NAME____*Solutions*_____**GRADE**_____

Prob	1	2	3	4	5	I	II	III	Σ
Max	12	12	12	12	12	26	26	26	100(+ ···)
Score									

CSc 520 **final exam** Wednesday 13 December 2000

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 two (2) of the ''long'' problems (I-III).

You may do **ONE AND ONLY ONE 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. Ordinary Variables and Pointer Variables

Consider the following two program fragments:

```
x = 6;
y = 7;
foo(x);
```

Program A

```
---------
int x; int y;
```

```
*p = 6;
*q = 7;
foo(*p);
```

Program B

```
---------
int *p; int *q;
```

where the definition of the called function is:

```
void function foo( int a) {
      printf("output = %d", a);
}
```

(a) At first sight, one might conclude that the call to `foo` in both programs **A** and **B** will print the same value. This is not true. Fill in the blank in **Program B** with a statement that will cause **Program A** and **Program B** to print *different* results.

(b) Explain, in words or using diagrams, what happens in each program to cause `foo` to print different values.

Solution

(a) `p = q`

(b) `p` and `q` are *pointer aliased*, so that both assignments write to the same location. `*q = 7` makes `*p` equal to 7 and so the call is to `foo` with by-value argument 7. In Program A, this cannot happen, since separately named variables are bound to separate references (locations).

2. Parameter Transmission

Could the instructions `a += 2` and `a = a + 2` ever behave differently? This question is explored below.

Consider the following two function definitions:

Program A

```
---------
void function increment( int a){
   a += 2;
}
```

Program B

```
---------
void function assign( int a){
   a = a + 2;
}
```

Consider the possible actual arguments (*arg*) in a call to each of these functions. Could a call to `increment(`*arg*`)` have a **different** side-effect from a call to `assign(`*arg*`)`

(a) If parameter transmission uses *call by copy-in/copy-out*?

(b) If parameter transmission uses *call by value/result*?

(c) If parameter transmission uses *call by reference*?

(d) If parameter transmission uses *call by name*? In each of your answers above, if they *cannot* have a different side-effect, explain why not. If they *can* have a different side-effect, give an example and explain.

Solution

(a) No. At call time, the l-value of the actual argument is saved for copy-out, and fixed at this time, unaffected by any later side-effects. The r-value is copied into `a` at call. Local `a` is updated by two by either instruction, and this new r-value is copied back to the saved l-value. Since the effect on `a` is the same, both give same result.

(b) No. In both cases the value copied back will be incremented by 2, although in some cases the copy-back will be to a different location: Suppose `i = 1` in the caller and the actual argument is `a[i++]`. When evaluated the first time, this will give the l-value of a[1], and this is what will be passed by value; call it *v*. At return time, `a[i++]` will again be evaluated with the l-value `a[2]`; this is the location that will receive *v+2*.

(c) No. Since both instructions, applied to the same l-value will give the same result, all is well.

(d) Yes. Suppose the call is with argument `a[i++]` with `i = 1` in caller, and `a[1]` set to 10. Then `a += 2` references this *once* and so `a[1]` will get 12, with `i = 2` at return.

But `a = a + 2` be equivalent to `a[i++] = a[i++] + 2`. Since the expression is referenced *twice*, the location assigned to will be different; depending on order, this will either assign 12 to `a[2]` (different) or `a[2]+2` to `a[1]`; again different.

3. Typing in C and ML

The languages C and ML declare the types associated with names in distinct ways. The table below compares a type declaration in C with the equivalent declaration in ML.

(a) Fill in the blanks in the table below. A few examples have been filled in already to guide you.

C	**ML**
`int x`	`x : int`
`int *p`	`p : int ref`
`int **p`	_____
`int (*a)(char)`	`a : (char -> int) ref`
`int  *b(char)`	`b : (char -> int ref)`
`int d(char)`	_____
`int (*d)(int (*) (char))`	`d : ((char -> int) ref -> int) ref`
`int *e(int (*) (char))`	_____
`int f(int (*) (char))`	_____

(b) In C, a function itself is not a variable, and a function cannot be directly passed to functions or returned by functions. However, it *is* possible to define pointers to functions, which can be passed to functions or returned by functions. Some examples are in (a) above.

For the following ML type declarations, fill in the blank with the correct type declaration in C, *provided* such a definition is legal in C. If it is *not legal*, fill in ‘‘not legal’’.

C	**ML**
_____	`q : ((char -> int) ref -> int) ref`
_____	`g : ((char-> int) ref -> int)`
`int (*(h(int)))(char)`	`h : (int -> (char -> int) ref)`
_____	`j : (int ref -> (char -> int))`
_____	`k : ((char -> int) -> int)`
_____	`l : ((char -> int) ref -> (char -> int) ref )`
_____	`m : ((int -> int) ref -> (int -> int) )`

(a)

`int **p`	`__ p : int ref ref`
`int d(char)`	`__ d : char -> int`
`int *e(int (*) (char))`	`__ e : ((char -> int) ref -> int ref )`
`int f(int (*) (char))`	`__ f : ((char -> int) ref -> int )`

(b)

`_ int (*q)(int (*) (char))`	`q : ((char -> int) ref -> int) ref`
`__ int g(int (*) (char))`	`g : ((char-> int) ref -> int)`
`__ not legal`	`j : (int ref -> (char -> int))`
`__ not legal`	`k : ((char -> int) -> int)`
`__ int *l(int (*) char)(char)`	`l : ((char -> int) ref -> (char -> int) ref )`
`__ not legal`	`m : ((int -> int) ref -> (int -> int) )`

4. Conditional Expressions

We want to add a new kind of `Expression` to the language IMP (Watt, Example 3.6, page 67), called a *conditional expression*. A conditional expression has the general form e_1 **?** e_2 **:** *esub*3 where the e_i are expressions of various kinds. The informal meaning of this expression is that, if e_1 evaluates to **true**, then this expression takes on the value of e_2 and otherwise takes on the value of e_3. The semantics is supposed to be ''short-circuit evaluation'', meaning that only one of the expressions e_2, e_3 will be evaluated.

(a) Add the appropriate syntax to the grammar of IMP. Give *only* the syntax *changes*.

(b) Do any of the semantic domains **Value**, **Storable**, and **Bindable** need to change as a result of this extension to the language? If so, show the changes, or say ''none''.

(c) State any new contextual constraints (static semantics) in English, or say ''none'' if there are none.

(d) Complete the following semantic equation:

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

(e) Now suppose expression evaluation can have side-effects, so that the signature of *evaluate* is Environ $\rightarrow$ Store $\rightarrow$ Value $\times$ Store. Complete the following semantic rule for *evaluate* under this new assumption. Do not give any other semantics rules.

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

Solution

(a)

```
Expression =:: . . .
            |   Expression ?Expression :Expression
```

(b) None.

(c) E_1 has to return type boolean and E_2 and E_3 have to agree in type.

(d)

$$
\begin{array}{l}
evaluate[\![E_1 \,\mathbf{?}\, E_2 \,\mathbf{:}\, E_3]\!]env\ sto\ = \\
\quad \textbf{let}\ truth\text{-}value\ b_1\ =\ evaluate[\![E_1]\!]\ env\ sto \\
\quad \textbf{in if}\ b_1\ =\ \textbf{true} \\
\qquad \textbf{then}\ \ evaluate[\![E_2]\!]\ env\ sto \\
\qquad \textbf{else}\ \ evaluate[\![E_3]\!]\ env\ sto
\end{array}
$$

(e)

$$
\begin{array}{l}
evaluate[\![E_1 \,\mathbf{?}\, E_2 \,\mathbf{:}\, E_3]\!]env\ sto\ = \\
\quad \textbf{let}\ (\ truth\text{-}value\ b_1, sto_1)\ =\ evaluate[\![E_1]\!]\ env\ sto \\
\quad \textbf{in if}\ b_1\ =\ \textbf{true} \\
\qquad \textbf{then}\ \ evaluate[\![E_2]\!]\ env\ sto_1 \\
\qquad \textbf{else}\ \ evaluate[\![E_3]\!]\ env\ sto_1
\end{array}
$$

5. Fixpoint Combinator

Define the combinator

$$\mathbf{Y} = (\lambda xy\,.\,y(xxy))(\lambda xy\,.\,y(xxy))$$

(a) Show that $\mathbf{Y}F$ is a fixed point for F, i.e., prove the *fixed point identity*:

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

Show all reductions used and label them by type beta or eta (β or η). *HINT:* Write $\mathbf{Y}$ as $\mathbf{AA}$ and reduce $\mathbf{AA}F$. Also, remember how to parenthesize the expression *uvw*.

(b) Discuss the result of applying $\mathbf{Y}$ to the identity combinator $\mathbf{I} = \lambda x\,.\,x$.

Solution

(a) First apply $\mathbf{Y} = \mathbf{AA}$ to F, group application left to right, and then use the definition of $\mathbf{A}$ in the left occurrence of $\mathbf{A}$:

$$\mathbf{Y}F = \mathbf{AA}F = (\mathbf{AA})F = (\ (\lambda xy\,.\,y(xxy))\mathbf{A}\ )\ F$$

This is a β-redex with respect to λx, so perform the reduction. This leads to a second β-redex with respect to λy:

$$\rightarrow_\beta\ (\lambda y\,.\,y(\mathbf{AA}y))\ F\ \rightarrow_\beta\ F(\mathbf{AA}F)\ =\ F(\mathbf{Y}F)$$

So $\mathbf{Y}F \rightarrow^*\ F(\mathbf{Y}F)$ as required.

(b) Using the result just proved above, with F set to $\mathbf{I}$:

$$\mathbf{YI} \rightarrow^*\ \mathbf{I}(\mathbf{YI})\ \rightarrow_\beta\ \mathbf{YI}$$

since $\mathbf{I}x = x$. Clearly $\mathbf{YI}$ has no normal form.

This is consistent with the observation that the program $f(x) = f(x)$ where $\tau = \mathbf{I}$ has a minimal fixpoint that is undefined for all arguments!

Long Problems (do 2)

3. Delayed Argument Evaluation

Parameter passing mechanisms in programming languages differ in regard to the time at which arguments (actuals) to procedure/function calls are evaluated. In languages using call-by-value, arguments are evaluated at the point of call. In languages using call-by-name or various forms of lazy evaluation, arguments are not evaluated until the corresponding formal is referenced in the procedure/function body.

Consider a language *Sloth* in which the time of evaluation of each argument is completely under the programmer's control. Assume *Sloth* has a syntax and semantics similar to Pascal. For simplicity, assume that variables can refer only to integers (there are no arrays, records and no other primitive data types.) Unlike Pascal, however, *Sloth* does not have call-by-value and call-by-reference; instead, the actual/formal association is controlled by the **in** and **out** commands described below. The keyword **var** does not appear in *Sloth* parameter lists: only the types of each parameter are given in a procedure/function definition.

In addition, *Sloth* has two new statements (commands), described below. These commands can appear only in procedure/function bodies. Here *X* refers to any identifier that is a formal parameter that is in scope.

in *X* — This command causes the argument (actual) corresponding to the parameter (formal) *X* to be evaluated in the environment of the *caller*. Any side-effects resulting from such an evaluation are incurred at this point. The value resulting from this evaluation becomes the *r*-value of *X*.

out *X* — This command updates the location given by the *l*-value of the argument (actual) corresponding to the parameter (formal) *X*. The designated location is updated with the current *r*-value of the parameter (formal) *X*. Flow of control is not affected.

If a parameter *X* is referenced in a procedure/function body before **in** *X* has been executed, it is a run-time error. If *X* is not a parameter, it is a syntax error.

(a) (8 points). In *Sloth*, complete the definition of the following procedure, which when called will swap the contents of its two arguments:

procedure *swap*(*X : integer*, *Y : integer*);

(b) (10 points). Describe how to implement the parameter passing mechanism of *Sloth*. In your answer, address the following points:

(1) For each argument, what is transmitted to the callee's activation at call time?

(2) What code must be generated for each **in** *X*?

(3) What code must be generated for each **out** *X*?

(4) What code must be generated when parameter *X* is referenced in the procedure/function body?

(5) What code must be generated when parameter *X* is the target of an assignment in the procedure/function body?

(c) (8 points). Sketch how to simulate the effect of a call-by-name parameter in *Sloth*.

Solution

(a). *Sloth* is flexible enough to simulate call-by-copy-in/copy-out:

```
procedure swap(X : integer, Y : integer);
        var T : integer;
        begin
                in X; in Y;
                T←X; X←Y; Y←T;
                out X; out Y;
        end ;
```

(b). The key implementation technique is to use a thunk for each argument.

(1) At the time of each call, a thunk is created for each actual; that thunk (or a pointer to it) is transmitted to the callee. The environment enclosed in the thunk will be the environment of the caller. When called, the thunk will evaluate the argument expression in the caller's environment, and return an *l*-value. If the actual argument is a simple identifier, the thunk will return its *l*-value (part of the caller's environment). If the actual argument is an expression, the thunk will return the *l*-value of a temporary location in the caller's activation that contains the appropriate *r*-value.

(2) Each **in** *X* results in a call to the thunk associated with *X*. This call returns an *l*-value that is dereferenced to obtain an *r*-value. The resulting *r*-value is assigned to a memory cell in the callee's AR associated with formal *X*.

(3) Each **out** *X* results in a call to the thunk associated with *X*. This call returns an *l*-value, which is then updated with the contents of the local memory cell associated with *X*.

(4) A reference to *X* in the body is simply a reference to the *r*-value of the local memory cell associated with *X*.

(5) An assignment to *X* in the body is simply an assignment to the *l*-value of the local memory cell associated with *X*.

(c) Whenever a formal *X* is referred to in the procedure body, replace this reference by the command **in** *X* followed by a reference to *X*. This might require introduction of new temporaries. For example, the assignment

```
Y←2*X + 1/X ;
```

would result in

```
in X ;
T1←2*X ;
in X;
T2←1/X ;
Y←T1 + T2 ;
```

Whenever the formal *X* appears as the target of an assignment

```
X←e ;
```

replace it by the assignment followed immediately by **out** *X*:

```
X←e ;
out X;
```

III. Label Parameters

Some languages (e.g., Algol 60) permit labels to be passed as parameters to procedures. Assume *P* is a statically bound language similar to Pascal, but extended to allow label parameters. To clarify the meaning of label parameters, consider a procedure declaration:

```
procedure P(X: integer; L: label);
     ...
     if X = 0 then goto L;
     ...
end P;
```

with formal label parameter `L`. Suppose this procedure is called via `P(3,T)` from some executing environment e_{caller}. The actual label parameter `T` is resolvable to some "label literal" `t` (like "`999:`") declared in some environment env_t (note that `T` might itself be a formal label parameter in the caller, or it might be a label literal whose declaration is visible to the caller). When execution of the call to `P` encounters **`goto`** `L`, control is transferred to the statement labeled `t` and execution resumes *in the environment of declaration* env_t *of the label* `t`. Some activations on the stack above env_t—in particular that of the activation of `P`—will need to be deleted since control can never return to them.

Describe the important features of the stack implementation of *P* that are relevant to the support of label parameters. Assume that *P* is like Pascal, with static binding, and with label literals denoted by numerals followed by a colon (`:`), like "`123:`". For simplicity, assume that *P* does not support procedural parameters. Include in your description

(1) the contents of an activation record,

(2) how a label that is used as an actual is represented (i.e., is it merely an address?)

(3) the implementation of a call having a label as its only actual argument, such as `U(999:)`, where `999:` is a literal of type `label` whose scope includes `U`;

(4) the implementation of a call having a formal label parameter as its only actual argument, such as `U(T)` where `T` is a parameter of type `label` in `U`;

(5) the implementation of a **`goto`** to a label formal parameter, such as **`goto`** `T`, where `T` is a parameter of type `label` whose definition is in scope.

Do not discuss issues irrelevant to the implementation of `label` parameters, such as access to integer variables, etc.

Solution

A label parameter can be thought of as a degenerate case of procedure parameter. However there is an important difference in that calls to procedure parameters normally return to their caller without the large-scale deletion of activation records that occurs upon transfer to label formals.

(1) One possible activation record format consists of a control (dynamic) link, an access (static) link, space for a resumption address and space for parameters, including label parameters.

(2) When a call is made to a procedure `P` having actual label parameter `T`, `T` can be either (i) a label constant `t` visible to the caller, or (ii) a formal parameter declared in the caller.

Label constants (but not label parameters) are resolved at code-generation time into fixed entrypoints into code segments that are accessible through the symbol table.

In case (i), the compiler knows the static distance *sd*(`t`) from the caller's activation to that of the label constant's definition. The compiler generates code to trace back along *sd*(`t`) static links and to set `ep` to point to the AR of definition. The compiler also knows the code `entrypoint` corresponding to label constant `t`. The compiler generates code to pass a *closure* consisting of the pair `(entrypoint, ep)` by copying this pair into the new AR for `P`, at the appropriate parameter position. Other code generated for a call consists of the usual: transmission of non-label parameters into the new AR, saving the resumption point in the caller's AR, setting the dynamic link in the new AR to point to the caller's AR, and setting the static link in the new AR to point to

the correct AR for the environment of definition of the name `P`. Finally the new AR is pushed, the current environment pointer is set to the new AR (context switch), and control is transferred to the (known) entrypoint of `P`.

The static link is needed to resolve references to names free in `P`, including possibly non-local label constants mentioned there in **`goto`**s or in calls.

In case (ii), the actual `T` is represented by a closure `cl(T)` at a known place in the caller's AR. The compiler simply generates code to copy this closure into the appropriate place in the AR of `P`. All other code is the same.

(3) In generating code for **`goto`** `L` where `L` is a formal label name, the compiler makes use of the fact that the closure `cl(L)` for `L` is at a known location in the current AR. The compiler generates code to pop *all* ARs above that of `cl(L).ep`, to set the current environment pointer to `cl(L).ep` (context switch), and to transfer control to `cl(L).entrypoint`. All the ARs are deleted since control can never resume in any AR above that of e_t. This follows since those environments came into being after e_t, and there is no way to transfer control from an environment to a subsequent one except by procedure call; however a procedure call creates a fresh environment.