (when parsing **with** expressions) you called a function dynamically/statically – what was the importance of your choices?
3. Explain where in the solution of section 6 you used call-dynamic and where you used call-static – what was the importance of your choices?
4. Would there be any difference if we used call-dynamic in some places in the following test? Explain your answer.

```
(test (run "{with {p {call-static cons {1 2 3} {4 2 3}}
                  S1 {}}
             {with {S {intersect {call-static first p {}}
                                 {call-static second p {}}}
                   S1 {}}
        {call-static {fun {x y} {union x S}}
                     {scalar-mult 3 S}
                     {4 5 7 6 9 8 8 8}}}}")
  =>  '(2 3 6 9))
```