

CM20167: Functional Programming

Coursework assignment 2007–8

Guy McCusker
G.A.McCusker@bath.ac.uk

Administrative details

This document describes the coursework assignment for CM20167, *Programming III*, also known as *Functional Programming*, for the 2007–8 academic session. Below are the vital facts and figures.

Date set:	19th October 2007
Submission deadline:	4pm, December 13th 2007
Submission location:	postboxes outside 1W2.23
Proportion of unit assessment:	25%
Feedback provided:	orally in class plus individual written comments

Problem description

For this assignment you are asked to implement the λ -calculus using Lisp, or Scheme if you prefer.

Preferably, use EuScheme (which is Lisp) or mzscheme (which is Scheme) as provided on the BUCS machines. If you plan to use another Lisp or Scheme system, please check with Guy McCusker that it will be acceptable.

1. λ -calculus terms can be represented using lists as follows:

Term	Representation	
x	(VAR x)	where x is some symbol
$\lambda x.M$	(LAMBDA x term)	where x is the symbol from the representation of x and term is the representation of M
MN	(APP term1 term2)	where term1 is the representation of M and term2 is the representation of N .

Thus a term such as $\lambda x.\lambda y.\lambda z.(xz)(yz)$ is represented as

```
(LAMBDA x (LAMBDA y (LAMBDA z (APP (APP (VAR x) (VAR z)) (APP (VAR y) (VAR z))))))
```

Write the following helper functions:

- (a) `variable?`, `abstraction?`, and `application?` which tell us whether or not a given (representation of a) term is a variable, an abstraction, and an application respectively. Thus `(variable? '(VAR x))` should return a true value while `(application? '(VAR x))` should return false (the empty list in Lisp).
- (b) `var-symbol` which returns the symbol from a variable term, and `make-var` which takes a symbol and returns the representation of a variable using that symbol.
- (c) `abs-var` and `abs-body` which respectively return the variable symbol and the body of an abstraction term, and `make-abs` which takes a symbol and (the representation of) a term and returns (the representation of) a corresponding abstraction term.
- (d) `app-fn` and `app-arg` which respectively return the function term and argument term of an application, and `make-app` which takes two (representations of) terms and returns an application term.

So for instance

```
(make-abs 'x (make-app (make-var 'x) (make-var 'x)))
```

should return

```
(LAMBDA x (APP (VAR x) (VAR x)))
```

and then applying `abs-body` to this should return

```
(APP (VAR x) (VAR x)).
```

2. Write a function `alphaconvert` for α -conversion of λ -terms.

`(alphaconvert term x y)` should return a term computed by replacing all free occurrences of the variable `(VAR x)` in `term` by `(VAR y)`.

3. Write a function `subs` to implement substitution.

If we have λ -terms M and N and a variable x , represented by Lisp terms `M`, `N` and `(VAR x)`, then `(subs M x N)` should compute a representation of the term $M[N/x]$.

Take care to avoid capture of free variables of N when performing substitution. You might like to use the Lisp function `gensym` to help you here.

4. Implement β -reduction:

- (a) write a function `redex?` which tells us whether a given term is a β -redex or not.
- (b) write a function `contains-redex?` which tells us whether a given term contains any redexes or not.

- (c) write a function `normalbeta` which takes a term and performs one step of β -reduction using the normal-order reduction strategy.
5. Implement reduction to normal form: (`normal-reduce M`) should compute the normal form of M , if it has one, using β -reductions in normal order.
 6. Implement reduction to weak head-normal form. A term is in weak head-normal form (whnf) if it is of any of the following forms:
 - x
 - $\lambda x.M$
 - MN where M is a whnf and is not an abstraction
- (a) write a function `whnf?` which tells us whether a term is in weak head-normal form or not.
 - (b) write a function (`whnf-reduce M`) should compute the weak head-normal form of M , if it has one, using β -reductions in normal order.

Deliverables

You should submit a listing of your source code, plus a document containing a selection of examples which demonstrate that you have tested your programs thoroughly. You should also include any information that will help the examiner to understand your code and your examples. In particular you should explain how to use `lambda-reduce` to obtain the same effect as `normal-reduce` and `whnf-reduce`.

Please make your source code available on-line, preferably by placing a readable copy on the BUCS network after the submission deadline, and include details of where your code can be found in your submission.

Conditions

This is individual coursework: all work must be your own. As with all assessment this coursework is subject to the University regulations on Plagiarism, a summary of which can be found in the assessment section of the Computer Science Undergraduate Programmes Handbook.

Marking Scheme

1. Helper functions: 2 marks
2. α -conversion: 5 marks
3. Substitution: 5 marks

4. β -reduction: 4 marks
5. Reduction to normal form: 2 marks
6. Reduction to weak head-normal form: 2 marks
7. Testing and documentation: 5 marks.

There are 25 marks available in total.