Spring 2004 Name: ______________________
Com S 342 Section: ______________________

Principles of Programming Languages

Exam 2 on Induction, Recursion, and Grammars

This test has 9 questions and pages numbered 1 through 10.

Special Instructions for this Test

Your code should type check for full credit. You may not use `trustme!` or `has-type-trusted` in your code. You are not permitted to use the procedures named `parse-`.

Reminders

For this test, you can use one (1) page (8.5 by 11 inches, one (1) side, no less than 9pt font) of notes. Handwriting is okay. No photo-reduction is permitted. Don't use anything with printing on the other side, please. These notes are to be handed in at the end of the test. Have your name in the top right corner. Use of other notes or failure to follow these instructions will be considered cheating.

If you need more space, use the back of a page. Note when you do that on the front.

This test is timed. We will not grade your test if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire test so that you can budget your time.

For programs, indentation is important to us for "clarity" points; if your code is sloppy or hard to read, you will lose points. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

You can use helping procedures whenever you like. If you write recursive helping procedures, please give a deftype declaration for them.

For Grading

Problem	Points
1	
2	
3	
4	
5	
6	
7	
8	
9	

1. (5 points) Write a curried version of the following procedure, including a `deftype` declaration for it.

```
(deftype present-award
  (-> ((list-of symbol) (list-of symbol) (list-of symbol)) (list-of-symbol)))
(define present-award
  (lambda (preamble category winner)
    (append preamble '(and the winner for) category '(is) winner)))
```

2. (10 points) Consider the following grammar.

   ⟨sym-exp⟩ ::= ⟨symbol⟩ | ⟨s-list⟩
   ⟨s-list⟩ ::= `(` {⟨sym-exp⟩}* `)`

   where ⟨symbol⟩ stands for a Scheme symbol, such as `x`. Now consider the following string.

```
( x ( ) )
```

   Either show how to derive the above string from the nonterminal ⟨sym-exp⟩, using the given grammar, or briefly explain why no derivation is possible.

   Please show all steps, and don't replace more than one nonterminal in a step.

3. (5 points) Write "yes" underneath each of the following that is a true statements about curried procedures. Write "no" underneath each of the following that is false.

   (a) Curried procedures are "spicier" than uncurried procedures, which is why they have to be taken with a grain of salt.

   (b) A curried version of a procedure contains a nest of `lambda`s, each of which takes exactly one argument.

   (c) A curried procedure can be considered a "tool maker", because when it is given an argument, it returns a procedure.

   (d) For each procedure, p, p's type is exactly the same as the type of the curried version of p.

   (e) Curried procedures are difficult to write in C++, because C++ doesn't provide automatic support for closures.

4. (10 points) Write "yes" underneath each of the following that is a true statements about syntactic sugars (syntax abstractions). Write "no" underneath each of the following that is false.

   (a) The `&&` and `||` forms in C++ are syntactic sugars for C++ if-expressions (`? :`).

   (b) Scheme's `let` form is a syntactic sugar for the application of a lambda-expression.

   (c) A syntactic sugar helps users of the language by making the language "sweeter" or easier to use for common patterns.

   (d) The Java programming language has no syntactic sugars.

   (e) Syntactic sugars can sometimes provide extra information to a compiler that can be used in optimizations.

5. (10 points) Write a Scheme procedure,

   ```
   (deftype leave-in
      (forall (T) (-> ((-> (T) boolean) (list-of T)) (list-of T))))
   ```

   that takes a predicate, `pred`, and a list, `lst`, and returns a list that is just like `lst`, except that it only contains the elements in `lst` for which `pred` returns `#t`. The following are examples.

   ```
   (leave-in odd? '()) ==> ()
   (leave-in odd? '(17 24 33 5 3 226 942)) ==> (17 33 5 3)
   (leave-in even? '(17 24 33 5 3 226 942)) ==> (24 226 942)
   (leave-in (lambda (x) (eq? x 'oscar)) '(oscar wins an oscar too))
       ==> (oscar oscar)
   (leave-in (lambda (x) (not (eq? x 'oscar))) '(an oscar too))
       ==> (an too)
   ```

6. (10 points) Write a Scheme procedure,

   ```
   (deftype has-plus? (-> ((list-of sym-exp)) boolean))
   ```

   that takes a list of ⟨sym-exp⟩s, `slst`, and returns `#t` just when `slst` contains the symbol `+`. Your answer must use the helpers for the ⟨sym-exp⟩ grammar, whose types are given on the next page, and must type check for full credit. If you write a helping procedure (hint, hint), be sure to give a `deftype` for it.

   ```
   (has-plus? (parse-s-list '())) ==> #f
   (has-plus? (parse-s-list '(x + (y + z)))) ==> #t
   (has-plus? (parse-s-list '(x + y))) ==> #t
   (has-plus? (parse-s-list '(x (+ y () ((z)))))) ==> #t
   (has-plus? (parse-s-list '(x ((y)) ((((q)))) z a b))) ==> #f
   (has-plus? (parse-s-list '(oscar (((((((() () x + y () ()))))))))) ==> #t
   (has-plus? (parse-s-list '(((lord of (the) rings () ()))))) ==> #f
   ```

The following are the types of the ⟨sym-exp⟩ and s-list helpers, from the file `$PUB/lib/sym-exp.scm` that you can use in the surrounding problems.

```
(deftype symbol->sym-exp (-> (symbol) sym-exp))
(deftype s-list->sym-exp (-> ((list-of sym-exp)) sym-exp))

(deftype sym-exp-symbol? (-> (sym-exp) boolean))
(deftype sym-exp-s-list? (-> (sym-exp) boolean))

(deftype sym-exp->symbol (-> (sym-exp) symbol))
(deftype sym-exp->s-list (-> (sym-exp) (list-of sym-exp)))
```

7. (20 points) Write a Scheme procedure,

```
(deftype duplicate (-> (symbol sym-exp) sym-exp))
```

that takes a symbol, `sym`, and a ⟨sym-exp⟩, `symexp`, and returns a ⟨sym-exp⟩ that is just like `symexp`, except that it has a list of two copies of `sym` each place where `sym` occurred originally occurred in `symexp`. Your answer must use the helpers for the ⟨sym-exp⟩ grammar, whose types are given on the previous page, and must type check for full credit. If you write a helping procedure, be sure to give a `deftype` for it.

```
(duplicate 'x (parse-sym-exp 'x)) ==> (x x)
(duplicate 'x (parse-sym-exp 'y)) ==> y
(duplicate 'x (parse-sym-exp '())) ==> ()
(duplicate 'y (parse-sym-exp '(x + (y + z)))) ==> (x + ((y y) + z))
(duplicate 'q (parse-sym-exp '(x + y))) ==> (x + y)
(duplicate 'q (parse-sym-exp '(x ((y)) ((((q)))) z a b)))
          ==> (x ((y)) (((((q q))))) z a b)
(duplicate 'y (parse-sym-exp '(oscar ((((((() () x + y () y ())))))) y))))
          ==> (oscar ((((((() () x + (y y) () (y y) ())))))) (y y)))
```

8. (20 points) Consider the following grammar, for the langauge lambda-1-exp, with comments on the right side enclosed in quotations (“ and ”). The comments are an aid to remembering the helping procedures, which are shown on the next page.

⟨expression⟩ ::= ⟨symbol⟩	“var-exp (id)”
| `( lambda (` ⟨identifier⟩ `)` ⟨expression⟩ `)`	“lambda-exp (id body)”
| `(` ⟨expression⟩ ⟨expression⟩ `)`	“app-exp (rator rand)”

Using the helpers whose types appear on the next page, write a procedure,

```
(deftype subst-for-free (-> (symbol symbol lambda-1-exp) lambda-1-exp))
```

that takes two symbols, `new`, `old`, and a lambda-1-exp, `exp`, and returns and a lambda-1-exp that is the same as `exp`, except that all uses of `old` as a free variable reference are replaced by `new`. Recall from class that a free variable reference x is a var-exp whose id is x and that is not enclosed in a lambda-exp whose id is also x. For example, `y` is free in the expression `(y x)`, but not in `(lambda (y) y)`. Hint: inside the body of a lambda expression of the form `(lambda (z) ...)`, there can be no free variable references named `z`.

The following are examples.

```
(subst-for-free 'y 'g (parse-lambda-1-exp 'g)) ==> y
(subst-for-free 'y 'x (parse-lambda-1-exp 'g)) ==> g
(subst-for-free 'k 'z (parse-lambda-1-exp '(f z))) ==> (f k)
(subst-for-free 'a 'z (parse-lambda-1-exp '(z (f z)))) ==> (a (f a))
(subst-for-free 'a 'z (parse-lambda-1-exp '(lambda (z) z))) ==> (lambda (z) z)
(subst-for-free 'a 'z (parse-lambda-1-exp '(lambda (x) z))) ==> (lambda (x) a)
(subst-for-free 'a 'z (parse-lambda-1-exp '((z z) (lambda (x) (z (f z))))))
                                          ==> ((a a) (lambda (x) (a (f a))))
```

The following define the types of the ⟨lambda-1-exp⟩ helpers, from the file `$PUB/lib/lambda-1-exp.scm` that you can use in the problem on the previous page.

```
(deftype var-exp? (-> (lambda-1-exp) boolean))
(deftype lambda-exp? (-> (lambda-1-exp) boolean))
(deftype app-exp? (-> (lambda-1-exp) boolean))

(deftype var-exp->id (-> (lambda-1-exp) symbol))
(deftype lambda-exp->id (-> (lambda-1-exp) symbol))
(deftype lambda-exp->body (-> (lambda-1-exp) lambda-1-exp))
(deftype app-exp->rator (-> (lambda-1-exp) lambda-1-exp))
(deftype app-exp->rand (-> (lambda-1-exp) lambda-1-exp))

(deftype var-exp (-> (symbol) lambda-1-exp))
(deftype lambda-exp (-> (symbol lambda-1-exp) lambda-1-exp))
(deftype app-exp (-> (lambda-1-exp lambda-1-exp) lambda-1-exp))
```

9. (10 points) Without using `string->list` or `list->string`, write a procedure,

```
  (deftype last-slash (-> (string) number))
```

that takes a string, `path`, and returns the 0-based index of the last slash, i.e. `#\/`, character in `path`, or -1 if no slash occurs in `path`. That is, the result r is such that, if $r \geq 0$, then the rth character in `path` is `#\/`, and for all $r < i < len$, where len is the length of `path`, the ith element of `path` is not the character `#\/`; furthermore, if r is -1, then no character of in `path` is `#\/`. The following are examples. (Recall that `(string #\a #\/ #\b) = "a/b"`.)

```
(last-slash (string #\a #\/ #\b)) ==> 1
(last-slash (string )) ==> -1
(last-slash "a/b/c") ==> 3
(last-slash "/foo/bar/baz/") ==> 12
(last-slash "/foo/bar/baz") ==> 8
(last-slash "foobarbaz") ==> -1
```

Hints: Remember that Scheme strings have indexes that start with zero (0). You can use

```
string-length : (-> (string) number)
string-ref    : (-> (string number) char)
```

to get the length and to access an element of a string, respectively. You can use `eqv?` to compare characters.