CS 565 March 18, 1998
Jens Palsberg

Midterm Exam

Submit your solutions on separate paper (don't submit this handout). Write your name and id number on the top of the first page you submit. I will stable your pages together. This exam is intended to take you 2 hours. Budget your time wisely. There are 4 questions on the exam. Each question is worth 5 points, for a total of 20 points maximum. Write the solutions for different questions on different sheets of paper. Make clear what problem each solution is for. I have spare paper, just ask for it.

1. [**CPS**] Here is a Scheme procedure `zip`. It takes two flat lists of integers as arguments (like `(1 2 3)`, but not `(1 (2 3) 4)`), and it produces as result a flat list of integers.

```
(define zip
  (lambda (l m)
    (cond
      ((null? l) m)
      ((null? m) l)
      ((< (car l) (car m)) (cons (car l) (zip (cdr l) m)))
      (else (cons (car m) (zip l (cdr m)))))))
```

Transform `zip` to a program with these four properties:

- all calls to programmer-defined functions are in tail position,
- all occurrences of `lambda` are in definitions at the top level,
- all data are either booleans, integers, or flat lists of integers, and
- all programmer-defined functions take no arguments.

Show the details of how you do it.

2. [**Interpreter**] Below is the grammar for a subset of Java where the type annotations have been removed. In a class, the methods have different names. Write in Scheme an interpreter for this language.

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Decl}_1 \ldots \mathit{Decl}_n \\
\mathit{Decl} & ::= & \texttt{class}\ \mathit{Identifier}\ \{\ \mathit{Method}_1 \ldots \mathit{Method}_n\ \} \\
\mathit{Method} & ::= & \texttt{public}\ \mathit{Identifier}\ \texttt{(}\ \mathit{Identifier}\ \texttt{)}\ \{\ \texttt{return}\ \mathit{Expression}\ \texttt{;}\ \} \\
\mathit{Expression} & ::= & \texttt{this} \\
 & | & \texttt{new}\ \mathit{Identifier}\ \texttt{()} \\
 & | & \texttt{(}\ \mathit{Expression}\ \texttt{.}\ \mathit{Identifier}\ \texttt{(}\ \mathit{Expression}\ \texttt{)}\ \texttt{)} \\
 & | & \texttt{(}\ \mathit{Expression}\ \texttt{instanceof}\ \mathit{Identifier}\ \texttt{)} \\
 & | & \mathit{Identifier}
\end{array}
$$

Use the following definitions to represent nodes in syntax trees.

```
(define-record Program (List EOFToken))
(define-record Decl (Token1 Identifier Token2 List Token3))
(define-record Method (Token1 Identifier1 Token2
                 Identifier2 Token3 Token4 Token5
                 Expression Token6 Token7))
(define-record Identifier (Token))
(define-record ThisExp (Token))
(define-record NewExp (Token1 Identifier Token2 Token3))
(define-record SendExp (Token1 Expression1 Token2
                 Identifier Token3 Expression2 Token4 Token5))
(define-record InstanceofExp (Token1 Expression Token2
                 Identifier Token3))
```

Here is an example program:

```
class C {
  public m(x) {
    return ( new D().p(x) );
  }
}

class D {
  public p(y) {
    return ( y instanceof C );
  }
}
```

Here is a representation of the example program as a syntax tree.

```
(define root '(Program
  ((Decl "class" (Identifier "C") "{" (
     (Method "public" (Identifier "m") "(" (Identifier "x") ")"
      "{" "return" (SendExp "("
            (NewExp "new" (Identifier "D") "(" ")")
             "." (Identifier "p") "(" (Identifier "x") ")" ")")
          ";"
      "}"))
   "}")
   (Decl "class" (Identifier "D") "{" (
     (Method "public" ( Identifier "p") "(" (Identifier "y") ")"
      "{" "return" (InstanceofExp "("
            (Identifier "y") "instanceof" (Identifier "C") ")")
          ";"
      "}"))
   "}"))
""))
```

3. [**Beta-reduction**] Consider this Scheme definition:

```
(define i
  ((lambda (g) ((g g) (g (lambda (y) (y 0)))))
   (lambda (f) (f (lambda (x) x)))))
```

What is produced by the following two expressions? Justify your answer by showing a step-by-step computation.

```
> i
> (i 2)
```

4. [**Visitors**] Here is the grammar for a subset of Scheme. Imagine that you have a javacc input file for this language, and that you run JTB on that file.

Program	::=	*Exp*
Exp	::=	*VariableReference* | *Lambda* | *Apply*
VariableReference	::=	*Identifier*
Lambda	::=	`(lambda` (*Identifier*) *Exp*)
Apply	::=	(*Exp* *Exp*)

We will say that a lambda-abstraction `(lambda (x) exp)` is *strict* if `x` occurs free in `exp` at least once. Examples:

```
(lambda (x) x)    ;; strict
(lambda (x) y)    ;; not strict
(lambda (x) (x x)) ;; strict
(lambda (x) (x y)) ;; strict
(lambda (x) (z (lambda (y) y))) ;; not strict
```

Write in Java one or more visitors which print the number of occurrences of strict lambda-abstractions in a given expression. For example, for the above five expressions, it should print 1, 0, 1, 1, 1. The last of these numbers is 1 because there is an occurrence `(lambda (y) y)` which is strict. For the expression

```
(((lambda (x) (x y)) (lambda (z) z)) (lambda (x) z))
```

your program should print 2 because we have that `(lambda (x) (x y)` and `(lambda (z) z)` are occurrences of strict lambda-abstractions in the expression.