CS1101S Notes

Contents

1 General Tips

1. When understanding a program, think from the highest possible level of abstraction. If you have truly understood what a function does, you should be able to explain it clearly and succinctly in a single sentence. Don't get bogged down by the details.
2. Avoid complicating your programs unnecessarily. Elegance lies in simplicity. Make use of existing abstractions like map, filter, accumulate, build_list, enum_list etc.
3. Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away - Antoine de Saint-Exupery
4. Use wishful thinking and recursion wherever possible. Ask yourself: If I had a function that could solve a smaller sub-problem, how would I use that to solve this problem?
5. There are two steps to programming a solution to a problem - solving the problem and writing the program. They must be done in that sequence. Solve the problem on paper before attempting to write the code. Don't even think of programming if you've no clue what to do
6. Think carefully while programming to avoid making careless mistakes. It's easy to make mistakes while typing if you don't stay alert and pay attention to what you're typing.
7. Use comments judiciously - explain the logic of the program but there's no need to explain each line if it is obvious.

2 Important Programs

2.1 List Processing

1. Permutations of a list

```
function permutations(xs) {
    if (is_null(xs)) {
        return list(null);
    } else {
        return accumulate(append,null,
            map(x =>
                   map(y => pair(x,y),permutations(remove(x,xs))),
               xs);
    }
}
```

2. Choosing k elements from a list

```
function combinations(xs,k) {
   if (k === 0) {
      return list(null);
   } else if (is_null(xs)) {
      return null;
   } else {
      const A = combinations(xs,k);
      const B = combinations(xs,k-1);
      const C = map(x => pair(head(xs),x), B);
      return append(A,C);
   }
 }
```

3. Subsets of a list

```
function subsets(xs) {
   if (is_null(xs)) {
      return list(null);
   } else {
      const A = subsets(tail(xs));
      const B = map(x => pair(head(xs),x), A);
      return append(A,B);
   }
 }
```

4. Mapping over a tree

```
function tree_map(f,tree) {
   if (is_null(tree)) {
      return null;
   } else {
      return !is_pair(head(tree))
         ?  pair(f(head(tree)),tree_map(f,tail(tree))
         :  pair(tree_map(f,head(tree)),tree_map(f,tail(tree)));
   }
 }
```

2.2 Stream Processing

1. Interweaving two infinite streams

```
function interweave(s1,s2) {
   return pair(head(s1), () => interweave(s2,stream_tail(s1));
}
```

2. Sieve of Eratosthenes (Stream of Primes)

```
function sieve(s) {
   return pair(head(s),
       () => sieve(stream_filter(x=>x%head(s) !== 0,stream_tail(s))));
}
const primes = sieve(integers_from(2));
```

2.3 Array Processing

1. Insertion sort for Arrays

```
function insertion_sort(A) {
   const len = array_length(A);
   for (let i = 0; i < len; i = i + 1) {
      const x = A[i];
      let j = i - 1;
      while (j >= 0 && A[j] > x) {
         A[j+1] = A[j];
         j = j - 1;
      }
      A[j +1] = x;
   }
 }
```

2. Reversing an array in place

```
function reverse(A) {
   const len = array_length(A);
   for (let i = 0; i < len/2; i = i + 1) {
      const temp = A[i];
      A[i] = A[len-i-1];
      A[len-i-1] = temp;
   }
 }
```

2.4 Miscellaneous

1. N-choose-k algorithm

```
function choose(n,k) {
    return n < 0 || n < k
        ?  0
        :  n === k
        ?  1
        :  choose(n-1,k) + choose(n-1,k-1);
 }
```

2. Implementing accumulate using continuation passing style (CPS)

```
function accumulate_cps(f,init,xs) {
    function acc(ys,cont) {
        return is_null(ys)
            ?  cont(init)
            :  acc(tail(ys), x => cont(f(head(ys),x)));
    }
    return acc(xs, x => x);
 }
```

3 Important Concepts

3.1 Identity: ===

1. True, false, null, undefined - each is identical to itself and nothing else.
2. Two numbers are identical if they have the same representation in the double precision floating point representation.
3. Two strings are identical if they have the same characters in the same order.
4. Functions are made by function expressions, and their creation bestows an identity upon them.
5. Pairs are made by the pair function, and their creation bestows an identity upon them.

3.2 Environment Model

1. Compound objects like user-declared functions, pairs, arrays, etc. are drawn outside the frames. Primitive data like numbers, strings, booleans, etc. are written inside the frame.

2. No empty frame is created.

3. To evaluate a function declaration, two circles are drawn - the left one points to the text of the function body (and parameters), while the right one points to the frame in which the function was declared.

4. To evaluate a function application, the function expression and arguments are evaluated in the current environment, a parameter frame is created that extends the function's environment, and the body block of a function is evaluated in a new frame that extends the parameter frame.

5. Each time the body of a while loop is run, a new frame is created if there are constant declarations made in the body.

3.3 Metacircular Evaluator

1. `evaluate(component, environment)` - evaluates the component with respect to the given environment. A component is either a statement or an expression.

2. `apply(fun, args)` - if the function is primitive, applies the function to the arguments. Otherwise, it evaluates the function body by extending the function's environment.

3. `list_of_values(args)` - evaluates each of the arguments passed to the function (before it is passed to apply, obeying applicative order reduction).

4. `extend_environment(symbols,vals,base_env)` - Creates a new frame by extending the base environment, which contains a list of symbols and a list of associated values.

5. We represent an environment as a list of frames. Each frame is a pair of lists - the first list contains the names of the declarations in that frame, and the second list contains the values associated with the names declared in that frame (in the same order)

6. Operator combinations are converted to function applications to be evaluated. Function declarations are converted to constant declarations whose values are lambda expressions.

7. Our MCE cannot handle arrays, for loops, while loops (although we discussed how to evaluate while loops and for loops), break and continue statements.

8. Our MCE does not distinguish between variables and constants. In particular, it allows assignment to constants, quite erroneously.

9. Our MCE does not detect undeclared names in statements that are not evaluated (for example, if the consequent of a conditional contains an undeclared name but the predicate evaluates to false, our MCE will not complain). The Source Academy, on the other hand, gives error messages for any name that is not declared in the program.

10. Our MCE does not support logical composition operators like `&&` and `||`. We can easily modify this by converting them to conditional expressions of the form `predicate ? consequent : alternative`. (Recall that `x && y` is identical to `x ? y : false`, and `x || y` is identical to `x ? true : y`.

11. When we transform our MCE to lazily evaluate the program (normal order reduction), we have to create objects called "thunks", which represent the expressions that have not been evaluated. We need to keep track of the environment in which these thunks need to be evaluated at some point. Once they have been evaluated, we can choose to memoize the result so that we don't need to evaluate the thunks again.

4 Some other Useful Facts

1. Source does not hoist functions to the top of the block in which they are declared. This is because function declaration is treated identical to a constant declaration.

2. All pre-declared list processing functions give rise to iterative processes since they are implemented using Continuation Passing Style.

3. In JavaScript, there is a critical difference between for loops and while loops: In case of for loops, each iteration of the body has its own "copy" of the loop control variable, which is not the case in while loops.

4. The `equal(x,y)` function does not use === to check for equality of pairs. It returns true if both the arguments have the same structure with respect to pair, and identical values at corresponding leaf positions (that are not themselves pairs). Strings, numbers, boolean, undefined, null etc. are directly compared using ===. Arrays cannot be compared using `equal(x,y)`.

5. `member(v,xs)` uses === to check if v occurs in the list xs. It does not use `equal(x,y)`.

6. Assignment and look-up takes constant time for arrays. Access of an array with an array index to which no prior assignment has been made on the array returns `undefined`.

7. Some stream functions are partially lazy. For example, `stream_filter(pred,s)` is lazy only after it has found the first element in the stream that satisfies the predicate.