NAME____________________________GRADE_______

	short					long		
Problem	1	2	3	4	5	I	II	Σ
Max	12	12	12	12	12	26	26	100
Score								

CSc 520 **final exam** Friday 5 May 2006

TIME = 2 hours

Write all answers ON THIS EXAMINATION, and submit it IN THE ENVELOPE at the end of the exam.

Do four (4) of the ‘‘short’’ problems (1-5)

Do *both* of the ‘‘long’’ problems (I-II).

You may do an additional problem for extra credit. If you do so, **circle the extra credit problem in the table above** or else only 6 problems will be counted.

Short Answer (do 4)

1. A Fixed Point Combinator

Show that the combinator

$$\mathbf{Y} \equiv (\lambda f\ .\ ((\lambda x\ .\ f(xx))(\lambda x\ .\ f(xx))))$$

satisfies the *fixed point identity*:

$$\mathbf{Y}F = F(\mathbf{Y}F)$$

Label each step that is a beta reduction by $\rightarrow_\beta$. Label each step that is an eta reduction by $\rightarrow_\eta$. Label each step that is an alpha conversion by $\leftrightarrow_\alpha$. Label each step that represents replacing a name by its defintion (or vice versa) by $\equiv$.

For example:

$(\mathbf{SELF\ SELF}) \equiv ((\lambda x\ .\ xx)\ \mathbf{SELF}) \rightarrow_\beta (\mathbf{SELF\ SELF})$.

2. Binding

The following text is modified from R.C. Holt and J.R. Cordy, "The Turing Programming Language", *CACM 31*, 12(Dec 1988), 1415.

‘‘The next example illustrates the use of Turing’s **bind** declaration.

```
 type r:
         record
                 s: string(15)
                 n: int
         end record
var a: array 1 .. 3 of r
var y: r
var i,j: int
 . . .
i := 3
bind var x to a(i)
x.s := "C.E. Shannon"
a(i).n := 7
```

‘‘The **bind** specifies that *x* is to be another name for $a(i)$. The **var** keyword specifies that *x*, and hence $a(i)$, may be changed in the scope of *x*; for example, the last statement assigns to a field of *x*. If **var** is omitted, the declared item cannot be changed. The binding is by reference, in that changes to *i* in the scope of *x* do not affect the element to which *x* is bound.’’

(a) Give a binding diagram to show the effect of **bind var** *x* **to** $a(i)$. Show the resulting diagram just after the **bind** command. Use a box with two sub-boxes to represent a record object; arrays can be represented by a row of contiguous boxes.

(b) Add the following line to the above program, and give a binding diagram to show the result. Show the resulting diagram just after the assignment is made, and fill in all relevant values and fields.

```
y := a(i)
```

3. Parameter Transmission

The following procedure is intended to swap the contents of two integer variable arguments `x` and `y`. Assume that the language is like Pascal, except that the parameter transmission method remains to be specified. The procedure avoids the allocation of a temporary local variable by using an add/subtract trick:

```
procedure swap(x,y: integer);
   begin
      x := x + y;
      y := x - y;
      x := x - y
   end;
```

Assume that in this problem you can ignore difficulties caused by integer addition overflow.

(a) Does the procedure work correctly for **any pair** of legal actual arguments if the parameters are passed *by reference*? If so, explain why. If not, give a counter-example and explain.

(b) Same question as (a), but assume the parameters are passed *by copy* (copy-in/copy-out). If so, explain why. If not, give a counter-example and explain.

(c) Same question as (a), but assume the parameters are passed *by name*. If so, explain why. If not, give a counter-example and explain.

4. Narrowing Scope

Suppose `p` and `q` are declared as pointer variables pointing to integers in Algol 68, i.e., each is of mode **`ref ref int`**. In the language Algol 68, an assignment statement of the form

```
        p := q
```

is correct *only if* ‘‘the scope of the variable on the right-hand-side is *at least as large as* the scope of the variable on the left hand side’’.

(a) Explain briefly why the language imposes the restriction quoted above.

(b) Give an example of a program fragment where this restriction is violated, and explain what can go wrong. Do not worry about Algol 68 syntax, but explain clearly your intent. Show all declarations and carefully mark scopes.

(c) Can the problem illustrated in (b) occur if `p` and `q` are declared as simple (*non-pointer*) integer variables? That is, does it occur if they are both of mode **`ref int`**? Why or why not?

5. **`loop`** Construct

We want to add to the simplest version of *IMP* (Example 3.6, p. 66 of Watt, called IMP_0 in class) another iterative construct, which we will explain by example. If *X* is a non-negative integer variable with, say, integer value *n*, and *C* is a command, then **`loop`** *X* **`:`** *C* **`end`** has the same effect as repeatedly executing *C* a total of *n* times in sequence. In other words, it is equivalent to the command *C* **`;`** *C* **`;`** $\cdots$ **`;`** *C* ($n \geq 0$ occurrences of *C*). If *X* has a negative integer value, then we define this to have the same effect as if *X* were zero. For example, the program

```
loop X : Y:=Y+1 end ;
```

has the same effect as `Y:=Y+X;`, provided *X* has non-negative value; otherwise the effect is **`skip`**. For another example,

```
Y:=3;
loop X : Y:=6 end;
```

has the same effect as **`if`** `X>0` **`then`** `Y:=6` **`else`** `Y:=3` **`;`** .

(a) Provide the required additional syntax rules for *IMP*.

(b) Give any requred contextual constraints (beyond what the grammar will enforce.)

(c) Assume *X* is the syntactic metavariable for the syntactic category `Identifier`, and *C* is the metavariable for `Command`. Provide the following semantic rule. Note that this is direct semantics, not continuation semantics.

execute ⟦ **`loop`** *X* **`:`** *C* **`end`** ⟧ *env sto* =

Long Problems (do 2)

I. Conditional Expressions

We want to add a new kind of `Expression` to the language IMP$_g$—the original IMP, but with command and expression continuations added. The new expression is called a *conditional expression*. A conditional expression has the general form $E_1 \,\mathbf{?}\, E_2 \,\mathbf{:}\, E_3$ where the E_i are expressions. The informal meaning of this expression is that, if E_1 evaluates to **true**, then expression E_2 is evaluated (incurring any side-effects or performing any jumps encountered) and otherwise expression E_3 is evaluated (incurring any side-effects or performing any jumps encountered). The semantics is supposed to be ‘‘short-circuit evaluation’’, meaning that *only one* of the expressions E_2, E_3 will be evaluated.

Provide ‘‘**expression continuation semantics**’’ for this new syntax. Assume the following important *contextual constraint*:

- no jumps are allowed between or among each of E_1, E_2, and E_3. Thus, if a `goto` L occurs inside E_i, $i = 1, 2, 3$, then the label L`:` must occur either outside the entire expression, or else is confined to the same subexpression E_i as the `goto` L.

  The above simplification means that the auxiliary function *bind – labels* will *not be needed* to describe the semantics of this expression, and so there will be *no* recursive semantic rules.

In the questions below, *env* represents an environment (a member of **Environ**), and *econt* is an expression continuation (a member of **Econt**).

(a) (5 points). Add the appropriate rules to the grammar of IMP$_g$. Give *only* the *changes*.

(b) (5 points). Are there any further contextual constraints (static semantics) introduced by conditional expressions (beyond that in ● above)? If so, state them in English, or say ‘‘none’’ if there are none.

(c) (5 points). Do any of the semantic domains **Value**, **Storable**, **Bindable**, **Cont** and **Econt** need to be changed as a result of this extension to the language? If so, show the changes, or say ‘‘none’’.

(c) (3 points). Give the (type domain) signature for the semantic function *evaluate* `:` .

(d) (8 points). Complete the following semantic equation:

$evaluate[\![E_1 \mathbf{?} E_2 \mathbf{:} E_3]\!]\ env\ econt\ =$

(problem I solution continued, if needed)

II. Fluid Binding

Below is the abstract syntax of a purely applicative (functional) language X. The language X has static binding (lexical scoping) of function declarations. The only functions that can be defined have one argument. The only data type in X is Integer, so that a given identifier denotes either an Integer or a function (an element of Integer $\rightarrow$ Integer).

```
 Expression    ::=    Numeral   |   Expression + Expression
                |     (Expression)    |   Identifier
                |     let Declaration  in Expression
                |     Identifier ( Expression )

Declaration    ::=    Identifier = Expression
                |     fun Identifier (Identifier) = Expression
```

Static binding means that in a function declaration such as **let** f(x) **=** x+y **in** *E*, the binding of free name y is taken from the environment existing at the point of definition of the function, and not from the environment existing at the point of some call inside the body *E*.

We wish to extend X to a language $\tilde{X}$ that has ‘‘fluid binding’’. We add to the syntax the rule

```
Expression ::= bind Identifier = Expression during Expression
```

The effect of **bind** *I* **=** *E* **during** *B* is to (1) temporarily bind the value of *E* to *I*, (2) evaluate the body *B* with this altered environment, (3) then rebind the name *I* to its original value, and (4) return the value of *B* as the value of the complete **bind** · · · expression. Thus the effect is to temporarily force a "dynamic" binding of *expr* to *all* free occurrences of *I* that may be present in the *body*, whether inside a function call or not. For example, the value of the expression

```
let a = 0 in
     let fun f( x) = a in
          ( bind a= 99 during f( 2) )
```

is 99 (and *not* 0).

(a) (9 points). In the language X (and $\tilde{X}$ as well), the expression

```
let x = 4 in
     let fun p( y) = x+ y in
          ( ( let x= 7 in p( 1) ) + p( 2) )
```

has the value 11. Explain why this is so, describing the bindings in force as each subexpression is evaluated.

(b) (9 points). In the language $\tilde{X}$, what is the value of the following expression, in which the inner **let** has been replaced by a fluid binding? Explain your evaluation, showing the bindings in force as each subexpression is evaluated.

```
let x = 4 in
     let fun p( y) = x+ y in
          ( ( bind x= 7 during p( 1) ) + p( 2) )
```

(c) (8 points). What is the advantage of augmenting a statically scoped functional language to allow fluid binding? Would Scheme, for example, benefit from such a facility?

(problem II continued)