Problem Set 6:

Problem 1

Consider the following BNF specification:

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

variable (id)

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

abstraction (id body)

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

application (ftn arg)

where <Identifier>, <Number>, "lambda", ".", "(" and ")" are terminal symbols.

- 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.
Problem 2

First Interpreter:

<program> ::= <expression>

<expression> ::= "<integer" "value" "=" <number> "/""

::= "<reference" "value" "=" <identifier> "/""

::= "<" <prim-op> "<arguments" {<expression>]* "="/" "/""

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

Reconstruct the first interpreter that has been presented in class.
Problem 3

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

```plaintext
<Expression> ::= "<list" {<Expressions>}* ">"

list-exp (exprs)

<Expression> ::= "<unpack" <Pattern> <Expression> ">"

unpack-exp (pat exp)

<Pattern> ::= "<pattern" <Variables> <Expression> ">"

pattern-exp (vars exp)

<Variables> ::= "<variables"

{"<variable" "value" ":=" <Identifier> ">"}*

"/>"

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.
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 s 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.