COMPUTER SCIENCE 320 (FALL TERM, 2005)
CONCEPTS OF PROGRAMMING LANGUAGES

Problem Set 5: A Small Interpreter (Part I)

OUT: FRIDAY, FEBRUARY 17, 2006
DUE: ***4:59 PM*** ON FRIDAY, FEBRUARY 24, 2006

There are 5 problems in this set, each worth as indicated. The harder problems are marked with a single $*$ (average difficulty) or two $**$ (higher-than-average difficulty). For the easy points, start with the unmarked problems.

Problems 4 and 5 will be part of Problem Set 6, but you can start working on these two as soon as you finish with the earlier problems.

For this assignment you will need to understand how the syntax of the programming language INTEX is defined and how programs written in INTEX are supposed to be interpreted. Most of the things you have done already in previous problem sets (e.g., higher-order functions, operations on lists, abstract data types, etc.) will help you greatly in doing the work for this assignment.

1 Getting Familiar With INTEX

We provide you with several files containing Scheme programs, in two directories, `Scheme-utilities` and `Scheme-code-for-INTEX`. Some of these files, which you will need to work on this assignment, are:

- `abstract-syntax.scm`, which defines the necessary procedures for a syntactic analysis of INTEX programs.
- `list-ops.scm`, which defines several useful operations on lists, most of which you have already used or programmed in earlier problem sets. `list-ops.scm` is automatically loaded by `abstract-syntax.scm`.
- `subst-eval.scm`, which defines procedures to evaluate INTEX programs using the "substitution model".
- `subst.scm`, which defines procedures to handle "substitutions". `subst.scm` is automatically loaded by `subst-eval.scm`.
- `exception.scm`, which defines procedures to handle "exceptions" and better error-reporting.
- `name-not-in.scm`, which defines procedure `name-not-in`, useful for Problem 3.
- `set-list-nodups.scm`, which is an implementation of the ADT of "sets" in terms of lists without duplicates. None of the functions in this package are used in the other packages, except for the function `set-member?` which is used in `name-not-in.scm`. This package is automatically loaded by `name-not-in.scm`.

For easy downloading, we made a "tar" file of each of the two directories. To restore the original directories and their contents on your local machine, issue the following Unix commands:

```
tar -xvf Scheme-utilities.tar
tar -xvf Scheme-code-for-INTEX.tar
```

To run and test some of the programs we provide you, we suggest that you open an Emacs window under the directory `Scheme-code-for-INTEX`. After launching your Scheme interpreter, within the Emacs window, you can experiment with the INTEX interpreter by executing the following Scheme code:

```
(load "load-intex")

;; Run the averaging program on the inputs 5 and 8
;; using the substitution model
(subst-run '(program (a b) (div (+ a b) 2)) '(5 8))

;; Perform a substitution in an expression
(subst (env-make '(a c) '(3 5)) '(div (+ a b) c))

;; Perform a substitution in an expression
(subst (env-make (list 'a 'c) (list 3 5)) '(div (+ a b) c))

;; Perform a substitution in an expression
(subst (env-make (list a c) (list 3 5)) '(div (+ a b) c))

;; Perform a substitution in an expression
(subst (env-make (list 'a 'c) (list 3 5)) (list (div (+ a b) c)) )

;; Perform a substitution in an expression
(subst (env-make (list 'a 'c) (list '(+ x y) '(+ x y))) '(div (+ a b) c))
```

The third and fourth substitutions above should return an error. Make sure you understand the reason. Because INTEX programs are represented as s-expressions, they can be named so that they are easily reusable. Next, execute the following commands:

```
(define avg '(program (a b)  (div (+ a b) 2)) )

(subst-run avg '(3 8))

(subst-run avg '(20 10))
```

When writing Scheme programs that manipulate INTEX programs you should only use the abstract syntax operators. Next, execute:

```
(program-formals avg)

(define body (program-body avg))

(binapp? body)
```

2 Extending INTEX With Sigma

We want to extend the INTEX interpreter to implement a new summation construct of the form:

($\mathbf{sigma}$ var lo hi $body$)

The first argument var must always be a variable name, the second and third arguments lo and hi must always be integers, and the fourth argument $body$ is an arbitrary valid expression. The interpretation of a **sigma** expression of the form above returns the sum of $body$ evaluated at all values of the index variable var ranging from lo to hi, inclusive. In more conventional mathematical notation, this sum is written as:

$$\sum_{var=lo}^{hi} body$$

If the value of lo is greater than the value of hi, the sum is 0.

Here are some examples of **sigma** in action:

```
(sigma k 1 5 k)
; 1 + 2 + 3 + 4 + 5 = 15
```

Renaming the index variable `k` to, say `m`, does not change the final result. This `k` is an instance of a "bound variable". Renaming bound variables does not change the meaning of programs.

```
(sigma m 1 5 m)
; 1 + 2 + 3 + 4 + 5 = 15
```

More interesting examples follow:

```
(sigma k 3 5 (* k k))
; 3*3 + 4*4 + 5*5 = 50

(sigma k 5 3 (* k k))
; evaluates to 0

(sigma i 2 3
   (sigma j 3 5 (* i j)))
; (2*3) + (2*4) + (2*5) + (3*3) + (3*4) + (3*5) = 60
```

The `sigma` construct can be manipulated via the following abstract syntax operations:

($\mathbf{make\text{-}sigma}$ var lo hi $body$)
which returns a `sigma` node whose index variable is var, whose lower bound integer is lo, whose upper bound integer is hi, and whose body expression is $body$.

($\mathbf{sigma\text{-}var}$ $sigma\text{-}node$)
which returns the index variable of $sigma\text{-}node$.

($\mathbf{sigma\text{-}lo}$ $sigma\text{-}node$)
which returns the lower bound of $sigma\text{-}node$.

(**sigma-hi** *sigma-node*)
which returns the upper bound of *sigma-node*.

(**sigma-body** *sigma-node*)
which returns the body of *sigma-node*.

(**sigma**? *node*)
which returns `#t` if *node* is a `sigma` node, and `#f` otherwise.

Problem 1 *(30 points)* Write the Scheme code for each of the 6 operations `make-sigma`, `sigma-var`, `sigma-lo`, `sigma-hi`, `sigma-body` and `sigma?`. These are needed to extend the package `abstract-syntax.scm`.

Integrate these 6 functions into `abstract-syntax.scm` to obtain a new package for the syntactic analysis of programs in INTEX augmented with `sigma`. Call the new package `abstract-syntax-1.scm`.

* **Problem 2** *(40 points)* Write the Scheme code of 3 functions: `free-vars`, `binding-vars` and `bound-vars` which, given a valid expression *exp* (not the text of an entire program) in INTEX augmented with `sigma`, will each return a set (implemented as a list) respectively of:

- all variables which have a free occurrence,
- all variables which have a binding occurrence, and
- all variables which have a bound occurrence,

in the given expression *exp*. In general, (`bound-vars` *exp*) will be a subset, not necessarily proper, of (`binding-vars` *exp*). On the other hand, there is no special relationship between (`free-vars` *exp*) and the other two sets of variables, i.e., they may or may not overlap.

The presence of bound variables in expressions containing `sigma` makes the substitution operation a little more delicate. In general, given an expression *exp1* containing occcurrences of variable *var* and another expression *exp2*, in order to substitute *exp2* for *var* in *exp1* we need to make sure that two conditions are satisfied:

1. We substitute *exp2* for the free occurrences of *var* only, not the binding occurrences nor the bound occurrences of *var*, in *exp1*.
2. We do this substitution provided no free variable occurrence in *exp2* is "captured" by a binding variable occurrence in *exp1*.

To illustrate these two requirements, consider the following example:

exp1 = `(+ a (sigma a 1 3 (* a b)))`
exp2 = `(+ a 5)`
var = `a`

Variable a has 3 occurrences in *exp1*: the first is a *free occurrence*, the second is a *binding occurrence* and the third is a *bound occurrence*. Hence, according to the first requirement, we can substitute `(+ a 5)` for the first occurrence of a only, but not for the second and third occurrences. The result of this substitution is:

```
(+ (+ a 5) (sigma a 1 3 (* a b)))
```

Suppose now we change *var* to b. There is only one occurrence of b in *exp1*, which is free. If we substitute `(+ a 5)` for b, we do not violate the first requirement, but we do violate the second requirement; indeed, the result of such a substitution is:

```
(+ a (sigma a 1 3 (* a (+ a 5))))
```

where the fourth occurrence of variable a has been "captured" by the binding occurrence of a.

The two requirements above have to do with preventing potential interference between free and bound variable occurrences. One simple way of satisfying both requirements is to first "rename" all variables that have binding occurrences in *exp1* so that there is no overlap between variables in (`binding-vars` *exp1*) and all the variables in (`free-vars` *exp1*) and (`free-vars` *exp2*), i.e.,

(`binding-vars` *exp1*) $\cap$ ((`free-vars` *exp1*) $\cup$ (`free-vars` *exp2*)) = $\varnothing$

For example, in our example above, we can rename every variable that has binding occurrences in *exp1*, in this case a, to a fresh variable, say n, to obtain:

```
(+ a (sigma n 1 3 (* n b)))
```

We can now substitute `(+ a 5)` for a (or b) without fear of breaking the intended meaning.

** **Problem 3** *(30 points)* Write the Scheme code of a function `rename` which takes two arguments: (1) a valid expression *exp* (not the text of an entire program) in INTEX augmented with `sigma`, and (2) a set (implemented as a list) of variables, called `reserved-var-names`. Executing (`rename` *exp* `reserved-var-names`) will first test whether (`free-vars` *exp*) is a subset of `reserved-var-names`, and if this test succeeds, it will return a new expression *exp'* which is identical to *exp* except that the name of every binding variable occurrence in *exp* is changed to a fresh variable name *not* in `reserved-var-names`.

For Problem 3, we want you to program the function `rename` using a help function `name-not-in` of two arguments which, given a name `'a` and the set `reserved-var-names`, returns the first subscripted version of `'a` not in `reserved-var-names`. So, `name-not-in` will generate in turn `'a_0, 'a_1, 'a_2, 'a_3, ...`, until it finds one which is not in the set `reserved-var-names`. The first challenge is how to implement the function `name-not-in`. We help you by providing the code for `name-not-in`. The second challenge is how to implement `rename` using `name-not-in`, which we leave to you.

Problem 4 *(Hand in with Problem Set 6)* Extend the definition of `subst` in `subst.scm` to correctly perform substitutions into valid expressions in INTEX augmented with `sigma`. Call the new package `subst-1.scm`.

Problem 5 *(Hand in with Problem Set 6)* Extend the definition of `subst-eval` in `subst-eval.scm` to correctly evaluate valid expressions in INTEX augmented with `sigma`. Call the new package `subst-eval-1.scm`.