Problem Set 6:

Problem 1

Consider the following BNF specification:

\[<\text{LambdaExp}> ::= <\text{Identifier}>\]

\[<\text{Number}> ::= \text{"lambda" }<\text{Identifier}> \text{." }<\text{LambdaExp}>\]

\[<\text{LambdaExp}> ::= ( <\text{LambdaExp}> <\text{LambdaExp}> )\]

where \(<\text{Identifier}>, \text{<Number>, "lambda", ",", "(", and ")" are terminal symbols.}\n
- Using the define-datatype, define the type LambdaExp according to the specification given above.
- Define a scanner specification ScannerP1 for the given BNF-specification.
- Define a grammar specification GrammarP1 for the given BNF-specification.
- Construct a parser, named LambdaParser, using the SLLGEN procedure sllgen:make-string-parser.
Solution:

(define-datatype LambdaExp LambdaExp?
  (variable
   (is symbol?))
  (constant
   (num number?))
  (abstraction
   (id symbol?)
   (body LambdaExp?))
  (application
   (ftn LambdaExp?)
   (arg LambdaExp?))
)

(define ScannerP1
  '( (white-sp
      (whitespace) skip)
    (identifier
     (letter (arbno (or letter digit "?"))) symbol)
    (number
     (digit (arbno digit)) number)
  )
)

(define GrammarP1
  '( (LambdaExp
      (identifier) variable)
    (LambdaExp
     (number) constant)
    (LambdaExp
     ("lambda" identifier "." LambdaExp) abstraction)
    (LambdaExp
     ("(" LambdaExp LambdaExp ")") application)
  )
)

(define LambdaParser
  (sllgen:make-string-parser ScannerP1 GrammarP1))

Problem 2

First Interpreter:

<program> ::= <expression>
   a-program (expr)

<expression> ::= "<integer" "value" "=" <number> ">"
   lit-exp (num)
 ::= "<reference" "value" "=" <identifier> ">"
   var-exp (id)
 ::= "<" <prim-op> "<arguments" {<expression>}* ">" ">"
   primapp-exp (prim rands)

<prim-op> ::= "add" | "sub" | "mul" | "inc" | "dec"
   add-prim | sub-prim | mul-prim | inc-prim | dec-prim

Reconstruct the first interpreter that has been presented in class.

Solution: 10
Problem 3

Start with "first-interpreter.scm". Add list expressions to the defined language. Use the following BNF-specification to extend the interpreter:

\[
\text{<Expression>} ::= \text{"<list" \{<Expressions>\}* "/>"}
\]

\[
\text{list-exp (exprs)}
\]

\[
\text{<Expression>} ::= \text{"<unpack" <Pattern> <Expression> "/>"}
\]

\[
\text{unpack-exp (pat exp)}
\]

\[
\text{<Pattern>} ::= \text{"<pattern" <Variables> <Expression> "/>"}
\]

\[
\text{pattern-exp (vars exp)}
\]

\[
\text{<Variables>} ::= \text{"<variables"}
\]

\[
\text{\{"<variable" "value" "=" <Identifier> "/>"\}* "/>"}
\]

\[
\text{variable-list (ids)}
\]

- Add the two new rules to the grammar specification grammar-spec.
- Extend the data type <Expression> with the two new variant tags.
- Define the required abstract data types for <Pattern> and <Variables>.
- Extend the procedure eval-expression to handle the two new constructs.
Solution:

Grammar extension:

```
(define grammar-spec
  `(...
    (expression
      ("<list" (arbno expression) "/") list-exp)
    (expression
      ("<unpack" pattern expression "/") unpack-exp)
    (pattern
      ("<pattern" variables expression "/") pattern-exp)
    (variables
      ("<variables"
        (arbno "<variable" "value" "=" identifier "/")"
        "/")) variable-list)
    ...
  )
)
```

Abstract syntax extension:

```
(define-datatype expression expression?
  ...
  (list-exp
    (exprs (list-of expression?)))
  (unpack-exp
    (pat pattern?)
    (exp expression?))
  ...
)

(define-datatype pattern pattern?
  (pattern-exp
    (vars variables?)
    (exp expression?))
)

(define-datatype variables variables?
  (variable-list
    (ids (list-of symbol?)))
)
```

Eval-expression extension:

```
(define eval-expression
  (lambda (exp env)
    (cases expression exp
      ...
      (list-exp (exprs)
        (eval-rands exprs env))
      (unpack-exp (pat exp)
        (let ((new-env (eval-pattern pat env))
          (eval-expression exp new-env)))
        ...
      )
    ))
```
(define eval-pattern
  (lambda (pat env)
    (cases pattern pat
      (pattern-exp (vars exp)
        (let ((le (eval-expression exp env))
              (ids (cases variables vars
                        (variable-list (ids) ids))))
          (if (list? le)
              (if (equal? (length ids) (length le))
                  (extend-env ids le env)
                  (error "arity mismatch")
              (error "value is not a list")
          ))))))

Total: 30+10+50 = 70
Example:
The evaluation of the following program should return the value 6:

```
<unpack
  <pattern
    <variables
      <variable value = x/>
      <variable value = y/>
      <variable value = z/> />
  <list
    <integer value = 1/>
    <integer value = 2/>
    <integer value = 3/> />
  <add
    <arguments
      <reference name = x/>>
    <add
      <arguments
        <reference name = y/>
        <reference name = z/> />
```

The expression `<list ... />` creates a list expression, whereas `<unpack ... />` extracts the elements of a list expression. That is, unpack binds `x` to `1`, `y` to `2`, and `z` to `3` in the body of the `<unpack-expression>`. The expression right to `<pattern>` must be a list value. Furthermore, the number of elements in the list expression has to be the same as the number of variable defined left to `<pattern>`. If one of these conditions is violated, the procedure `eval-expression` has to report an error. (e.g. `(error "value is not a list")` or `(error "arity mismatch")`).

Submission deadline: Thursday, March 29, 2007, 2:10 p.m.
Submission procedure: on paper in class.