Homework #3: Basic Interpreter Extension & Identifying Free Instances

Out: Saturday, May 17, 2020, Due: May 31, 2020, 23:55

Administrative

The language for this homework is:

```
#lang pl 03
```

The homework is twofold. The first part is basically a simple extension of the WAE language that we have seen in class. The extension itself is relatively straightforward, and it will be easy to follow the simple steps described below complete it. In this sense, it is also a kind of an introduction, although not as basic as the previous homeworks. 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).

If you choose to consult any other person or source of information, this **must be** clearly declared in your comments.

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 much of the focus of this homework is put on the contract and purpose statements, good style (indentation, comments, etc), and good tests. Note that your tests should not only cover the code, but also all end-cases and possible pitfalls.

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:

Introduction

After completing the WAE implementation (which we did together in class), you are now required to implement a "proper" implementation of `sqrt`: one that returns ***two*** results (one positive and one negative). This will have a great impact on the language (and hence on your implementation), since it will now allow several returned values, and hence, also several arguments for operations.

To give some examples:

- evaluating `{sqrt 9}` would return two results, `3` and `-3`
- `{+ 3 {sqrt 9}}` returns `6` and `0`
- `{+ {sqrt 1} {sqrt 9}}` returns four results: `4`, `2`, `-2`, and `-4`

Since the revised language deals with multiple values, it will be called "Muwae".

The WAE language is itself quite simple, so this work is not going to take long. You will do this in two steps: add a plain `sqrt` expression to the language, and then extend it so it can deal with multiple values so the `sqrt` can be made into the proper one.

Begin by downloading the WAE language implementation to serve as the basis of your work.

In the following you will be guided through the changes you are required to make to the code. You are advised to follow it step by step, but not obliged to. The end result is what you are graded for.

- **As a first step, change the name of the language to MUWAE by replacing all occurrences of the name `WAE`s in the file with `MUWAE`.**

Complete Coverage

In its original form, the WAE implementation does not have complete coverage. Extend the set of tests to get complete coverage. Make sure that when you finish your work you still have complete coverage, as required by the server. Note that your tests should all be written using `run` — this is the public interface for your language, and therefore it is the only thing that you should test.

Adding `sqrt` to the Language

The first step is to add a simple `sqrt` to the language. This is a quick job of adding a few one-liners in the right places:

1. a new rule for a unary (i.e., that takes a single input) `sqrt` in the BNF definition.
2. a new variant to the AST type definition: `Sqrt`.
3. a line to parse these expressions.
4. a line to the formal definition of `subst` for these expressions.

5. the corresponding case in the `subst` implementation.
6. a line in the formal definition of `eval`. (Note that you should use the pl procedure `sqrt` on the right-hand side as our own *simple* square-root function.)
7. the corresponding case in the `eval` definition.
8. Finally, some tests for the new functionality.

Now, if you follow the above steps and do all of these simple changes, you'll have code that should work, but instead you'll probably see a type error like:

```
Type Checker: Expected Real, but got Complex
```

The thing is that Racket has complex numbers as part of its numeric type hierarchy, which means that it can handle square roots of negative numbers just

fine. But in our language `Number` is actually the type that is known in proper Typed Racket as `Real` — this usually simplifies things, for example, you can always compare two values of type `Number`, whereas in Typed Racket you can't because it includes complex numbers too.

To make a long story short, you can resolve that by checking the result of evaluating `sqrt`'s argument expression – throw an error if it is negative, and otherwise proceed as usual. The type checker is smart enough to know that since you checked that the input is not negative, then the result must be a plain real number. To make things nice and tidy, make sure that you update the formal definition of `eval` with a similar condition (that is make sure that the formal specification of eval say that if the argument expression is evaluated to a negative number, then an error message is returned).

To help with all of this, here are some tests that you can use:

```
(test (run "{sqrt 9}") => 3)

(test (run "{sqrt 1}") => 1)

(test (run "{sqrt 0}") => 0)

(test (run "{sqrt -1}") =error> "`sqrt' requires a nonnegative
input")
```

(Note that the last test specifies the error messags that you should use in case of a negative input to `sqrt`.)

Multiple Values

Now comes the interesting part of the homework: we're going to extend the language so that instead of plain numeric values we actually deal with multiple values. As a representation for this, we replace the `Number` in the output of `eval` (and `run`) with `(Listof Number)`. The easy way to do this is to change the types, then run the code to see which type errors you get: each one is an indication of code that should change to either wrap a result number in a `list` or pull out a number from a `list`. In the latter case, just use `first` for now, to get the code into working shape as soon as possible. When you have no more type errors, you will still have the tests to modify: they should also expect to get a list of numbers instead of just the numbers.

```
(test (run "{sqrt 9}") => '(3))

(test (run "{sqrt 1}") => '(1))
```

```
(test (run "{sqrt 0}") => '(0))
```

Now that the code works again and passes all tests, you have a version of the interpreter that has the semantic change, but not much has changed since it always assumes a single number in those lists. “Fixing” this will get us the actual language that we want to have. If you followed the instructions above, then you now have `first` conveniently appearing in all places that need to change.

Fixing `sqrt`

We will begin with `sqrt`. First of all, this is the only case where instead of returning a list with a single number, you need to return a list of the two results (for each ‘sqrt’ operation in the expression).
Once you do that, some of the tests will fail, and you’ll need to fix them too as your evaluator gets closer to what it should do. For example, here’s the first test from the above, revised for this stage:

```
(test (run "{sqrt 9}") => '(3 -3))
```

An issue that you’ll run into here is what to do with `(sqrt 0)` — should it return a single `0` or two of them (the other being the result of `(- 0)`)? For this homework, we’ll go with the simple and uniform treatment, and end up with two of them:

```
(test (run "{sqrt 0}") => '(0 0))
```

We now have `sqrt` produce two outputs as it should, but it still grabs only the first number in its input. To fix this, we need to loop over all of the numbers in the input, and return a result list holding the two roots of each of these inputs. Do this by moving the code into a utility function which you should call `sqrt+`. The code in `eval` should now look like:

```
[(Sqrt e) (sqrt+ (eval e))]
```

To help you, here is a skeleton of `sqrt+`:

```
(: sqrt+ : (Listof Number) -> (Listof Number))
```

```
;; a version of `sqrt' that takes a list of numbers, and return
a list
;; with twice the elements, holding the two roots of each of
the inputs;
;; throws an error if any input is negative.
(define (sqrt+ ns)
  (cond [(null? ns) —«fill-in»—]
        [(< (first ns) 0) —«fill-in»—]
        [else —«fill-in»—]))
```

The first two “holes” are simple, the last one involves `sqrt`, two `cons`s, and a recursive call. (We won’t worry about tail-recursive calls here.)

Fixing the Arithmetic Operators

Next, we need to fix the arithmetic operators. This is a bit tricky, since each of them receives two inputs that are both lists, and they should apply the operator on each pair from these two inputs, and collect a list of all of the results. So to make it easy, here is again a skeleton of a utility function that will do this work. This time it is near-complete, and you have a tiny hole to fill:

```
(: bin-op : (Number Number -> Number) (Listof Number) (Listof Number)
            -> (Listof Number))
;; applies a binary numeric function on all combinations of numbers from
;; the two input lists, and return the list of all of the results
(define (bin-op op ls rs)
  (: helper : Number (Listof Number) -> (Listof Number))
  (define (helper l rs)
    (: f : Number -> Number)
    —«fill-in»—
    (map f rs))
```

```
  (if (null? ls)     null
    (append (helper (first ls) rs) (bin-op op (rest ls) rs))))
```

Here are some tests that should work once you're done with this part:

```
(test (run "{+ {sqrt 1} 3}") => '(4 2))
(test (run "{+ {/ {+ {sqrt 1} 3} 2} {sqrt 100}}")
       => '(12 -8 11 -9))
(test (run "{sqrt {+ 16 {* {+ 1 {sqrt 1}} {/ 9 2}}}}")
       => '(5 -5 4 -4))
```

Note that these tests are not too great: they test that the result is a list of values in a specific order, and the order can change if we modify our implementation. It would therefore be better to consider these lists as *unordered sets*, and test that the result is set-equal to some list. But we will just ignore this detail for now, to keep things simple.

Fixing the `With`

The last `first` that requires fixing is the one used in the evaluation of `With`. The problem there is that we're still using the hack of wrapping the numeric result of `eval` in a `Num` so we can use it with `subst`, and `Num` expects a single number.

One way to resolve this would be to add a new variant called `Nums` to our AST definition. But this would require reworking new cases for it in a few places... So instead, we will choose an easier solution: just change the existing `Num` so instead of holding a single `Number` it will hold a `(Listof Number)`. Once you do that, you will have three easy fixes to do. First, the code that parses numbers should put a list with the number in the `Num`. Next, there are two small fixes left in the `eval` function, and everything will work fine with it.

Don't forget to add tests that demonstrate that this works: that using `with` to bind a name to a multi-valued expression works as expected. (You need to do this test even though you should already have complete coverage at this point.)

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.

1. 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, otherwise list of the free instances.

Here are some tests:

```
(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)))) => '())
```

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.