Principles of Programming Languages, Spring 2015

Assignment 4

Submission Instruction

1. Submit your answers in a zip file named id1_id2.zip, where id1 and id1 are the IDs of the students submitting the exercise (use id1.zip if there is a single student in the group. Also, consider finding someone to work with. Contact the TAs for match-making help, if needed). A template for the archive file is provided with this exercise.
2. Questions 1 and 4 are a concept questions. Submit your answer to them in a file named ex4_q1.pdf and ex4_q4.pdf respectively. Those files should be in the top-level directory of the submitted archive (see template).
3. The rest of the questions are coding exercises - submit your answers to each of them in a directory named ex4_qn, where n is the number of the question.
4. A contract must be included for every implemented procedure.
5. Make sure the .rktfile is in the correct format (see the announcement about "WXME format" in the course web page).
6. As your code is tested automatically, you must use exact procedure and file names. Again, we recommend using the provided template.
7. Each directory contains a test file with some tests for your code. These tests are meant to help you write a bugless code. Use them! make sure your code passes all the tests (of course all the regular tests of the evaluators should pass too). *Your code will be checked with another set of tests, so write a more thorough set of tests to make sure your code is correct.* Note that in some folders, the tests are in a nested folder (e.g. “substitution-interpreter” and not to top-level one).

1. Concepts

a. Explain the concept of *Value* data structure in the Substitution evaluator.
b. Discuss the advantages and disadvantages of keeping a small language kernel and large libraries of derived operators?
c. What are the advantages of lexical scoping computation policy over dynamic scoping computation policy?
d. What is the main reason for switching from the Substitution Model to the Environment Model?
e. The constructor `make-primitive-procedure`: Where is it used in the evaluators?
f. What is the conceptual improvement in the Analyzer over the Environment interpreter?
g. Can we apply the partial evaluation (Currying) technique used in the analyzer on the substitution model?

h. In question 3 in this assignment you are asked to implement the procedure `exists?` as a special form (see below). Is it possible to implement it as a derived expression? explain.

2. Abstract Syntax Parser (ASP): `or->if`

An `or` expression can be written as an `if` expression, for example, the expression:

```
(or #f 4 #t)
```

Is equivalent to the expression:

```
(if #f #f
    (if 4 4
        (if #t #t #f)))
```

(notice that side effects are not supported. Think why! just think, no need to answer). Write the function or>if that derives an or expression into an if one. In addition, you will need to alter two functions: shallow-derive and derived?(both in the same file).

Step 1:

Implement the methods: `or?`, `or-args`, `or-first-arg`, `or-rest-args`, `or-last-arg`, `or-empty-args?` and `make-or`.

Step 2:

Modify `derived?` and `shallow-derive` to support the new syntax.

Step 3:

Implement `or->if`. Notice that there's no need to derive all of the arguments. You can derive only the first one and `derive` will take care of the rest.

3. Substitution Model Interpreter

Add a special form `exists?` that is applied on a variable, that returns whether this variable is bound to a value in the global environment. For example:

```
> (derive-eval '(define x 5))
> (derive-eval '(exists? x))
#t
> (derive-eval '(exists? y))
#f
```

Step 1: ASP

Add `exists??` (notice the two '?') and `var-exists?` to the ASP.
Modify `special-form?` to support the new special form.

Step 2: Core & DS

Add `eval-exists?` to the core. Since this procedure needs to lookup the variable in the global environment, you will need to use the global environment ADT in the DS. You may modify the DS if you think you need to. When you're done with it, don't forget to modify `eval-special-form`.

```
;; Signature: eval-exists?(exists-exp)
;; Purpose: checks is a variable is bounded to a value in the global
;; environment.
;; Example:
;; (eval-exists? '(exists x)) => '(value #f)
;; (derive-eval '(define x 5)) => 'ok
;; (eval-exists? '(exists x)) => '(value #t)
```

4. Environment Model

Draw an environment diagram for the following computation. Make sure to include the static block markers (copy the given code and mark the block markers on it), the control links and the returned values.

```
(define even?$
  (lambda (n c)
    (if (= n 0)
        (c #t)
        (odd?$ (- n 1) c))))

(define odd?$
  (lambda (n c)
    (if (= n 0)
        (c #f)
        (even?$ (- n 1) (lambda (x) (c x))))))

(even?$ 3 (lambda (x) x))
```

5. Environment Model Interpreter

Add a special form `exists?` that is applied on a variable, that returns whether this variable is bound to a value in the current environment. For example:

```
> (derive-eval '(define x 5))
> (derive-eval '(exists? x))
#t
> (derive-eval '(exists? y))
#f
```

```
> (derive-eval `(let ((y 5))
(exists? y)))
#t
> (derive-eval `(let ((y 5))
(exists? x)))
#t
> (derive-eval `(let ((y 5))
(exists? z)))
#f
```

Notice that `exists?` should look for a variable in the current environment, meaning that the implementation should get the environment as a parameter. Your implementation should traverse the environment and look for the variable. Use the environment ADT.

Step 1: ASP

Add `exists??` (notice the two '?') and `var-exists?` to the ASP.
Modify `special-form?` to support the new special form.

Step 2: Core & DS

Add `eval-exists?` to the core. Since this procedure needs to lookup the variable in the global environment, you will need to use the global environment ADT in the DS. You may modify the DS if you think you need to. When you're done with it, don't forget to modify `eval-special-form`.

6. Analyzer (lambda variadic)

Scheme procedures can be defined to have a changing number of parameters. For example, the primitive functions '+' and '*' can get an arbitrary number of arguments:

```
(+ 1 3 4) -> 8
```

Such procedures are called *variadic*. This question deals with extending the compiler (analyzer) to support variadic user procedures.

Variadic user procedures are defined using the special operator lambda, with a single parameter with no parenthesis. For example:

```
(lambda lst (length lst))
```

The concrete syntax of lambda-variadic expressions is:

```
(lambda <parameter> <body>)
```

Evaluation of a lambda-variadic expression produces a procedure-variadic value. A procedure-variadic has a single parameter which is a list of the arguments from

the application.

The syntactically of an application expression with a variadic procedure is the same as that of a regular application. The difference is in the way the single parameter is bound to the arguments: The parameter is bound to a list containing the arguments. Some examples:

```
> ((lambda args (car args)) 1 2 3) -> 1
> ((lambda lst (length lst)) 'a 'b 6) -> 3
> ((lambda x x) 5) -> (5) ; Note the parenthesis!!
```

Extend the analyzer to support lambda variadic expressions:

Step 1

Add the variadic lambda expression to the ASP interface: add all the needed procedures: (`lambda-variadic?`, `lambda-variadic-parameters` etc) and modify `special-form?` to support the new special form)

Step 2

Add a new data structure (much like the data structure for user procedures).

Step 3

Modify the core: note that the modification involves both the definition and the application of variadic procedures. lambda-variadic should be handled as a new special operator. apply-procedure should be modified to apply variadic procedures as well.