CS 565 October 15, 1997
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**] The Fibonacci series of numbers $f_0, f_1, \ldots$ is defined such that

$$\begin{array}{rcll} f_0 & = & 1 & \\ f_1 & = & 1 & \\ f_n & = & f_{n-1} + f_{n-2} & \quad \text{where } n > 1. \end{array}$$

Write a Scheme function `fib` of one argument n which computes f_n. Convert `fib` to tail-form, then to first-order form, then to imperative form. The resulting program must have 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 structures are either booleans, integers, or lists of integers, and
- all programmer-defined functions take no arguments.

2. [**Interpreter**] Below is the grammar for a toy programming language. Write in Scheme an interpreter for that language.

Program	::=	*Block*
Block	::=	$Decl_1 \ldots Decl_n$ *Statement*
Decl	::=	`var` *VariableName*
	|	`procedure` *ProcedureName Block* `end`
Statement	::=	`begin` $Statement_1 \ldots Statement_n$ `end`
	|	`while` *Expression* `!= 0 do` *Statement* `od`
	|	*VariableName* `:=` *Expression*
	|	`call` *ProcedureName*
	|	`print` *Expression*
Expression	::=	*VariableName* `+` *Expression*
	|	*VariableName* `-` *Expression*
	|	*VariableName* `*` *Expression*
	|	*VariableName*
	|	`0`
	|	`1`

A *Block* is like `let*` in Scheme. The initial value of variables is 0.

Use the following definitions to represent nodes in syntax trees.

```
(define-record Program (block))
(define-record Block (decl-list statement))
(define-record VarDecl (id))
(define-record ProcedureDecl (id block))
(define-record StatementList (statement-list))
(define-record WhileStatement (expression statement))
(define-record AssignmentStatement (id expression))
(define-record CallStatement (id))
(define-record PrintStatement (expression))
(define-record PlusExpression (id expression))
(define-record MinusExpression (id expression))
(define-record MultiplicationExpression (id expression))
(define-record Identifier (id))
(define-record Zer ())
(define-record One ())
```

Here is an example program which computes the factorial of `n`.

```
var n
var a
begin
  n:=1
  a:=1
  while n != 0 do
    begin
      a:=a*n
      n:=n-1
    end
  od
  print a
end
```

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

```
(define program '(Program
  (Block
    (VarDecl (Identifier n))
    (VarDecl (Identifier a))
    (StatementList
      ((AssignmentStatement (Identifier n) (One ))
       (AssignmentStatement (Identifier a) (One ))
       (WhileStatement (Identifier n)
         (StatementList
           ((AssignmentStatement (Identifier a)
              (MultiplicationExpression
                (Identifier a) (Identifier n)))
            (AssignmentStatement (Identifier n)
              (MinusExpression (Identifier n)(One ))))))
       (PrintStatement (Identifier a)))))))
```

3. [**Beta-reduction**] Consider these Scheme definitions:

```
(define bb
  (lambda (x) (lambda (y) (lambda(z) ((x y) z)))))

(define tt
  (lambda (x) (lambda (y)
    x)))

(define ff
  (lambda (x) (lambda (y)
    y)))

(define oo
  (lambda (x) (lambda (y)
    (((bb x) y) ff))))

(define i
  (lambda (x)
    (((((bb ((oo tt) ff)) tt) ff) x) (lambda (z) (z x)))))
```

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

```
> (i 2)
> ((i 2) (lambda (x) x))
```

4. [**Visitors**] Here is the grammar for a subset of Scheme:

$$
\begin{array}{lcl}
\mathit{Exp} & ::= & \mathit{VariableReference} \\
 & | & \texttt{(lambda (} \ \mathit{Variable} \ \texttt{)} \ \mathit{Exp} \ \texttt{)} \\
 & | & \texttt{(} \ \mathit{Exp} \ \mathit{Exp} \ \texttt{)} \\
 & | & \texttt{0} \\
 & | & \texttt{(+} \ \mathit{Exp} \ \texttt{1)}
\end{array}
$$

Write the grammar part (not the lexical part) of a javacc input file for this language. Imagine that you run jtb on that input file. Write in Java a visitor which prints all occurrences of free variables.