University of Virginia, Department of Computer Science
cs150: Computer Science — Spring 2007

Out: 26 March

Problem Set 7: Charming Snakes and Mesmerizing Memoizers

Due: Friday, 6 April

Collaboration Policy - Read Carefully

For this assignment, you make work either alone or with a partner of your choice.

Regardless of whether you work alone or with a partner, you are encouraged to discuss this assignment with other students in the class and ask and provide help in useful ways. You may consult any outside resources you wish including books, papers, web sites and people. If you use resources other than the class materials, indicate what you used along with your answer.

Purpose

- Understand how language interpreters work
- Understand the meta-circular evaluator
- Gain experience with Python
- Learn how changing the evaluation rules changes programming

Read Chapter 12 of the course book. Become familiar with the Python Guide as a reference.

Download: Download *charme.py* to your machine and save it into your home directory `J:\cs150\ps7`. This file contains a Python implementation of an interpreter for the Scheme-like language Charme. The implementation closely matches the description in Chapter 12.

For most of the questions in this assignment, you will modify *charme.py*. When you turn in your answers, please just turn in the changes you have made for each question, in a clearly marked way. You should not turn in print-outs of provided code that you did not modify.

Background

Languages are powerful tools for thinking. One way to solve problems is to think of a language in which a solution to the problem can be easily expressed, and then to implement that language. This is the "great Hop forward" that Grace Hopper made in the 1950s: we can produce programs that implement languages. The input to the program is an expression specification in some language. If the program is an interpreter, the result is the value of that expression.

In this problem set, we provide a working interpreter for the Charme language, which is approximately a subset of Scheme. The interpreter implements the Scheme evaluation rules with state (from Chapter 9). Your assignment involves understanding the interpreter and making some additions and changes to it. If you are successful, you will produce an interpreter for the language Mesmerize in which the original Fibonacci procedure can be applied to 60 to produce the correct result in a reasonable amount of time.

Getting Started

First, try running the Charme interpreter. We recommend using, IDLE, the Python interpreter and development environment provided in the ITC lab machines. IDLE provides an editor for Python code and an interpreter somewhat similar in style to DrScheme. After you have extracted *charme.py* from the *ps7.zip* file, you can open it in IDLE by right-clicking on the file and selecting "Edit with IDLE" (the second option listed). This opens both a Python Shell (analagous to the interactions buffer) and the editor containing *charme.py* in separate windows.

Try evaluating some Python statments in the interpreter window. To get familiar with Python, try to define the `intsto` procedure that takes a positive integer as its input and produces a list of the integers from 1 up to the input value. For example, `intsto(5)` should evaluate to `[1, 2, 3, 4, 5]`. If you get stuck, follow this link for one definition.

In the *charme.py* window, select `Run | Run Module` (F5) to load the definitions into the intepreter. You can try some evaluations using `meval` directly. First, evaluate

```
initializeGlobalEnvironment()
```

to initialize the global environment. It is stored in the global variable `globalEnvironment`. The `meval` procedure takes two parameters. The first is a structured list specifying a Charme expression (the result of `parse`); the second is the environment in which that expression should be evaluated. For example,

```
>>> meval("23", globalEnvironment)
23
>>> meval(['+', '1', '2', ['*', '3', '4']], globalEnvironment)
15
```

This is a painful way to evaluate expressions, since the input needs to be a structured list. The `parse` procedure takes a string representing one or more Charme expressions and produces as output a list containing each input expression as a structured list. For example,

```
>>> parse("23")
['23']
>>> parse("(+ 1 2 (* 3 4))")
[['+', '1', '2', ['*', '3', '4']]]
>>> parse(""(define square (lambda (x) (* x x)))")
[['define', 'square', ['lambda', ['x'], ['*', 'x', 'x']]]]
```

The `evalLoop()` procedure combines `parse` and `meval` to provide a convenient interpreter for Charme. Try evaluating `evalLoop()` and evaluating some Charme expressions.

> **1.** Define a factorial procedure in Charme. (Note that Charme does not provide the if expression, so the standard Scheme definition will not work.)

Adding Primitives

The set of primitives provided by our Charme interpreter is sufficient (that is, enough to express every computation), but very impovished (not enough to express every computation in a convenient way).

> **2.** Extend the Charme interpreter by adding a primitive procedure `<=` to the global environment. You will need to define a procedure that implements the primitive, and modify `initializeGlobalEnvironment` to install your primitive.

Our Charme interpreter does not provide any primitives for lists. As we saw in Chapter 5, it is possible to define `cons`, `car` and `cdr` using only the language already defined by Charme. However, it would be more convenient if some primitives for manipulating cons cells and lists are provided.

> **3.** Extend the Charme interpreter by adding primitive procedures `cons`, `car` and `cdr` that behave similarly to the primitive Scheme procedures.

One suggestion for implementing these primitives is to start by defining a class that represents a cons cell. For example, you could define a `Cons` class that has a constructor (`__init__`) that takes two inputs (the first and second parts of the pair), and provides methods for `getFirst` and `getSecond` that retrieve the respective parts of the pair.

You may also want to change the `evalLoop` procedure to display cons cells similarly to how they are displayed in Scheme. You could do this by adding a `toString` method to your `Cons` class, and adding a clause starting with `elif isinstance(res, Cons):` to the `evalLoop`.

Once your new primitives are installed, you should get the following interactions in the `evalLoop()`:

```
Charme> (cons 1 2)
(1 . 2)
Charme> (car (cons 1 2))
1
Charme> (car (cdr (cons 1 2)))
2
```

> **4.** Extend the Charme interpreter by defining the `null` and `null?` primitives.

You could use Python's `None` value to represent `null`.

> **5.** Extend the Charme interpreter by defining the `list` primitive prodcedure. Like the Scheme `list` primitive procedure, it should take any number of operands and produce as output a list containing each operand as an element in order.

Once your new primitives are installed, you should get the following interactions in the `evalLoop()`:

```
Charme> (define a (list 1 2 3 4))
Charme> (car a)
1
Charme> (null? a)
False
Charme> (cdr (cdr a))
(3 4)
```

It is acceptable if your list does not print out quite like this, but better if you can make it print out correctly.

```
Charme> (null? (list ))
True
```

Special Forms

> **6.** Extend the Charme interpreter to support the `if` expression special form, with the same meaning as the Scheme if expression.

After adding `if` to your Charme interpreter, you should get the following interactions (note: we recommend testing it with simpler tests before trying this):

```
Charme> (define fibo (lambda (n) (if (= n 1) 1 (if (= n 2) 1 (+ (fibo (- n 1))
(fibo (- n 2)))))))
Charme> (fibo 5)
5/p>
```

Memoizing

Memoizing is a technique for improving efficiency by storing and reusing the results of previous procedure applications. If a procedure has no side effects and uses no global variables, everytime it is evaluated on the same operands it produces the same result.

To implement memoizing, we need to save the results of all procedure applications in a table. When a procedure application is evaluated, first, we lookup the application in the table to see if it has already been computed. If there is a known value, that is the result of the evaluation and no further computation need be done. If there is not, then the procedure application is evaluated, the result is stored in the table, and the result is returned as the value.

> **7.** Modify the Charme interpreter to support memoizing for all procedure applications. (Hint: the Python dictionary datatype will be very useful for this.)

Once you have modified the interpreter, you should be able to evaluate `(fibo 60)` without changing the definition of `fibo` above. By changing the meaning of the application evaluation rule, a procedure that previously had running time exponential in the value of the input, now has running time that is linear in the value of the input!