University of Arizona, Department of Computer Science

CSc 520 — Assignment 1 — Due noon, Wed Feb 2 — 5%

Christian Collberg
January 26, 2005

1 Introduction

The purpose of this assignment is for you to become familiar with Scheme, DrScheme, and writing recursive functions.

Before starting this assignment, set your DrScheme language level to **Standard (R5RS)**.

All your function definitions should be *pure*, i.e. they should not use any of Scheme's imperative features such as `set!`. Also, never use iteration, always recursion.

Every function should be commented. At the very least, the comments should state what the function does, which arguments it takes, and what result it produces.

This assignment is graded out of 100. It is worth 5% of your final grade.

2 Simple functions

1. Define a recursive function `(copy-string s n)` which returns a string consisting of `n` copies of the string `s`: [10 points]

```
(define (copy-string s n)
   (cond
      [(<= n 0) ...]
      [(= n 1) ...]
      [else ...]
   )
)
```

Your function should have the following behavior:

```
> (copy-string "hello" -1)
""
> (copy-string "hello" 0)
""
> (copy-string "hello" 1)
"hello"
> (copy-string "hello" 2)
"hellohello"
> (copy-string "hello" 10)
"hellohellohellohellohellohellohellohellohellohello"
```

2. Define a recursive function `(power-of-two? n)` which returns `#t` if `n` is a power of two (i.e. $n = 2^m$), and `#f` otherwise: [10 points]

```
(define (power-of-two? n)
    (cond
        ....
    )
)
```

Your function should have the following behavior:

```
> (power-of-two? 0)
#f
> (power-of-two? -4)
#f
> (power-of-two? 1)
#f
> (power-of-two? 2)
#t
> (power-of-two? 3)
#f
> (power-of-two? 4)
#t
> (power-of-two? 6)
#f
> (power-of-two? 8)
#t
```

3 A Metacircular Interpreter

Extend the metacircular interpreter from lecture notes #9 with the functionality below. In all cases you can assume that the input programs are correct, i.e. you don't need to check for error conditions such as *undeclared identifier*.

1. `(begin` *arg1* $\cdots$ *argn*`)`: [20 points]

```
> (mEval '(begin (display 55) (newline) (display 66) (newline) (+ 4 5)))
55
66
9
```

Note that `begin` should return the last value computed. *Don't* use Scheme's built-in `begin`-function in your implementation!

2. `(cond (`*expr1 arg1*`)` $\cdots$ `(`*exprn argn*`))`: [20 points]

```
> (mEval '(cond ((eq? 1 2) (display 55)) ((eq? 2 2) (display 66))))
66
```

Don't use Scheme's built-in `cond`-function in your implementation (you can use `if`, however)!

3. `(equal?` *expr1 expr2*`)`: [20 points]

```
> (mEval '(equal? (quote (1 (2))) (quote (1 (2)))))
#t
> (mEval '(equal? (quote (1 (2))) (quote (1 (2 (3))))))
#f
```

Don't use Scheme's built-in `equal?`-function in your implementation (you can use `eq?`, however)!

4. `define` and variable references: [20 points]

```
> (mEval '(begin
            (define x 44)
            (display x) (newline)
            (define x (+ x 11))
            (display x) (newline))
)
44
55
```

To implement this funcitonality you need to add an argument `Env` to each function. `Env` stores current variable values. It should be implemented as an association-list of variable/value pairs.

4 Submission and Assessment

The deadline for this assignment is noon, Wed Feb 2. You should submit the assignment (a text-file containing the function definitions) electronically using the `Unix` command ⌜`turnin cs520.1 <files>`⌝. This assignment is worth 5% of your final grade.

Don't show your code to anyone, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the instructor or TA.