CS 242. Autumn 2001.
October 30,2001

CS 242 *Sample* Midterm Exam

This is a *closed*-book exam. The maximum possible score is 100 points. Make sure you print your name legibly and sign the honor code below. All of the intended answers may be written within the space provided. You may use the back of the preceding page for scratch work. If you need to use the back side of a page to write part of your answer, be sure to mark your answer clearly.

The following is a statement of the Stanford University Honor Code:

A. *The Honor Code is an undertaking of the students, individually and collectively:*

   (1) *that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;*

   (2) *that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.*

B. *The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.*

C. *While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.*

I acknowledge and accept the Honor Code.

(Signature)

(Print your name)

Prob	# 1	# 2	# 3	# 4	# 5	# 6	# 7	Total
Score								
Max	8	12	20	14	12	14	20	100

1. (*8 points*) .... True or False

Mark each statement *true* or *false,* as appropriate.

_____ (a) If evaluation of a Lisp expression does not terminate, its value is the special symbol `nil`.

_____ (b) In Lisp, the expressions `(cons 'A 'B)` and `'(A B)` evaluate to equivalent memory structures.

_____ (c) All standard general-purpose programming languages are capable of defining the same class of functions on the integers.

_____ (d) In a standard compiler, type checking comes before parsing since this avoids the unnecessary work of parsing expressions that do not typecheck.

_____ (e) The variable x occurs bound in $\lambda y.\,(y\,((\lambda x.\,(z\,x))x))$.

_____ (f) In lambda calculus, *confluence* means that the expressions $(M\,N)\,P$ and $M\,(N\,P)$ eventually evaluate to the same normal form, possibly by reduction paths of different length.

_____ (g) If the ML typechecker rejects a program, this implies that running the program would have produced a run-time type error.

_____ (h) `Callcc` calls a function with the current continuation as argument.

2. (*12 points*) .... Terminology

Define the following terms.

(a) (*3 points*) Run-time type error

(b) (*3 points*) Pure functional language

(c) (*3 points*) Access link

(d) (*3 points*) Exception handler

3. (*20 points*) .... Single-Assignment Languages

A number of so-called *single-assignment languages* have been developed over the years, many designed for parallel scientific computing. Single assignment conditions are also used in program optimization and in hardware description languages, which may allow only one assignment to each variable per clock cycle.

One example single-assignment language is *SISAL,* which stands for Streams and Iteration in a Single Assignment Language. Another is *SAC,* or Single-Assignment C. Programs in single-assignment languages must satisfy the following condition:

> *Single-Assignment Condition:* During any run of the program, each variable may be assigned a value only once, within the scope of the variable.

The following program fragment satisfies this condition

```
if ( ... ) then x = 42+29/3  else x = 13.39;
```

since only one branch of the if-then-else will be executed on any run of the program. The program `x=2; loop_forever; x=3` also satisfies the condition since no execution will complete both assignments.

Single-assignment languages often have specialized loop constructs, since otherwise it would be impossible to execute an assignment inside a loop body that gets executed more than once. Here is one form, from SISAL:

```
for ⟨range⟩
   ⟨optional body⟩
returns ⟨returns clause⟩
end for
```

An example illustrating this form is the following loop, which computes the dot (or inner) product of two vectors:

```
for i in 1, size
   elt_prod := x[i] * y[i]
returns value of sum elt_prod
end for
```

This loop is parallelizable since different products `x[i] * y[i]` can be computed in parallel. A typical SISAL program is composed as a sequential outer loop containing a set of parallel loops.

Suppose you have the job of building a parallelizing compiler for a single-assignment language. Assume that the programs you compile satisfy the single-assignment condition and do not contain any explicit process fork or other parallelizing instructions. Your implementation must find parts of programs that can be safely executed in parallel, producing the same output values as if the program was executed sequentially on a single processor.

Assume for simplicity that every variable is assigned a value before the value of the variable is used an in expression. Also ssume that there is no potential source of side effects in the language you are compiler other than from assignment.

(a) (*5 points*) Explain how you might execute parts of the sample program

```
x = 5;
y = f(g(x),h(x));
if y==5 then z=g(x) else z=h(x);
```

in parallel. More specifically, assume that your implementation will schedule the following processes in some way:

process 1 – set x to 5
process 2 – call `g(x)`
process 3 – call `h(x)`
process 4 – call `f(g(x),h(x))` and set `y` to this value
process 5 – test `y==5`
process 6 – call `g(x)` and then set `z=g(x)`
process 7 – call `h(x)` and then set `z=h(x)`

For each process, list the processes that this process must wait for and list the processes that can be executed in parallel with it. For simplicity, assume that a call cannot be executed until the parameters have been evaluated and assume that processes 6 and 7 are *not* divided into smaller processes that execute the calls but do not assign to `z`. Assume that parameter passing in the example code is by value.

(b) (*3 points*) If you further divide process 6 into two processes, one that calls `g(x)` and one that assigns to `z`, and similarly divide process 7 into two processes, can you execute the calls `g(x)` and `h(x)` in parallel? Could your compiler correctly eliminate these calls from processes 6 and 7? Explain briefly.

(c) (*2 points*) Would the parallel execution of processes you describe in parts (a) and (b), if any, be correct if the program does not satisfy the single-assignment condition? Explain briefly.

(d) (*6 points*) Is the single-assignment condition decidable? Specifically, given an program written in a subset of C, for concreteness, is it possible for a compiler to decide whether this program satisfies the single-assignment condition? Explain why or why not. If not, can you think of a decidable condition that would imply single-assignment and allow many useful programs to be written?

(e) (*4 points*) Suppose a single-assignment language has no side-effecting operations other than assignment. Does this language pass the pure functional language test (also called the declarative language test) discussed in class and in the reader? Explain why or why not?

4. (*14 points*) .... Lambda Calculus

This problem asks about the following ML code fragment, which can be translated into lambda calculus:

```
(let fun currysubtract(a) = fn (b) =>
     let fun negate(y) = ~y
     in
        a + negate(b)
     end
in
     currysubtract 23
end) 19
```

Note that ML uses $\sim$ instead of $-$ to represent unary negation of numbers.

The translation of this expression into lambda calculus is

$$(\lambda cs.\, cs\ 23)\ (\lambda a.(\lambda b.((\lambda n.\, a + (n\ b))\ (\lambda c.\, -c))))\ 19$$

where `currysubtract` has been abbreviated cs and `negate` abbreviated to n.

(a) (*7 points*) Use leftmost β reduction with the above expression to arrive at a normal form. Recall that leftmost reduction means that if two lambda evaluations are possible at a given step, you should expand the one furthest to the left.

$$(\lambda cs.\, cs\ 23)\ (\lambda a.(\lambda b.((\lambda n.\, a + (n\ b))\ (\lambda c.\, -c))))\ 19$$

$=_{LM\beta}$

$=_{LM\beta}$

$=_{LM\beta}$

$=_{LM\beta}$

$=_{LM\beta}\quad 23 + -19$

$=\quad 4$

(b) (*7 points*) Use rightmost β reduction with the above expression to arrive at a normal form. Recall that rightmost reduction means that if two lambda evaluations are possible at a given step, you should expand the one furthest to the right.

$$(\lambda cs.\, cs\ 23)\ (\lambda a.(\lambda b.((\lambda n.\, a + (n\ b))\ (\lambda c.\, -c))))\ 19$$

$=_{RM\beta}$

$=_{RM\beta}$

$=_{RM\beta}$

$=_{RM\beta}$

$$\begin{aligned} &=_{RM\beta} && 23 + -19 \\ &= && 4 \end{aligned}$$

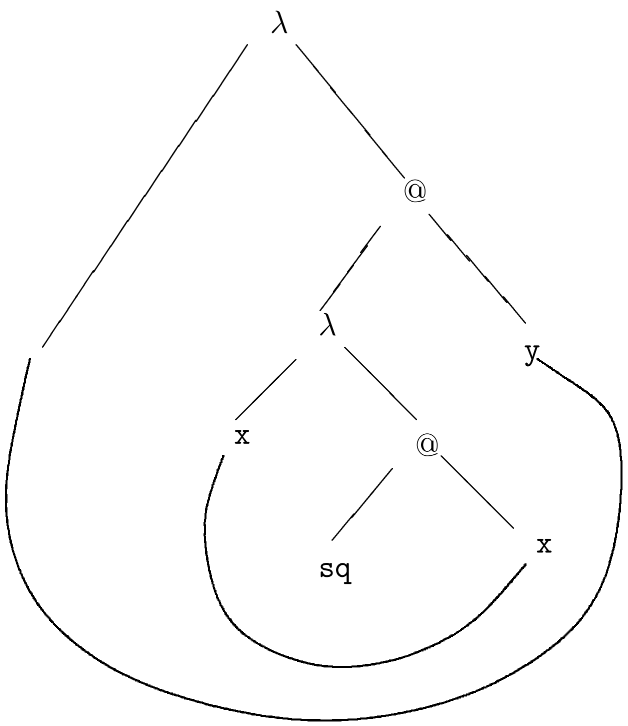
5. (*12 points*) .... **Type Inference**

Use the parse graph below to calculate the ML type for the function:

$\lambda y.((\lambda x.\mathrm{sq}(x))y)$

Where sq is a function `sq: int` $\to$ `int`.

Show your work on the parse graph.


λ
@
λ
y
x
@
sq
x

6. (*14 points*) .... ML, Function Calls and Memory Management

This question asks about memory management in the evaluation of the following ML expressions (with line numbers). This input consists of two declarations and an expression with a declaration that is local to the expression. There are therefore three important scopes, the one containing the first declaration of `myop`, the one following with the declaration of `recurse`, and the scope containing the second declaration of `myop` that is local to the expression.

```
1       fun myop(x,y) = x*y;
2       fun recurse(n) =
3             if n=0 then 2
4             else myop(n, recurse(n-1));
5       let
6         fun myop(x,y) = x + y
7       in
8         recurse(1)
9       end;
```

(a) (*8 points*) Assume that expressions are evaluated using static scope. Fill in the missing information in the following depiction of the run-time stack after the last call to `myop`, caused by execution of line 4 of this code fragment. (At this point, a call to `recurse(0)` has been made, it has returned and then its activation record has been popped off the stack). Remember that function values are represented by closures, and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code. Remember also that in ML function arguments are evaluated before the function is called.

In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure, compiled code or list cell. Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link" and the number of the activation record of the dynamically enclosing scope in the slot labeled "control link." The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables, function parameters and line numbers between the brackets, [ ], given in the diagram.

Activation Records				*Closures*	*Compiled Code*
(1)		access link	(0)		
		control link	(0)		
		myop	•		
(2)		access link	(1)	⟨ (**1**), • ⟩	code for myop defined line [**1**]
		control link	(1)		
		recurse	•		
(3)		access link	(**2**)		
		control link	(**2**)	⟨ (**2**), • ⟩	code for recurse
		myop	•		
(4)	recurse(**1**)	access link	(**2**)		
		control link	(**3**)		
		n	[**1**]	⟨ (**3**), • ⟩	code for myop defined line [**6**]
(5)	myop(**1**,**2**)	access link	(**1**)		
		control link	(**4**)		
		x	[**1**]		
		y	[**2**]		

The arrows should be drawn from `myop` in record 1 to the first closure, from the first closure to the code for `myop` defined in line 1, from the `recurse` in record 2 to the second closure, from the second closure to the code for recurse, from `myop` in record 3 to the third closure and from the third closure to the code for `myop` defined in line 6.

(b) (*6 points*)

i. (*2 points*) What is the value of this expression under static scope? Briefly explain how your stack diagram from part (a) allows you to find which value of `myop` to use.

ii. (*4 points*) What would be the value if dynamic scope were used? Explain the difference between this case and static scope by explaining how specific links in your diagram will be used in each case.

7. (*20 points*) .... Overlapping memory allocation

Fortran and Pascal have different features for giving space allocated to a program multiple uses. A Fortran program can consist of separately compiled functions, called *subroutines.* When functions are linked together to form an executable program, a static memory requirement for the program is determined (at least in all Fortrans used before 1980). This static amount of memory is allocated to the program when the program is started and is neither increased nor decreased as the program runs.

Fortran common blocks provide a mechanism for two separate subroutines to share memory or use the same piece of memory differently. This example common statement

```
COMMON / ACOM / A(100,100), B(100,100), C(100,100)
```

places three arrays in a common block called ACOM. The space allocated for this common block must include at least $3 * 100 * 100$ locations, enough for three arrays with dimension 100×100. These are arrays of floating-point numbers, since the FORTRAN conventions is that names A–H and O–Z are real; names beginning I–N are integer by default. The array names are not part of the common block, so another subroutine could contain the common statement

```
COMMON / ACOM / I(100,200), J(100,100)
```

and access the same space as two integer arrays. This makes the static space requirement of the program less than if the two subroutines each declared their own arrays and did not place them in common. If two Fortran subroutines do not have common statements naming the same common block (such as ACOM here), then the memory allocated to the two subroutines will be disjoint. There are no "global" variables in Fortran subroutines except those contained in common blocks.

Pascal variant records provide another way of giving multiple names to a single memory location. Here are some example type declarations, the second one a variant record.

```
type kind = (square, rectangle, circle);
type shape = record
               center : point;
               case k : kind of
                  square : (side : real);
                  rectangle : (length, height : real);
                  circle : (radius : real);
              end;
```

Let us assume that a point requires 2 words (for two real coordinates) and a kind requires one word. Then a shape is represented using 5 words. The first of these two words always contain the center point of the shape and the third the value k specifying the kind of the shape. If the shape is a square or circle, then only one of the last two words is used (for the side or radius). If the shape is a rectangle, then all five words are used, two for the center point, one for the kind, one for length and one for height.

Here is an example function that computes the area of a shape. The `with` statement allows record components to be accessed without using the dot (.) notation, and the `case` statement is used to access the variant last two words of a `shape` record in different ways, depending on the `kind` of the record.

```
function area( s : shape) : real;
    begin
        with s do
           case k of
               square: area := side * side;
               rectangle : area := length * height;
               circle : area := 3.1415926536 * radius * radius;
         end
end;
```

It is also possible to declare variant record types without giving a record component that distinguishes the types of variants. More specifically, if we declare shapes this way (with no `k:kind`), then a shape is represented as four words, two for the center point and two for the overlapping square, rectangle and circle parts.

```
type kind = (square, rectangle, circle);
type shape = record
               center : point;
               case kind of
                  square : (side : real);
                  rectangle : (length, height : real);
                  circle : (radius : real);
              end;
```

It is possible to manipulate shapes declared this way simply by accessing all components directly. For example,

```
var s : shape;
s.side := 4.52;
s.height := 27
```

will compile and run even though no shape has both `side` and `height`.

Questions:

(a) (*3 points*) Consider a "representative" Fortran program consisting of several subroutines that each declare and use several independent arrays. How much can the static space requirement of the program be reduced by inserting COMMON statements? (This may depend on the program, but make some reasonable assumptions.) Given what you know about Fortran code and the era in which Fortran was popular, do you think this is significant?

(b) (*4 points*) List one other reason why Fortran common blocks are useful (besides saving space) and list one potential problem with Fortran common blocks.

(c) (*3 points*) Consider an example Pascal program using variant records to store geometric information. How much space might be saved using Pascal variant records, in comparison with records that have the same components but do not use variants to place more than one component in a single location? Do you think this is significant?

(d) (*4 points*) List one other reason why Pascal variant records are useful (besides saving space) and list one potential problem with the second (unchecked) form of variant records.

(e) (*3 points*) Can you think of any advantages to having the unchecked variant records in Pascal? Do you think it would be OK to have the first (safe) form only?

(f) (*3 points*) Do you think there is a practical way to make common blocks type safe? What would be the disadvantage of doing this?