

# Solutions to Homework 1

Chris Fensch & Chris Stork

August 23, 2004

All exercises are taken from Mitchell: *Concepts in Programming Languages*. Numbers in parenthesis refer to exercise numbers.

Use Scheme (R5RS) to answer the Lisp questions. This way you can experiment with DrScheme to find the solutions.

## 1 Halting Problem on No Input (2.2)

In principal, there are two ways to answer this question. The first one is to repeat the proof for the original halting problem. The second answer is to show that using  $\text{Halt}_0$  the original halting problem would be solvable. Since, we know it cannot be solved, it would mean that such a function  $\text{Halt}_0$  cannot exist.

The book gives you already an idea how to do it. The idea with a single integer is just a simplification, however the general idea would be the same. First assume that the program  $P$ , which we want to test, is written in C. According to the book we can assume that it will start like this:

```
[...]  
  
void main() {  
    int x = read(); /* read a single integer as input */  
  
    [...]  
  
}
```

Now we construct a program that checks if  $P$  halts. Our program is passed the following arguments:

- $P$ 's source code as a string
- the integer, e.g. 12345678, that the program should read

Also our program has access to the function  $\text{Halt}_0$  and can simply call it. We can now write the following program:

- i. Modify  $P$ 's `read()` statement in such a way that it no longer reads a value, but instead assigns the value it should read directly. For example:

```

void main() {
    int x = 12345678;

    [...]

}

```

- ii. Call  $\text{Halt}_0$  with the modified program as an argument.
- iii. Return the result of  $\text{Halt}_0$ .

Obviously this program would solve the general halting problem, if such a function as  $\text{Halt}_0$  would exist. Hence, it cannot exist.

## 2 Conditional Expressions in Lisp (3.2)

- (a) There is no way to implement the interpretation as described. Perhaps certain undefined expressions (such as division by zero) could be handled with modern languages (i.e. catch the exception), but expressions that are undefined due to non-termination cannot be handled in general due to the Halting Problem. (It is impossible to determine in advance whether evaluation of a particular condition will halt or not.) If a function does not halt, occurs in a conditional expression, it will never return a value (or throw an exception); therefore subsequent conditions will never be evaluated.

Concrete counterexample:  $p_1$  runs forever, but due to the Halting Problem this cannot be determined in advance;  $p_2$  is the truth constant. Then, according to McCarthy's definition  $e_2$ 's value should be returned, but since we have to wait for  $p_1$ 's evaluation to terminate, we'll never get to  $p_2$ .

- (b) By using parallel processing, each conditional could be evaluated in increments. Therefore, even if a particular conditional expression does not halt, subsequent expressions would still be allowed to evaluate. Once some condition evaluated to true, other evaluations would be terminated and the appropriate value would be returned. It would still be undefined if none of the expressions evaluated to true (allowed per property i) and it might not be the first true expression in the code that completes its evaluation first (allowed per property ii).
- (c) The problem with the code is that several conditions can be true at the same time (e.g. the  $\tau$  condition is always true). For example, consider an implementation of b) that ends up alternating between evaluating the third and fourth branch, oscillating between the values 5 and 3. Hence, in order to work with the implementation in b) we have to make sure that exactly one condition is true at a time (in Scheme not Lisp):

```

(define odd
  (lambda (x)

```

```
(cond ((eq x 0) #f)
      ((eq x 1) #t)
      ((> x 1) (odd (- x 2)))
      ((< x 0) (odd (+ x 2))))))
```

- (d)
- The original interpretation (NOT Part A) would best be used to implement the short-circuiting OR since the original definition of `cond` is already short-circuiting.
  - The second interpretation (Part B, parallel processing) would best be used to implement the parallel OR. Parallel processing is necessary to ensure evaluation of  $e_2$ , even if  $e_1$  does not halt.
  - The parallel processing interpretation (b) and the unimplementable version (a) would make implementation of the short-circuiting OR difficult because they both return true in the case that  $e_1$  is undefined and  $e_2$  is true. SCOR should be undefined in this case.
  - The original interpretation makes implementation of the parallel OR difficult because if  $e_1$  was undefined or did not halt,  $e_2$  would never be evaluated. This contradicts the notion of the parallel OR which requires a return value of true if either  $e_1$  or  $e_2$  is true, regardless of whether or not the other condition is undefined.

### 3 Lisp and Higher-Order Functions (3.4)

First you want to look at `maplist`. In the example, `maplist` is passed a function `f` that takes a list as an argument and returns a list as a result. Now consider the following. What would `maplist` return, if you pass `car` that takes a list and returns just a single element? Type it into DrScheme and look what happens! It basically just returns the argument:

```
> (maplist car '(1 2 3 4))
(1 2 3 4)
```

Now, this exercise is about playing with functions. We want to construct a function that allows us to compose the `car` with a function `f` so that we get the same effect as the well-known `mapcar` function. More concretely, we try to construct a convoluted function `compose2` which allows us to write

```
((compose2 maplist car) f xs) = (mapcar f xs)
```

So `compose2` is a function that expects two functions as arguments. These two functions are passed through `((lambda (g h) ...))`. Hence

- (b) `g` is being replaced by `maplist` and `h` is replaced by `car`.
- (c) Looking at `compose2` after the replacements of (b) we have

```
(lambda (f xs)
  (maplist (lambda (xs) ...) xs))
```

and we see that `(lambda (xs) (...))` should be replaced by `(compose f car)` in order to have `f` operate on the elements of the original list. (That's the intuition gained in the intro.) We see that `h` is actually `car`, so we can we have the final answer `(compose f h)`.

- (a) Given (c), we notice that

```
(compose f h) = (lambda (x) (f (h x)))
```

which means that we are looking for the substitution

```
f (h xs)
```

QED.

## 4 Definition of Garbage (3.5)

- (a) If a memory location is garbage according to the book's general definition, it is not necessarily garbage according to McCarthy's definition. For example, a declared variable might never be referred to in a program and be classified as garbage according to the book's definition, but would not be classified as garbage according to McCarthy's definition because it is still accessible if the program would just "decide" to do so.
- (b) If a memory location is garbage according to McCarthy's definition, it is necessarily garbage according to the book's general definition. We know that if a memory location is not accessible by an operation on any base register (aka pointer in the root set) then it is safe to assume that no possible course of program execution should access the location.
- (c) It is impossible to write a garbage collector to collect everything that is garbage according to the book's general definition of garbage. To do so would require an algorithm that anticipates every possible course of program execution. Since no algorithm can even tell us whether or not a program will halt, we certainly cannot anticipate every possible execution path.

## 5 Lambda Calculus Reduction (4.3)

$$\begin{aligned}
 & (\lambda x. \lambda y. xy)(\lambda x. xy) \\
 =_{\alpha} & (\lambda x. \lambda z. xz)(\lambda x. xy) \\
 =_{\alpha} & (\lambda x. \lambda z. xz)(\lambda v. vy) \\
 =_{\beta} & \lambda z. ((\lambda v. vy)z) \\
 =_{\beta} & \lambda z. zy
 \end{aligned}$$

Not performing the first  $\alpha$ -substitution could lead to a capture of the free variable  $y$ . (The second  $\alpha$ -substitution was just performed for clarity's sake.)

## 6 Symbolic Evaluation (4.4)

In the following description a  $\lambda^*$  refers to a lambda abstraction that can be applied.

**left-most**

$$\begin{aligned}
 & \overline{(\overline{(\lambda^* f. \lambda g. f(g1))(\lambda x. x + 4)})} (\lambda y. 3 - y) \\
 = & \overline{(\lambda^* g. (\lambda^* x. x + 4)(g1))(\lambda y. 3 - y)} \\
 = & \overline{(\lambda^* x. x + 4)((\lambda^* y. 3 - y)1)} \\
 = & \overline{((\lambda^* y. 3 - y)1) + 4} \\
 = & ((3 - 1) + 4)
 \end{aligned}$$

**right-most**

$$\begin{aligned}
 & \overline{(\overline{(\lambda^* f. \lambda g. f(g1))(\lambda x. x + 4)})} (\lambda y. 3 - y) \\
 = & \overline{(\lambda^* g. \overline{(\lambda^* x. x + 4)(g1)})} (\lambda y. 3 - y) \\
 = & \overline{(\lambda^* g. (g1) + 4)} (\lambda y. 3 - y) \\
 = & \overline{((\lambda^* y. 3 - y)1) + 4} \\
 = & ((3 - 1) + 4)
 \end{aligned}$$

## 7 Order of Evaluation (4.7)

Each of the arguments e1, e2 and e3 has to be a list (or at least a pair). The idea is now that all of these expressions return the same list (for example a). Since function f takes the first element from the lists that are passed through e1 and e2, it would be a good idea to change one of these using `rp1aca`.

```

(define a '(1 2 3 4))
(define (rplaca x y) (set-car! x y) x)
(define f (lambda (x y z) (cons (car x) (cons (car y) (cdr z)))))

(f a (rplaca a 5) a)

```

Depending on the order of evaluation we get different results:

**left to right** (1 5 2 3 4)

**right to left** (5 5 2 3 4)

*Additional Question: What's the evaluation order for functions (aka procedures) according to R5RS?*

According to Section 4.1.3 of the Revised<sup>5</sup> Scheme Report (R5SR), “The operator and operand expressions are evaluated (*in an unspecified order*) and the resulting procedure is passed the resulting arguments.” or “In contrast to other dialects of Lisp, the order of evaluation is unspecified...”.

## 8 Denotational Semantics (4.8)

(a)

$$\begin{aligned}
 & \mathcal{C}[[x := 1; x := x + 1]](s_0) && \text{with } s_0(x) = 0 \\
 & = \mathcal{C}[[x := x + 1]](\mathcal{C}[[x := 1]](s_0)) \\
 & = \mathcal{C}[[x := x + 1]](\text{modify}(s_0, x, 1)) \\
 & = \mathcal{C}[[x := x + 1]](s_1) && \text{with } s_1(x) = 1 \\
 & = \text{modify}(s_1, x, x + 1) \\
 & = s_2 && \text{with } s_2(x) = 2
 \end{aligned}$$

(b) There are several different ways to compute the the same functions. For example, to compute  $f(x) = 2x$  you could:

- simply add two times x: `return x + x;`
- use multiplication: `return x * 2;`
- use a binary shift to left by one: `return x << 1;`

All this does not change that all these programs compute the same function (ignoring some edge cases). If they are the same function, it means that they have the same denotation.

Hence, using denotational semantics, we see that both sides of the equation in (b) have the same meaning:

$$\lambda v. \text{ if } v = x \text{ then } 2 \text{ else } s(v).$$

## 9 Functional and Imperative Programs (4.13)

There are a lot of reasonable answers here. As long as you wrote a nice explanation, why you think your view is correct, you got full credit. A simple “yes” or “no” won’t do.

- (a) There aren’t really any inherent reasons why imperative languages are better suited than functional languages for day-to-day programming tasks. There may be reasons why one style or the other is better suited to a particular problem; operating system kernels are probably more easily written in imperative languages, for example. The emergence of Java (an imperative language which none-the-less includes, since the actions of an operating system have a do this, do that flavor. But even this statement is debatable: programmers who like using functional languages have found ways to think functionally about most kinds of programming problems.
- (b) Imperative languages (as defined in this problem) require less run-time support, and are therefore better-suited to machines with limited resources.
- (c) Functional languages will generally produce larger executables. In particular, in order to do garbage collection at run-time, executables must contain code to do garbage collection. This can make the minimum size of an executable large; many LISP systems will generate “Hello, world.” programs that are as many as four megabytes big.
- (d) In the early days of computing, resources were limited. Therefore, it’s easy to see why imperative languages took off. Once people had learned one imperative language that worked well enough for them, there was little motivation to learn a new functional language, even when machines began to have more memory than we know what to do with.
- (e) As we enter the 21st century, these concerns still matter; there are more programs being written for small embedded devices than for full-size computers at this point. Many of these devices have as little as 128 kilobytes of memory; there is simply no room for support for garbage-collection and higher-order functions. However, this might change soon again...