Concepts of programming languages: Scheme exercise

J. Kok and Jeroen de Bruin,
edited by Sander van der Maar and Thijs van Ommen

September 20, 2007

1 Scheme implementation

For this exercise, we use an Open Source [1] implementation of the Scheme language: the Free Software Foundation's [2] Guile ("GNU's Ubiquitous Intelligent Language for Extension") [3].

Version 1.8.2 of the Guile interpreter has been compiled for Linux/x86 systems, and is available as

```
/home/mvommen/cvp/bin/guile
```

It can be used interactively, or via a script. To make a script, put your Scheme code in a file (say `foo.scm`), have that file start with

```
#! /home/mvommen/cvp/bin/guile -s
!#
```

to denote the interpreter[1] and make that file executable: `chmod +x foo.scm`

There are many other Scheme implementations available, but there may be slight differences between them. While it is perfectly alright to use one of them to develop your code, the code you submit has to work with Guile.

2 Exercises

Scheme is not a pure functional language because it has some procedures that have side effects (for example `set-car!` which is like variable assignment in an imperative language). In this exercise, we restrict ourselves to the pure functional subset of Scheme. `set-car!` and other procedures that have side-effects **may not be used!**

Exercise 1: Substitution

Suppose you have a (nested) list `'((a (b c) d) a (f b))` and a list of pairs `'((a z) (b y))`, that describe replacements ('a' is replaced by 'z', 'b' is replaced by 'y'), you can perform substitution on the original list (resulting in `'((z (y c) d) z (f y))`).

Exercise 1A: Substition in Scheme

Write a Scheme function `substitute` that takes two arguments: the first a (possibly nested) list (where the atoms are 1-character strings); the second a list of replacements (each describing a 1-character string and its 1-character replacement); it should yield the list which results from performing the substitutions specified in the list pairs on the input list.

E.g. the following should return `#t`:

[1] Unix shells read the `#! /path/to/binary arguments` in scripts to tell them how to invoke the relevant interpreter; Guile regards `#! ... !#` as a comment (similar to C's `/* ... */`).

```
(equal?
        (substitute
                '((a (b c) d) a (f b))
                '((a z) (b y))
        )
        '((z (y c) d) z (f y))
)
```

Exercise 1B: Substition in C++

Write a C++ program to do the same, and compare the Scheme code to the C++ code. Your C++ program must reflect the nature of (recursive) lists (so no quick hacks using strings). What differences do you notice? Are these due to the difference in programming paradigm (functional vs. imperative/object-oriented) or do they have other causes?

Exercise 2: Treewalk

Write a Scheme function `treewalk` that walks through a binary tree, encoded in infix order (<left-tree> <node> <right-tree>), in breadth first order and that yields a list containing the node contents.

E.g. the following (walking through the tree in Figure 1) should return `#t`:

```
(equal?
        (treewalk
                '( (b) a ( ((f) d (g)) c (() e (h)) ) )
        )
        '(a b c d e f g h)
)
```

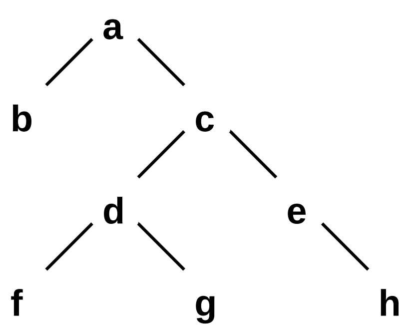

Figure 1: The example tree for the second exercise

3 How to submit

Your programs should be submitted together with a written report in which you explain your programs, to Thijs van Ommen (mvommen@liacs.nl).

References

[1] http://www.opensource.org

[2] http://www.fsf.org

[3] http://www.gnu.org/software/guile/guile.html