Homework 2

Due 10 October

Handout 2
CS242: Autumn 2012
3 October

Reading

1. Read these sections of Chapter 4 of *Concepts in Programming Languages:* Section 4.1.1, Structure of a simple compiler; Section 4.2, Lambda calculus, except Reduction and Fixed Points; and Section 4.4, Functional and imperative languages.
2. An Operational Semantics for JavaScript - Try to read up through section 2.4 for the main ideas. Do not worry about details beyond what is covered in lecture or homework. See CourseWare link under the Mon, Oct 3 lecture for a link to this paper.
3. Chapter 7 of *Concepts in Programming Languages.*

Problems

1. Lambda Calculus Reduction

Use lambda calculus reduction to find a shorter expression for $(\lambda p.\lambda q.\lambda r.p\ q\ r)(\lambda p.\lambda q.p\ q\ r)$. Begin by renaming bound variables. You should do all possible reductions to get the shortest possible expression. What goes wrong if you do not rename bound variables?

2. Translation into Lambda Calculus

A programmer is having difficulty debugging the following `C` program. In theory, on an "ideal" machine with infinite memory, this program would run forever. (In practice, this program crashes because it runs out of memory, since extra space is required every time a function call is made.)

```
int f(int (*g)(...)){ /*  g points to a function that returns an int */
   return g(g);
}
int main(){
  int x;
  x = f(f);
  printf("Value of x = %d\n", x);
  return 0;
}
```

Explain the behavior of the program by translating the definition of `f` into lambda calculus and then reducing the application `f(f)`. This program assumes that the type checker does not check the types of arguments to functions.

3. Lazy Evaluation and Parallelism

In a "lazy" language, a function call `f(e)` is evaluated by passing the *unevaluated* argument to the function body. If the value of the argument is needed, then it is evaluated as part of the evaluation of the body of `f`. For example, consider the function g defined by

```
fun g(x, y) = if x = 0
              then 1
              else if x + y = 0
                   then 2
                   else 3;
```

In a lazy language, the call g$(3, 4+2)$ is evaluated by passing some representation of the expressions 3 and $4+2$ to g. The test x $= 0$ is evaluated using the argument 3. If it were true, the function would return 1 without ever computing $4+2$. Since the test is false, the function must evaluate x + y, which now causes the actual parameter $4+2$ to be evaluated. Some examples of lazy functional languages are Miranda, Haskell and Lazy ML; these languages do not have assignment or other imperative features with side effects.

If we are working in a pure functional language without side-effects, then for any function call f(e_1, e_2), we can evaluate e_1 before e_2 or e_2 before e_1. Since neither can have side-effects, neither can affect the value of the other. However, if the language is lazy, we might not need to evaluate both of these expressions. Therefore, something can go wrong if we evaluate both expressions and one of them does not terminate.

As Backus argues in his Turing Award lecture, an advantage of pure functional languages is the possibility of parallel evaluation. For example, in evaluating a function call f(e_1, e_2) we can evaluate both e_1 and e_2 in parallel. In fact, we could even start evaluating the body of f in parallel as well.

(a) Assume we evaluate g(e_1, e_2) by starting to evaluate g, e_1, and e_2 in parallel, where g is the function defined above. Is it possible that one process will have to wait for another to complete? How can this happen?

(b) Now, suppose the value of e_1 is zero and evaluation of e_2 terminates with an error. In the normal (i.e., eager) evaluation order that is used in C and other common languages, evaluation of the expression g(e_1, e_2) will terminate in error. What will happen with lazy evaluation? Parallel evaluation? (*Hint* [parallel evaluation]: If needed, the result of an evaluated expression is used once it's available, i.e., evaluation of said expression is complete.)

(c) Suppose you want the value for every expression to be the same as that of lazy evaluation, but you want to evaluate expressions in parallel to take advantage of your new pocket-sized multiprocessor. What actions should happen, if you evaluate g(e_1, e_2) by starting g, e_1, and e_2 in parallel, if the value of e_1 is zero and evaluation of e_2 terminates in an error?

(d) Suppose, now, that the language contains side-effects (as in C or ML). What if e_1 is z, and e_2 contains an assignment to z. Can you still evaluate the arguments of g(e_1, e_2) in parallel? How? Or why not?

4. Modifying functional programs

Quicksort is a well-known sorting algorithm. The algorithm works by choosing some element of the list to be sorted, called the *pivot,* and then splitting the list into elements less than the pivot and elements greater than the pivot. Then each sublist is sorted and the results concatenated. While the worst-case behavior of Quicksort is not very good, the average behavior is excellent. Here are implementations of Quicksort in Haskell, a pure functional language, and C, an impure and not really functional language.

The first five lines below are the Haskell program. The first line says that qsort of the empty list is the empty list. The second line applies to a list argument matching the pattern x:xs, binding x to the first element of the list and xs to the list of remaining elements. Lines 2–5 of the code "say" that Quicksort of a nonempty list x:xs is the list obtained by concatenating the Quicksort of the elements less than x with x itself and the Quicksort of the elements greater than or equal to x. Questions about the programs appear after the code.

```
qsort []     = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
               where
                 elts_lt_x   = [y | y <- xs, y < x]
                 elts_greq_x = [y | y <- xs, y >= x]
```

```
qsort( a, lo, hi ) int a[], hi, lo;
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];

    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    t = a[l];
    a[l] = a[hi];
    a[hi] = t;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

(a) Which list element is used as the pivot in each example?

(b) Which code seems easier to understand? Why? What if you had never programmed in either language before?

(c) Each Haskell list comprehension, such as `[y | y <- xs, y < x]`, is computed by a separate recursive pass through the list `xs`. What might be a better approach?

(d) Explain how to modify the C code to use a randomly chosen list element as the pivot.

(e) How much extra space (memory), in addition to the space used to represent the input list, do you think the Haskell program uses? Since we haven't talked about how Haskell is implemented, just try to make a reasonable, educated estimate. How does this compare to the memory requirements of the C program?

5. Operational Semantics

This problem asks you about operational semantics for a simple language with assignment, addition, and functions. The expressions of this language are given by the grammar

$$e \quad ::= \quad n \mid x \mid x = e \mid e + e \mid \lambda x.\, e \mid ee$$

where n can be any number $(0, 1, 2, 3, \ldots)$ and ee is the application of one expression (assumed to be a function) to another (an argument to the function). Like the example discussed in lecture, we can define the operational semantics with respect to a mapping $\sigma : \mathit{Var} \to \mathcal{N}$, where Var is the set of variables that may appear in expressions and $\mathcal{N}$ is the set of numbers that can be values of variables. To have some reasonable terminology, we will call σ a *store* and a pair $\langle e, \sigma \rangle$ a *state*.

When a program executes, we hope to reach a *final state* $\langle e, \sigma \rangle$ where e is either a number or a function beginning with a λ. We call an expression that is either a number or a function beginning with a λ a *value*.

(a) Let's first look at arithmetic expressions as in lecture. Two rules for evaluating summands of sum are

$$\frac{a_1 \quad \rightarrow \quad a_1'}{\langle a_1 + a_2,\ \sigma\rangle \quad \rightarrow \quad \langle a_1' + a_2,\ \sigma\rangle}$$

$$\frac{a_2 \quad \rightarrow \quad a_2'}{\langle n + a_2,\ \sigma\rangle \quad \rightarrow \quad \langle n + a_2',\ \sigma\rangle}$$

We assume that it is a single execution step to add two numbers, so we have the rule

$$\frac{n, m, p \text{ are numbers with } n + m = p}{\langle n + m,\ \sigma\rangle \quad \rightarrow \quad \langle p,\ \sigma\rangle}$$

The value of a variable depends on the store, as specified by this evaluation rule

$$\langle x, \sigma\rangle \quad \rightarrow \quad \langle \sigma(x), \sigma\rangle$$

Show how these three rules let you evaluate an expression $x+y$ to a sum of numbers. Assume σ is a store with $\sigma(x) = 2$ and $\sigma(y) = 3$. Write your answer as an execution sequence of the form below, *with an explanation.*

$$\langle x + y,\ \sigma\rangle \quad \rightarrow \quad \langle \_\_\_\_ + \_\_\_\_,\ \sigma\rangle \quad \rightarrow \quad \langle \_\_\_\_ + \_\_\_\_,\ \sigma\rangle \quad \rightarrow \quad \langle \_\_\_\_,\ \sigma\rangle$$

(b) Put is the function on stores with $Put(\sigma, x, n) = \sigma'$ with $\sigma'(x) = n$ and $\sigma'(y) = \sigma(y)$ for all variables y other than x. Using Put, the execution rule for assignment can be written

$$\langle x = n, \sigma\rangle \quad \rightarrow \quad \langle n, Put(\sigma, x, n)\rangle$$

In this particular language, an assignment changes the store, as usual. In addition, as you can see from the operational semantics of assignment, an assignment is an expression whose value is the value assigned.

If we have an assignment with an expression that has not been evaluated to a number, then we can use this evaluation rule:

$$\frac{\langle e, \sigma\rangle \quad \rightarrow \quad \langle e', \sigma'\rangle}{\langle x = e, \sigma\rangle \quad \rightarrow \quad \langle x = e', \sigma'\rangle}$$

Combining these rules with rules from part (a), show how to execute $\langle x = x + 3,\ \sigma\rangle$ when σ is a store with $\sigma(x) = 1$. Write your answer as an execution sequence of the form below, *with an explanation.*

$$\langle x = x + 3,\ \sigma\rangle \quad \rightarrow \quad \langle x = \_\_\_\_ + \_\_\_\_,\ \sigma\rangle \quad \rightarrow \quad \langle x = \_\_\_\_,\ \sigma\rangle \quad \rightarrow \quad \langle \_\_\_\_,\ \_\_\_\_\rangle$$

(c) Show how to execute $\langle (x = 3) + x,\ \sigma\rangle$ in the same level of detail, *including an explanation.*

(d) Show how to execute $\langle x = (x = x + 3) + (x = x + 5),\ \sigma\rangle$ when σ is a store with $\sigma(x) = 1$ in the same level of detail, *including an explanation.*

6. Parameter passing comparison

For the following Algol-like program, write the number printed by running the program under each of the listed parameter passing mechanisms.

```
begin
   integer i;

   procedure pass ( x, y );
      integer x, y;  // types of the formal parameters
      begin
         x := x + 1;
         y := x + 1;
         x := y;
         i := i + 1
      end

   i := 1;
   pass (i, i);
   print i
end
```

(a) pass-by-value

(b) pass-by-reference

(c) pass-by-value/result

Note: in *pass-by-value-result*, also called call-by-value-result and copy-in/copy-out, parameters are passed by value, with an added twist. More specifically, suppose a function `f` with a pass-by-value-result parameter `u` is called with actual parameter `v`. The activation record for `f` will contain a location for formal parameter `u` that is initialized to the R-value of `v`. Within the body of `f`, the identifier `u` is treated as an assignable variable. On return from the call to `f`, the actual parameter `v` is assigned the R-value of `u`.

The following pseudo-Algol code illustrates the main properties of pass-by-value-result.

```
var x : integer;
x := 0;
procedure p(value-result y : integer)
   begin
      y := 1;
      x := 0;
   end;
p(x);
```

With pass-by-value-result, the final value of `x` will be 1: since y is given a new location distinct from `x`, the assignment to `x` does not change the local value of `y`. When the function returns, the value of `y` is assigned to the actual parameter `x`. If the parameter were passed by reference, then `x` and `y` would be aliases and the assignment to `x` would change the value of `y` to 0. If the parameter were passed by value, the assignment to `y` in the body of `p` would not change the global variable `x` and the final value of `x` would also be 0.

7. Closures and Access Links

Consider the following Javascript code.

```
var x = 1;
function f(y) {
    return y + x;
```

```
}
var q = 2;
function h(z) {
    return f(z) * q;
}
var w = h(3);
```

(a) What is the value of w?

(b) Fill in the missing parts in the following diagram of the run-time structures for execution of this code up to the point where the call inside h(3) is about to return. You can draw pointers to show the access links on the stack and in closures, or simply write in the number of the appropriate activation record. The activation records are numbered 1–7, from the top.

Activation Records | *Closures* | *Compiled Code*

(1)		access link	(0)
		x	
(2)		access link	( )
		f	•
(3)		access link	( )
		q	
(4)		access link	( )
		h	•
(5)		access link	( )
		w	
(6)	h(3)	access link	( )
		z	
(7)	f(3)	access link	( )
		y	

⟨ ( ), • ⟩ — code for f …

⟨ ( ), • ⟩ — code for h

8. Recursion in JavaScript

JavaScript allows functions to be declared in syntactically different ways that may look equivalent but are not. This question asks you to figure out some examples by drawing activation records and pointers to closures.

Here is sample code illustrating two ways to define a recursive function:

```
var f1 = function(x){...; ⟨ recursive call to f1 ⟩}
var f2 = function g(x){...; ⟨ recursive call to g ⟩}
```

We will call the first an "un-named recursive function" and the second a "named recursive function".

Un-named recursive function: In the first declaration, variable `f1` is initialized to an anonymous function which has recursive call to `f1`. When this line of code is executed, a closure for the function is created. The environment pointer of this closure points to the activation record of the scope where `f1` is declared.

Named recursive functions: In the second declaration, variable `f2` is initialized to the "named" function `g`, which is a recursive function. We will call `g` the "inner function" and call `f2` the "outer function". Note that the recursive call is to the inner function `g` and not the outer function `f2`. When this code is executed, a closure for the function `g` is created and `f2` is set to point to this closure. However, an additional activation record is also created to represent the scope where function name `g` is defined. In this additional activation record, a field for the value of `g` points to the the closure for function `g`. The access link for the second activation record points to the

activation record where `f2` is defined. Finally, the environment pointer for the function closure points to the second activation record (rather than the first) to allow the function to refer to the function name `g`. We will explore how this works in parts (d) and (e) below.

Questions:

Consider the following Javascript code fragment:

```
1:   var f = function(x){if (x === 1){return 1;}  else {return x*f(x-1);}};
2:   var h = f;
3:   var f = function(x){if (x === 2){return 0;} else{return 10;}};
4:   h(2);  // return 20
```

(a) Fill in the missing information in the following depiction of the run-time stack *just after the recursive call* in `h(2)` on line 4. For simplicity, `f` and `h` are shown in the same activation record.

In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code. 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". The first one is done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.

Activation Records *Closures* *Compiled Code*

(1)		access link	(0)
		f	•
		h	•
(2)	h(2)	access link	( )
		x	
(3)	f(1)	access link	( )
		x	

A⟨( ), • ⟩ | function, line 1 |

B⟨( ), • ⟩ | function, line 3 |

(b) Which closure is used to find the function code for `f(1)` in activation record (3) above: A or B?

(c) If we call `h(2)` after line 2, we get `h(2)` = 2. However, the call on line 4 returns `h(2)` = 20. Explain briefly how this happens, using $f_1()$ to refer to the version of `f` on line 1 and $f_3()$ to refer to the version of `f` on line 3.

(d) Suppose we change the first declaration of `f` to use a named recursive function:

```
1:  var f = function g(x){if (x === 1){return 1;} else {return x*g(x-1);}}
2:  var h = f;
3:  var f = function(x){if (x === 2){return 0;} else{return 10;}};
4:  h(2);
```

Fill in the missing information in the following depiction of the run-time stack *just after the recursive call* in `h(2)`.

Activation Records

(1)		access link	(0)
		f	•
		h	•
(2)		access link	( )
		g	•
(3)	h(2)	access link	( )
		x	
(4)	g(1)	access link	( )
		x	

Closures

A⟨ ( ), • ⟩

B⟨ ( ), • ⟩

Compiled Code

| function, line 1 |

| function, line 3 |

(e) What is the value returned by the call `h(2)` on line 4? Explain in a few words.