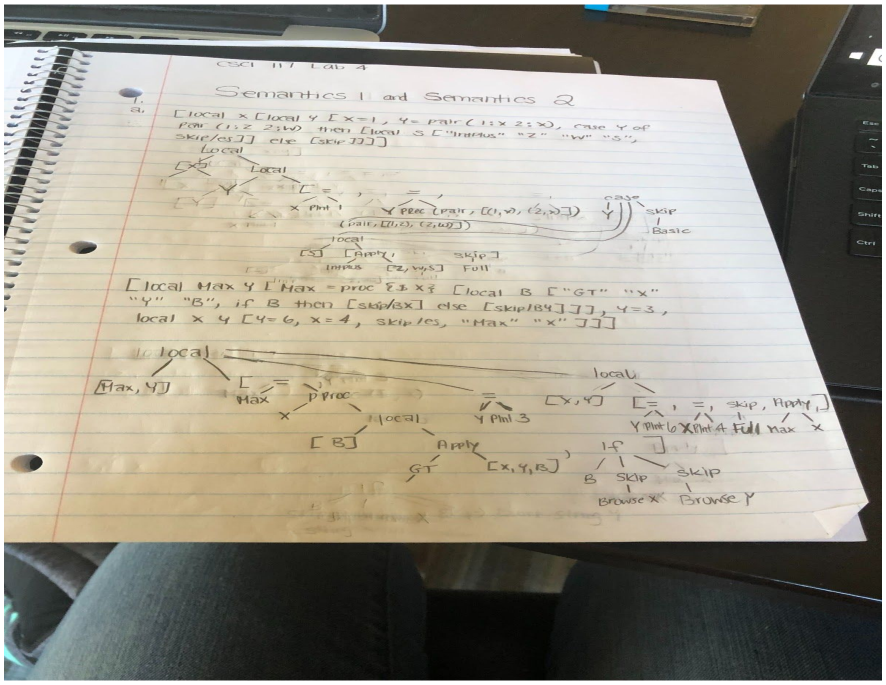
Madhuri Pyreddy
CSCI 117
9/19/19

CSCI 117 Lab 4

1a.



CSCI 117 Lab 4

Semantics 1 and Semantics 2

1.
a.

```
[local x [local y [x=1, y= pair(1:x 2:x), case y of
pair (1:z 2:w) then [local s ["IntPlus" "z" "w" "s",
skip/es]] else [skip]]]]
```

```
[local Max y ["Max =proc {$ x} [local B ["GT" "x"
"y" "B", if B then [skip/Bx] else [skip/By]]], y=3,
local x y [y=6, x=4, skip/es, "Max" "x" ]]]
```

1b. For Semantics 1, we have the output:
*Main> main' "Sem1.txt" "Sem1Out.txt"
Store : ((7), 2), ((6), pair(1:5 2:5)), ((5), 1), ((1), Primitive Operation), ((2), Primitive Operation), ((3), Primitive Operation), ((4), Primitive Operation)
Current Environment : ("S" -> 7, "Z" -> 5, "W" -> 5, "Y" -> 6, "X" -> 5, "IntPlus" -> 1, "IntMinus" -> 2, "Eq" -> 3, "GT" -> 4)
Stack : "".
Semantics 1 file is a case statement. The case statement is used for pattern matching with the record list variable. If it matches, it moves on to statement one <s1> and then to statement two <s2>. It continues with the rest of the execution statements in the stack until the variables are not bound with what we are matching. Once that happens, it should give an error. For Semantics One File, it will bind the unbound variable X to the integer 1, and add information to the store. For Y,

however, it will bind the unbound pair of Z and W to the integers 1 and 2. Then comes S, which is sequence. In sequence, it will bind to Z, W, and S and skips. It ends the program.

For Semantics 2, we have the output:

```
*Main> main' “Sem2.txt” “Sem2Out.txt”
Store : ((7), 4), ((8), 6), ((5), proc), ((6), 3), ((1), Primitive Operation), ((2), Primitive
Operation), ((3), Primitive Operation), ((4), Primitive Operation)
Current Environment : ("X" -> 7, "Y" -> 8, "Max" -> 5, "Y" -> 6, "IntPlus" -> 1, "IntMinus" ->
2, "Eq" -> 3, "GT" -> 4) Stack : "
\"Max\" \"X\"".
X : 4.
```

Semantics 2 file has a procedure. The procedure is part of the syntax statement that includes X1 and Xn as variables and S as a sequence of statements (the body of the procedure). Once such a procedure value is assigned to a variable, then that variable can be used in a procedure call. It assigns Y as a max integer and then starts the procedure for Max Y. It later assigns B as the boolean for integers X and Y. The reason is that it will determine if X is greater than Y. If it is true, then it will skip X. If not, it will skip Y. It will bind the unbound variable Y to the integer 3. It laters assigns X and Y as the integers equal to 6 and 4. It will determine to see if X is greater than Y. If it is, it will declare X as a maximum.

For Semantics 3, we have the output:

```
*Main> main' "Sem3.txt" "Sem3Out.txt"
Store : ((15, 16), spaghetti()), ((11, 12), 12), ((7, 8), kid(age:11 food:15)), ((17, 18), Unbound),
((13, 14), 1978), ((9, 10), 54), ((5, 6), person(age:9 dob:13 food:17 kid:7)), ((1), Primitive
Operation), ((2), Primitive Operation), ((3), Primitive Operation), ((4), Primitive Operation)
Current Environment : ("P1" -> 5, "P2" -> 6, "K1" -> 7, "K2" -> 8, "A1" -> 9, "A2" -> 10, "A3"
-> 11, "A4" -> 12, "DB1" -> 13, "DB2" -> 14, "F1" -> 15, "F2" -> 16, "F3" -> 17, "F4" -> 18,
"IntPlus" -> 1, "IntMinus" -> 2, "Eq" -> 3, "GT" -> 4)
Stack : ""
```

Semantics 3 file has a unification. In unification, it will assign P1, P2, K1, K2, A1, A2, A3, A4, DB1, DB2, F1, F2, F3 and F4. It unifies these variables and sets it equal to other variables like kid, age, food, and dob. It assigns integers for those variables.

2a. //Iterative form in Haskell fib

```
//fib n = fib_it n (1,1) where
//fib_it n (a,b)
//|(n == 0) = a
//| otherwise = fib_it (n-1) (b, a+b)

// Rewrite fib in the Oz syntax below
```

```
local Fib X Result FibIt Zero One in
  Zero = 0
  One = 1
  FibIt = proc {$ In A B Out}
    local B0 B1 in
      {Eq In Zero B0}
      {Eq In One B1}
      if B0 then
        Out = A
      else
          if B1 then
           Out = B
           skip Stack
          else
           local J SUM in
           {IntMinus In One J}
           {IntPlus A B SUM}
           {FibIt J B SUM Out}
          end
      end
    end
    end
  end
  X = 4
  Fib = proc {$ X Result}
    {FibIt X One One Result}
  end
  {Fib X Result}
  skip Browse Result
end
```

2b. There are more stacks in the recursive form than in the iterative form. That is because in recursive form of Fibonacci, the Oz code is calling itself to repeat the code, which causes the stacks to build up when running the code. However, in the iterative form, the stack in the iterative form of Fibonacci does not build up because the iterative form loops only once to repeat part of the code. As a result, the iterative form for Fibonacci has only one stack.