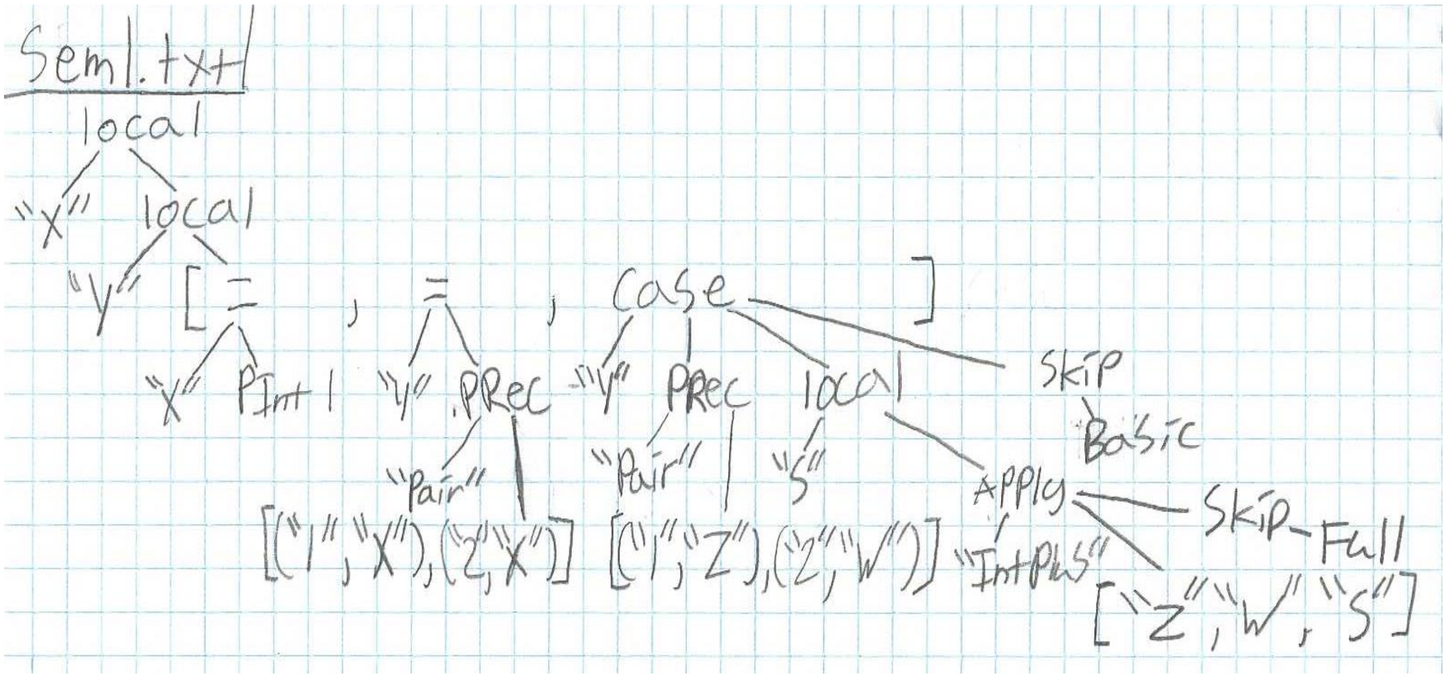
Eric Smrkovsky
Lab 4
CSCI 117
9/14/19

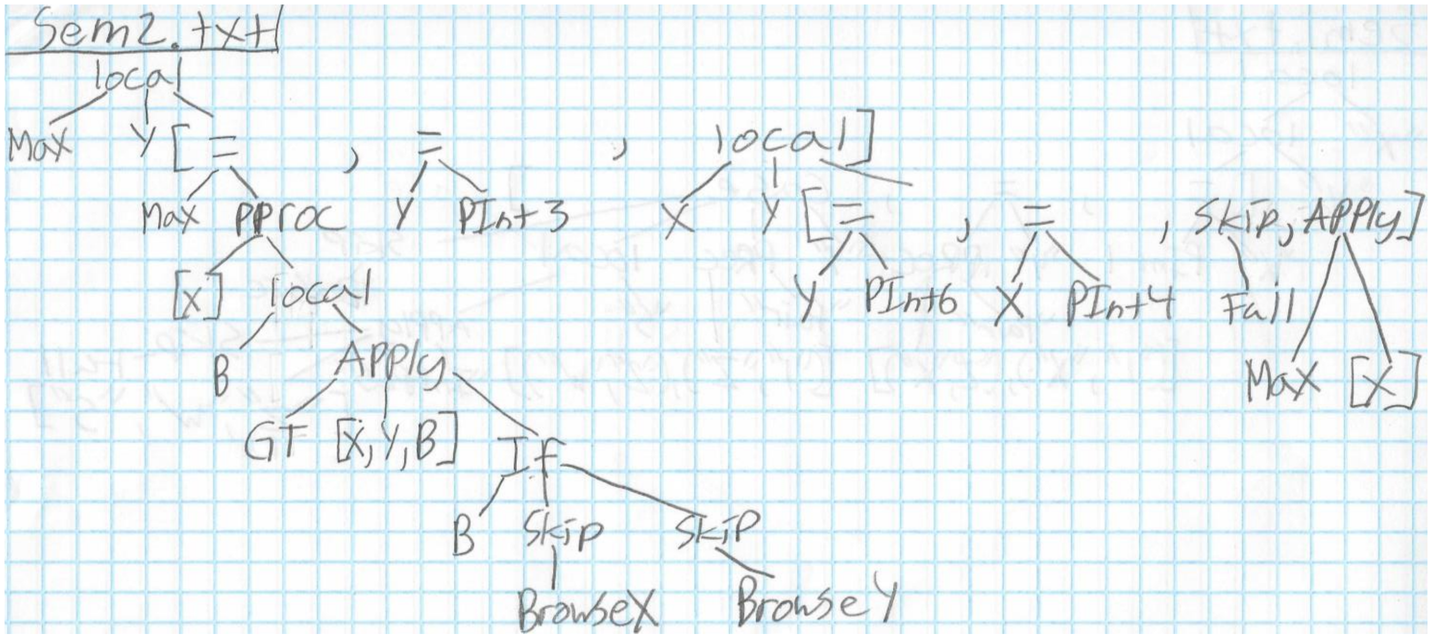
(1a) Sem1.txt

[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]]]]



(1a) Sem2.txt

[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"]]]



Eric Smrkovsky
Lab 4
CSCI 117
9/14/19

(1b) Sem1.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 : ""

A **Case** statement is used to see **if** a variable(**x**) can be matched to a pattern, if it can it does and executes the code after the **then** otherwise it executes the code after the **else**.

The **variables** in the program Sem1.txt relate to the concept of case as follows:

This Oz program creates two variables “X” and “Y” in the environment and assigns them store locations 5 and 6 respectively. The program then binds the value “1” to the location “X” points to in the store (5) and then binds a record (pair(1:5 2:5)) to the location “Y” points to in the store(6) . At this point the program executes a **case** statement which pattern matches “Y” with “pair(1:Z 2:W)”, checking to see that they have the lame label and features. Since the two records can match “Z” and “W” are each assigned the same store location as “X” (5). Since this worked the statement after the **then** executes.

The **then** part creates a new variable “S” and assigns it a store location(7). Next a procedure is executed called IntPlus which adds “Z” and “W” binding the value to “S”. (Actually, for each variable it gets the store location, goes into the store grabs the numbers that are in the store, adds them then places that sum in the store location that is assigned to S.

(1b) Sem2.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\""

Scoping : When a variable is introduced with the local statement an identifier for it is produced but only refers to the variable within the *scope of the identifier* which is between the *local* and the *end*. If the identifier is not bound it is considered a free identifier but will cause the statement that contains it not to run. If the variable is declared in a statement that surrounds a procedure declaration, that procedure declaration is within the scope of the variable and the variable can be used, I believe this would be considered lexical scoping as well as an external reference for the procedure.

Eric Smrkovsky
Lab 4
CSCI 117
9/14/19

The **variables** in the program Sem2.txt relate to the concept of scoping as follows:

The first local statement in the program declares two variables "Max" and "Y" and they are defined within the *local* and the *end* so they are considered lexically scoped. The next line of the program contains a procedure definition which is bound to the store location that is assigned to "Max," within this definition another variable "B" is defined but is only used within a small local statement. Within that scope "B" is used in an **if** statement to be assigned as a return value after a procedure is executed to determine if X is greater than Y. The "Y" here is an external reference because it is defined outside of the procedure definition. The "X" here is not defined in the procedure definition either but is passed by reference during the procedure call, so it is able to be used when it is called with the procedure. The remaining variable "Y" which has the store location of (8) is never used because it is not defined within an environment where the procedure is declared.

Closure : A closure is a record that stores a procedure definition and the environment that exists when the procedure is declared.

The **variables** in the program Sem2.txt relate to the concept of closure as follows:

Within the "Max" procedure definition the "X" is passed into the procedure when it is called as a formal parameter. The closure occurs because the "Y = 3" is declared within the environment that the "Max" procedure is declared. This "Y" is considered an external reference of the procedure.

<u>(1b) Sem3.txt</u>

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 : ""

Unification can be thought of as an "operation that adds information to the SAS" (pg 101, CTM). Binding a variable to a value, a list of values or a procedure is also considered unification. So what happens during unification of a variable to variable binding is we look at the store locations of the variable names and bind those two store locations together and this depends on the state of the variable. Everything is fine when we try and bind variables that are unbound, it becomes more complicated when we need to bind variables that have store locations with values within. Since a variable can only be bound to one thing we check to see if the values are the same, if they are we place them in a variable group, if the values don't match then we have an error.

The **variables** in the program Sem3.txt relate to the concept of unification as follows:

(5)First of all, the first line of code (line 5) in this oz program declares the variables: P1 P2 K1 K2 A1 A2 A3 A4 DB1 DB2 F1 F2 F3 F4 and they all start out as unbound but they have been given the store locations 5 through 18 in the environment.

(6)Next, the program creates a record named person and binds it to the store location of the variable P1 which is 5. Within this record age is age, dob, food and kid are assigned the store locations 9, 13, 17, and 7 respectively. This is a successful unification.

(7) Next, the program creates a record named person and binds it to the store location of the variable P2 which is 6. Within this record age is age, dob, food and kid are assigned the store locations 10, 14, 18 and 8 respectively. This is a successful unification.

(8)Next, the program creates a record named kid and binds it to the store location of the variable K1 which is 7. Within this record age is age and food are assigned the store locations 11 and 15 respectively. This is a successful unification.

(9)Next, the program creates a record named kid and binds it to the store location of the variable K2 which is 8. Within this record age is age and food are assigned the store locations 12 and 16 respectively. This is a successful unification.

(10)Next store location of the variable A3 (11) has the value of 12 assigned to it. This is a successful unification.

(11)Next the store location of the variable F2 (16) has the record named spaghetti (spaghetti is an empty record at this point) assigned to it. This is a successful unification.

(12)Next store location of the variable A1 (9) has the value of 54 assigned to it. This is a successful unification.

(13) Next store location of the variable DB2 (14) has the value of 1978 assigned to it. This is a successful unification.

Finally in the last line of code the store location within the variables P1 and P2 are attempted to be unified through a variable to variable binding. Up until this point all of the store locations have been unbound when values were assigned.

Eric Smrkovsky
Lab 4
CSCI 117
9/14/19

So this is the complicated part of unification that Dr. Wilson described in his lecture.

P1 = P2

person(kid:K1 age:A1 dob:DB1 food:F3) = person(kid:K2 age:A2 dob:DB2 food:F4)

So when these two records are unified the store locations that math in each record are placed into the same variable group. Now each K1 and K2 are bound together in the same store location, since they are records, they also bind the variables within it's list to each other because they match as well(A3 and A4, F1 and F2). The same thing happens with A1 and A2, DB1 and DB2 as well as F1 and F4. Interestingly, the person doesn't have a favorite food, but his child likes spaghetti.

(2a)

```
//Haskell iterative version of fib

fib:: Int -> Int

fib x = fib_it x 1 1 where

  fib_it :: Int -> Int -> Int -> Int

  fib_it 0 a b = a

  fib_it 1 a b = b

  fib_it n a b = fib_it (n-1) b (a+b)


// Rewrite fib in the Oz syntax below


local Fib Zero One X Result in
```

Eric Smrkovsky
Lab 4
CSCI 117
9/14/19

```
Fib = proc {$ In Out}              //begin fib def
 local FibIt N A B in              //FibIt var declarations
  FibIt = proc {$ N A B Out}       //begin Fibit def
  Zero = 0                         //var for base case
  One = 1                          //var for base case and IntMinus call
  local B0 B1 in                   //declare vars for base case checks
   {Eq N Zero B0}                  //is input zero? B0 is a bool
   {Eq N One B1}                   //is input one? B1 is a bool
   if B0 then                      //check condition
      Out = A                      //base case 1
    skip Browse Out
    skip Full                      //displays store/env/stack
   else                            //if condition fails
    if B1 then                     //check another condition
     Out = B                       //base case 2
     skip Browse Out
     skip Full                     //displays store/env/stack
    else                           //previous two conditions failed
     local AB N1 in                //declare vars for recursive call
      {IntMinus N One N1}          //subtract One from input N1=N-1
      {IntPlus A B AB}             //add A and B from input A+B=AB
      {FibIt N1 B AB Out}          //recursive call
      end
     end
    end
   end
```

```
     end
     N = In                  //N becomes the input
     {FibIt N One One Out}     //starts recursive algorithm defined above
   end
 end
 X = 2                //value for fib input
 {Fib X Result}        //starts code defined above
 skip Browse Result     //displays answer
end
```

(2b)

For both files what is left in the stack when the base case code is ran is the last thing that the program has to do which is to Browse the Result. This is the final step of the recursive algorithm in both cases. So after the base case is reached, the programs only have one statement remaining in the stack.

Now, when trying the programs with a value of 2 for X the iterative version of Fib still only has the same Skip Browse Result left in the stack, this is because the stack is only going to have things to do within the final else of the if statement... It is a very different situation for the standard version.

In the standard version in the final else the fib procedure calls itself twice and when one of the base cases is reached it still has things to do in the stack because Fib has either been called once and will be called again, or Fib only needs to be called once more.

Another thing to notice is that the iterative version only will reach one of the base cases once. The standard version will reach a base case multiple times. So every time that the standard version reaches a base case, it displays all of the statements that are left for the program to accomplish, the first in the list being the next thing and the last is when the result is to be displayed.

Another thing I have noticed is that the standard program will display the stack and a base case the exact amount of times as the result. The stack size grows with the size of the value that Fib is called with! This standard version also creates a huge store! The iterative version doesn’t do this. I guess this could be another way to predict how fast these programs are when compared to each other.

Eric Smrkovsky
Lab 4
CSCI 117
9/14/19