CS 565 Out: September 18, 1997
Jens Palsberg Due: September 25, 1997

Homework 4:

Implement a Translator of Simple Scheme programs into Java

Here is the grammar for a subset of Scheme.

Program	::=	$Def_1 \ldots Def_n$
Def	::=	*FunctionDef* | *VariableDef*
FunctionDef	::=	`(define` *FunctionName* `(lambda () Exp ))`
VariableDef	::=	`(define` *VariableName* `'*integer*)`
	|	`(define` *VariableName* `'*boolean*)`
	|	`(define` *VariableName* `'*intlist*)`
Exp	::=	*VariableReference*
	|	`(` *FunctionName* `)`
	|	*Constant*
	|	`(` *UnaryOp Exp* `)`
	|	`(` *BinaryOp Exp Exp* `)`
	|	`(if` *Exp Exp Exp* `)`
	|	`(set!` *Variable Exp* `)`
	|	`(begin` $Exp_1 \ldots Exp_n$ `)`
Constant	::=	`#t` | `#f` | *IntegerNumeral* | `'()`
UnaryOp	::=	`zero?` | `null?` | `car` | `cdr`
BinaryOp	::=	`+` | `-` | `*` | `cons`

Two example programs are given below.

1. Write a JavaCC grammar for the above language. Use JTB and JavaCC to generate a parser, syntax-tree-node classes, and a default visitor. Write one or more visitors which translate programs in the above language into Java. You can assume that variables with initial value `'*integer*` are only used for storing integers. Similarly, variables with initial value `'*boolean*` are only used for storing booleans, and variables with initial value `'*intlist*` are only used for storing flat lists of integers. Moreover, you can assume that all functions return an integer.

For example, here is the factorial function written in the above language. It takes it input via the `input` variable.

```
(define input '*integer*)
(define n '*integer*)
(define a '*integer*)

(define fact-iter
  (lambda ()
    (begin
      (set! n input)
      (set! a 1)
      (aux))))

(define aux
  (lambda ()
    (if (zero? n)
        a
        (begin
          (set! a (* a n))
          (set! n (- n 1))
          (aux)))))
```

Here is the append function written in the above language. It appends the contensts of the variables `l` and `m` and places the result in the variable `res`.

```
(define l '*intlist*)
(define m '*intlist*)
(define res '*intlist*)
(define k '*intlist*)
(define v '*intlist*)

(define append-iter
  (lambda ()
    (begin
      (set! k '())
      (append-iter-aux))))

(define append-iter-aux
  (lambda ()
    (if (null? l)
        (begin
          (set! v m)
          (apply-cont))
        (begin
          (set! k (cons (car l) k))
          (set! l (cdr l))
          (append-iter-aux)))))

(define apply-cont
  (lambda ()
    (if (null? k)
        (begin
          (set! res v)
          0)
        (begin
          (set! v (cons (car k) v))
          (set! k (cdr k))
          (apply-cont)))))
```