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. Symbolic Evaluation

The JavaScript program fragment

```javascript
function f(x) {
  return x*3;
}
function g(y) {
  return 5-y;
}
f(g(1));
```

can be written as the following lambda expression:

\[
\left( (\lambda f. \lambda g. f (g 1)) \right) \left( (\lambda x. x * 3) \right) \left( \lambda y. 5 - y \right)
\]

Reduce the expression to a normal form in two different ways, as described below.

(a) Reduce the expression by choosing, at each step, the reduction that eliminates a \( \lambda \) as far to the left as possible.

(b) Reduce the expression by choosing, at each step, the reduction that eliminates a \( \lambda \) as far to the right as possible.

2. Lazy Evaluation and Parallelism

In a “lazy” language, a function call \( f(e) \) is evaluated by passing the unevaleduated 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

```javascript
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?

3. .............................................. Lazy and Eager Evaluation

Function calls in most languages, including ML, Lisp, C, and Java, are evaluated by first evaluating the arguments to the function and then performing the function call. This is called eager evaluation. In the functional language Haskell, function arguments are not evaluated until they are needed—arguments are left unevaluated until the program actually uses their value. This is called lazy evaluation. In lazy evaluation, if an argument is never used, then it is never evaluated.

Lazy evaluation can be implemented using thunk functions, which are no-argument functions that are called when an expression needs to be evaluated. For example, a lazy expression $f(e)$ can be compiled to eager code that passes a thunk for $e$ to the function body of $f$. The thunk for $e$ will only be evaluated if both $f$ is evaluated during the course of the program, and the body of $f$ needs the value of $e$. For example, if the body of $f$ contains $e + 1$, then the thunk for $e$ will be evaluated whenever $f$ is.

Consider the following code fragment, written in a Javascript-like language.

```javascript
function times3 (x) { return x + x + x; }
times3 (5 * 7);
```

Let us think about how we can implement the eager and lazy evaluation strategies for this code using the (eager) Javascript language.
Because Javascript is an eager language, we don’t have to make any changes at all to implement an eager evaluation strategy in Javascript. Simply running the code as written will first compute $5 \times 7 = 35$ and then call `times3` with the argument 35.

We can implement lazy evaluation in Javascript by writing the following code:

```javascript
function times3 (x_thunk) { return x_thunk() + x_thunk() + x_thunk(); }
times3 (function () { return 5 * 7; } );
```

The `times3` function expects a thunk that produces the value of its argument, and calls the function where the value of the argument is needed. Rather than receiving the value 35, `times3` now takes a function that evaluates $5 \times 7$ when needed.

(a) Haskell is a pure functional language. What does this imply about the three values of $x$ in any call to `times3`?

(b) Since `times3(x)` contains three occurrences of $x$, the thunk for the function argument will be called three times. What trick could you use in a Haskell compiler to improve the efficiency of the compiled code for `times3`?

(c) Translate the following eager Javascript code into a lazy version using thunks. In the spirit of Haskell, the lazy version should return thunks.

```javascript
function g (x, y) {
  if (x < 0) return 0;
  else return y;
}
```

```javascript
function g ( ________ , ________ ) {
  if ( ______________ < 0 )
    return ______________ ;
  else ______________ ;
}
```

(d) Does the choice of evaluation strategy change the eventual output of a pure functional program? More specifically, if $f$ is a function and $e_1 \ e_2 \ldots \ e_n$ are arguments, is the value of $f(e_1, e_2, \ldots, e_n)$ the same under eager and lazy evaluation strategies, assuming that all are written in a pure functional language? If so, explain briefly, if not, provide a counterexample and brief explanation. (Hint: Consider the example functions used in this question, and all possible arguments they might be applied to.)

4. ........................................................................................................... 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

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

where $n$ can be any number (0, 1, 2, 3,…) 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 $\sigma : \text{Var} \rightarrow \mathcal{N}$, where $\text{Var}$ 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 $\sigma$ a store and a pair $(e, \sigma)$ a state.

When a program executes, we hope to reach a final state $(e, \sigma)$ where $e$ is either a number or a function beginning with a $\lambda$. We call an expression that is either a number or a function beginning with a $\lambda$ a value.
(a) Let’s first look at arithmetic expressions as in lecture. Two rules for evaluating summands of sum are

\[
\begin{align*}
  a_1 & \rightarrow a'_1 \\
  \langle a_1 + a_2, \sigma \rangle & \rightarrow \langle a'_1 + a_2, \sigma \rangle \\
  a_2 & \rightarrow a'_2 \\
  \langle n + a_2, \sigma \rangle & \rightarrow \langle n + a'_2, \sigma \rangle
\end{align*}
\]

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

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

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

\[
\langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle
\]

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

\[
\langle x + y, \sigma \rangle \rightarrow \langle \_, \_, \sigma \rangle \rightarrow \langle \_, \_, \sigma \rangle \rightarrow \langle \_, \sigma \rangle
\]

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

\[
\langle x = n, \sigma \rangle \rightarrow \langle n, \text{Put}(\sigma, x, n) \rangle
\]

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:

\[
\frac{\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle}{\langle x = e, \sigma \rangle \rightarrow \langle x = e', \sigma' \rangle}
\]

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

\[
\langle x = x + 3, \sigma \rangle \rightarrow \langle x = \_, \_, \sigma \rangle \rightarrow \langle x = \_, \sigma \rangle \rightarrow \langle \_, \_, \_ \rangle
\]

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

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

Recall that for a nonnegative number \( x \), the number \( x! \), read \( x \) factorial, is the product of all the positive integers less than or equal to \( x \). By convention, \( 0! = 1 \). Consider the following procedure, written in a JavaScript-like notation, that is intended to set \( y \) to \( x! \).

```
function fact(x, y) {
    y = 1
    while ( x > 0 ) {
        y = x*y;
        x = x-1;
    }
}
```

This procedure may not behave as intended for certain combinations of parameter-passing methods. We will consider three: call-by-value, call-by-reference, and call-by-value-result.

In call-by-value-result, also called copy-in/copy-out, parameters are passed by value, with an added twist. Suppose a function \( f \) with a call-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 \( u \).

(a) Assume \( a \) and \( b \) are assignable integer variables with distinct L-values. Assume the initial value of \( a \) is greater than 0 and \( b \) is arbitrary. Which parameter-passing combinations will make \( b = a! \) after a call `fact(a, b)` returns, where \( a_i \) is the initial R-value of \( a \)? Express your answer by writing yes/no in the empty places in following table.

<table>
<thead>
<tr>
<th></th>
<th>( x ) by value</th>
<th>( x ) by ref</th>
<th>( x ) by value-result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y ) by value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( y ) by ref</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( y ) by value-result</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) The binomial coefficient \( \binom{n}{r} = \frac{n!}{r!(n-r)!} \), sometimes pronounced “\( n \) choose \( r \),” is the number of different ways of choosing \( r \) unique items from a group of \( n \) items. Since \( n \) choose \( r \) can be expressed using factorial, we can write a procedure for this as follows:

```
function choose(n, r, z) {
    var a, b, c;
    fact(n, a);
    n = n - r;
    fact(n, b);
    fact(r, c);
    z = a / (b * c);
}
```

Assume that `fact` uses one of the parameter-passing methods you selected above to compute factorial correctly, and assume that `choose` takes all parameters by reference, so that a call `choose(s, s, t)` results in aliasing between \( n \) and \( r \).

Is there any numeric value of \( s \) such that `choose(s, s, t)` sets \( t \) to \( s_i \) \( \text{choose } s_i \), where \( s_i \) is the initial value of \( s \)? If so, explain which initial values of \( s \) will work. Otherwise, explain why not.
6. Consider the following Javascript code.

```javascript
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.

<table>
<thead>
<tr>
<th>Activation Records</th>
<th>Closures</th>
<th>Compiled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) access link x</td>
<td>(0)</td>
<td></td>
</tr>
<tr>
<td>(2) access link f</td>
<td>( )</td>
<td>((\ ), \bullet )</td>
</tr>
<tr>
<td>(3) access link q</td>
<td>( )</td>
<td>((\ ), \bullet )</td>
</tr>
<tr>
<td>(4) access link q</td>
<td>( )</td>
<td>((\ ), \bullet )</td>
</tr>
<tr>
<td>(5) access link h</td>
<td>( )</td>
<td></td>
</tr>
<tr>
<td>(6) access link w</td>
<td>( )</td>
<td></td>
</tr>
<tr>
<td>(7) access link f</td>
<td>( )</td>
<td></td>
</tr>
<tr>
<td>(8) access link z</td>
<td>( )</td>
<td></td>
</tr>
<tr>
<td>(9) access link y</td>
<td>( )</td>
<td></td>
</tr>
</tbody>
</table>

7. 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:

```javascript
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.

<table>
<thead>
<tr>
<th>Activation Records</th>
<th>Closures</th>
<th>Compiled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td>( 0 )</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>h(2)</td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td>( )</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>f(1)</td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td>( )</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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 f1() to refer to the version of f on line 1 and f3() 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* \( h(2) \).

<table>
<thead>
<tr>
<th>Activation Records</th>
<th>Closures</th>
<th>Compiled Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) access link (0)</td>
<td></td>
<td>A(( ), ⋆) [function, line 1]</td>
</tr>
<tr>
<td>f ⋆</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h ⋆</td>
<td></td>
<td>B(( ), ⋆) [function, line 3]</td>
</tr>
<tr>
<td>(2) access link ( )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g ⋆</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(3) h(2) access link ( )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x ⋆</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(4) g(1) access link ( )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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