

THIS IS NOT AN OPEN-BOOK EXAMINATION.
CANDIDATES MAY NOT CONSULT ANY REFERENCE MATERIAL DURING THE SITTING.

NO CALCULATOR PERMITTED IN THIS EXAMINATION

THE UNIVERSITY OF BIRMINGHAM

Degree of B.Sc. with Honours
Computer Science/Software Engineering. Final Examination.
Computer Science/Software Engineering with Business Studies. Final Examination.
Artificial Intelligence and Computer Science. Final Examination.

Joint Degree of B.Sc. with Honours
Mathematics and Computer Science. Final Examination.

Degree of MSc in Computer Science.

06 02552

()

Principles of Programming Languages

May 2001 2 hours

[Answer ALL questions]

Turn Over

-
1. Consider a simple expression language for literals (constants), addition and multiplication. The abstract syntax is given by the following Scheme records:

```
(define-record lit (datum))  
(define-record add (left right))  
(define-record mult (left right))
```

- (a) Write a recursive Scheme procedure `eval-exp` that interprets expressions in this language: `add` should be interpreted by `+` and `mult` by `*`, and a `lit` should evaluate to its `datum`. (Recall that records are pattern-matched with `variant-record`.) [10%]
(b) Extend the expression language with variables and declarations as follows:

```
(define-record varref (var))  
(define-record letdec (var defexp body))
```

The `letdec` is intended to bind a (single variable) `var` to `defexp` in `body`.

Rewrite `eval-exp` for the extended language.

(You may use two helper functions `lookup` and `extend` like those mentioned in the course.)

[10%]

- (c) For each of your `eval-exp` procedures, state if it takes any of these as an argument:
- environment
 - store
 - continuation

If so, explain what they are used for, otherwise why they are not needed. [5%]

-
2. Assume that Scheme works exactly the same way as the interpreters in the course. Then this Scheme expression evaluates to a closure:

```
(let ((x 1) (y 2))  
  (lambda (x z) (+ x (* y z))))
```

- (a) What are the formals, the body and the environment of the closure? [10%]
(b) Assume the expression is called `E`. What is the value of the following expression:

```
(let ((x 70))  
  (E 3 5))
```

Justify your answer by explaining in which environment `(+ x (* y z))` is evaluated. Point out which bindings are relevant and which are not. [15%]

-
3. (a) Consider the following two expressions:

```
((let ((x 1) (y 2))  
  (+ x y))
```

and

```
((lambda (x y) (+ x y))  
  1 2)
```

Why are these expressions equivalent? Explain in terms of environments. [10%]

- (b) Explain why `let` bindings can always be rewritten using `lambda`. [10%]
- (c) Can one also rewrite assignments (`set!`) in the same way, replacing them with `lambdas`? Argue whether that can or cannot work. [5%]

-
4. Consider this procedure in Scheme (the `or` is just logical “or”):

```
(define fib  
  (lambda (n)  
    (if (or (= n 0) (= n 1))  
        1  
        (+ (fib (- n 1)) (fib (- n 2))))))
```

- (a) Write a version of `fib` in Continuation Passing Style. [15%]
- (b) Note that it does not make sense to call `fib` with a negative number. Modify your CPS version of `fib` so that it takes *two* continuations: the second one should be used as an emergency exit if `n` is negative. (Numbers can be compared using `<` in Scheme.) [10%]
-