CSCC24 Week 12 Notes

Semantics I: Expressions, Bindings, Functions:

- Is used to implement interpreters.
- **Definition: static X** means doing X by analysing the code, without needing to run it and wait, while **dynamic X** means doing X while running the code.
- For a basic setup, I will start with dynamic checking of types and dynamic checking that variables exist when they are used.

Setup:

- I will have several kinds of run-time errors such Type errors, variable not found, and if you support division, you will also have division by zero.
- I use the Either monad to represent this possibility.
- We should use an algebraic data type for error messages, especially if later you support catching and handling errors.
- Here's the data type for my interpreter:

  ```
  mainInterp :: Expr -> Either String Value
  ```

  Note: For simplicity, the professor used String for error messages. We should use an algebraic data type.
  If we get an error, we get something of type String.
  Otherwise, we get something of type Value.
- We can use a monad to model the fact that my language has the effect of errors/exceptions.
- I have a function, raise, for raising errors. It is defined to be simply Left in this lecture but it won't be that easy in a more featureful interpreter for a more complex language, such as stateful languages.
  Here's the code for raise:

  ```
  raise :: String -> Either String a
  raise = Left
  ```
- I will also be passing around a dictionary that maps variables to values. So I need this:

  ```
  mainInterp expr = interp expr Map.empty
  interp :: Expr -> Map String Value -> Either String Value
  ```

  The recursion happens in interp, which is a helper interpreter.
  Map String Value is used to map variable names to Values.
  Now I will implement interp for each construct.

Basic constructs:

- This language has number literals, boolean literals, and binary operators:

  ```
  data Expr = Num Integer
          | Bln Bool
          | Prim2 Op2 Expr Expr        -- Prim2 op operand operand
          | ...

  data Op2 = Eq | Plus | Mul
  ```
- I will be evaluating them and more to number values, boolean values, and later another kind of values.
- A clean habit is not to re-use the abstract syntax tree type, but to define a separate type, since the values have much fewer possibilities, and some possibilities will not correspond well to any abstract syntax tree.

E.g.

```
data Value = VN Integer
           | VB Bool
```

- Here is how I evaluate a number literal. Boolean literal is similar.

```
interp (Num i) _ = pure (VN i)
```

Arithmetic (all operands evaluated):

- Here is how I evaluate addition; most other arithmetic operators are similar. The insight is to use structural recursion to evaluate the operands, then you will have number values to add. The annoying part is to check that the values are actually numbers, so I re-factor out the checking to a helper function.
- Here's the code:

```
interp (Prim2 Plus e1 e2) env =
   interp e1 env
   >>= \a -> intOrDie a
   >>= \i -> interp e2 env
   >>= \b -> intOrDie b
   >>= \j -> return (VN (i+j))

intOrDie :: Value -> Either String Integer
intOrDie (VN i) = pure i
intOrDie _ = raise "type error"
```

Note: env is a dictionary.
Note: When checking division, we have to check for division by 0 by checking if the denominator is 0.

Short-circuiting, conditionals (operands selectively evaluated):

- I have an if-then-else:

```
data Expr = ...
          | Cond Expr Expr Expr      -- Cond test then-branch else-branch
```

If test is true, then we evaluate the "then-branch" only.
Otherwise, we evaluate the "else-branch" only.

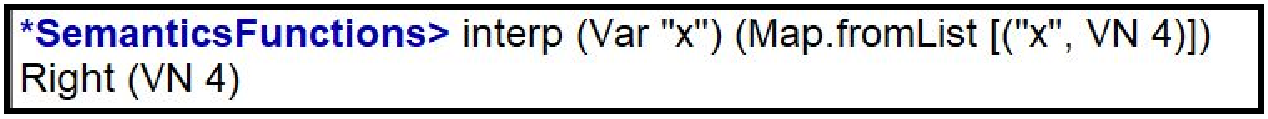
This is a short-circuiting operator: some operands are selectively evaluated, others skipped.
Here is the code:

```
interp (Cond test eThen eElse) env =
   interp test env
   >>= \a -> case a of
     VB True -> interp eThen env
     VB False -> interp eElse env
     _ -> raise "type error"
```

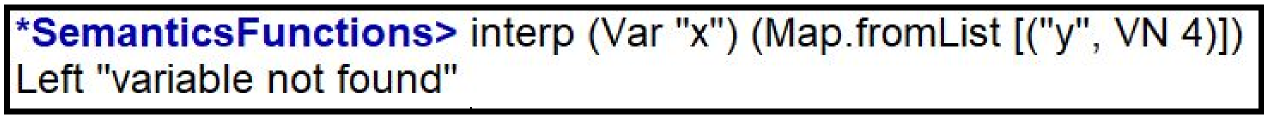
You can add short-circuiting logical operators, and, or, and their semantics will be similar.

Variables, local bindings, environments (scopes):

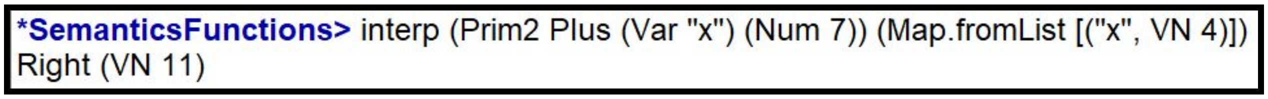
- Here's the data type:
  ```
  data Expr = ...
          | Var String
  ```
  However, where do we get the contents of variables from?
  A nice solution is to maintain a dictionary that maps variables to contents, so we can just look up.
  This dictionary is called **environment** and mapping a variable to its content is called **binding**.
  Also, the nature of the contents depend on the evaluation strategy of the language. For example, call by value just needs values, while lazy evaluation needs something more complex to cater for partly evaluated, partly unevaluated expressions.
- We will use Data.Map for dictionaries. Practical interpreters use hash tables and compilers use an array because they compile variable names to addresses.
- To evaluate a variable, we just look it up.
  If we find it, we return it.
  Otherwise, we raise an exception.
- Here's the code:
  ```
  interp (Var v) env = case Map.lookup v env of
    Just a -> pure a
    Nothing -> raise "variable not found"
  ```
- E.g.



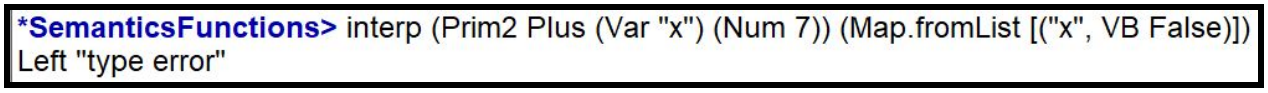
```
*SemanticsFunctions> interp (Var "x") (Map.fromList [("x", VN 4)])
Right (VN 4)
```



```
*SemanticsFunctions> interp (Var "x") (Map.fromList [("y", VN 4)])
Left "variable not found"
```



```
*SemanticsFunctions> interp (Prim2 Plus (Var "x") (Num 7)) (Map.fromList [("x", VN 4)])
Right (VN 11)
```



```
*SemanticsFunctions> interp (Prim2 Plus (Var "x") (Num 7)) (Map.fromList [("x", VB False)])
Left "type error"
```

- My local binding construct wraps an expression inside a new local context of 0 or more "name = expr" bindings.
  Here's the data type:
  ```
  data Expr = ...
          | Let [(String, Expr)] Expr   -- Let [(name, rhs), ...] eval-me
  ```
  E.g.
  let { x=1; y=0 } in x+y is represented as
  Let [("x", Num 1), ("y", Num 0)] (Prim2 Plus (Var "x") (Var "y")).

- There are a number of decisions to make about the semantics of the local binding construct. It is also possible to offer many different local binding constructs, one for each way of making these decisions.
    1. **Scoping and recursion:**
    Suppose you have let { x=2+3; y=x+4; } in ….
        a. Here, it is equivalent to let { x=2+3 } in let { y=x+4 } in …
        So for y=x+4, we use the x in x=2+3.
        This is called **sequential binding**.
        If you choose sequential binding, right after you process one equation, you have to extend the environment to include its new binding, under which you process the remaining equations and eventually the wrapped expression.
        b. An alternative is that y=x+4 uses an outer x.
        This is called **parallel binding**.
        Note: This does not always imply parallel computing. It only implies semantic independence.

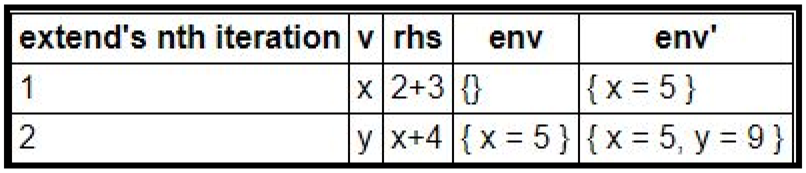
    Note: Those two choices don't support mutual recursion. The third alternative supports mutual recursion, so every equation may use every variable defined in the same group. If you do call by value, then you also need to place restrictions on the RHSes.
    2. **Evaluation strategy:**
    The choices are call by value, lazy evaluation, and call by name.
    With call by value, first go through the equations in the given order, evaluate the RHS of each one right away, and lastly evaluate the wrapped expression.
    E.g.
    I will evaluate 2+3 and store the result in x.
    Then, I will evaluate x+4, where x=5, and store the result in y.
    Then, I will evaluate the stuff after "in".
- Here's the code. This is for call by value.

```
interp (Let eqns evalMe) env =
  extend eqns env
  >>= \env' -> interp evalMe env'
  -- Example:
  --    let x=2+3; y=x+4 in x+y
  -- -> x+y   (with x=5, y=9 in the larger environment env')
  -- "extend env eqns" builds env'
 where
  extend [] env = return env
  extend ((v,rhs) : eqns) env =
    interp rhs env
    >>= \a -> let env' = Map.insert v a env
              in extend eqns env'
```

For extend, if the list is empty, we give back the environment
Otherwise, for each tuple in the list, "v" is the variable and "rhs" is what the variable is set to.
Since this is call by value, right away, we evaluate RHS under the given environment. This is the line "interp rhs env".

“interp rhs env” gives back some value. We bind that value with the lambda function. The lambda function puts the variable and the evaluated rhs into the original environment and then it makes a recursive call to process the rest of the equation.

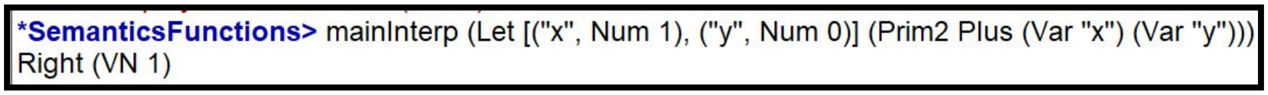
E.g. Working on let { x=2+3; y=x+4; } in x+y, we get:



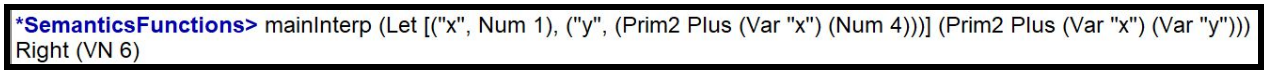
extend's nth iteration	v	rhs	env	env'
1	x	2+3	{}	{ x = 5 }
2	y	x+4	{ x = 5 }	{ x = 5, y = 9 }

Now we're ready to evaluate x+y under {x = 5, y = 9}.
However these are local variables unknown to the outside.
Suppose I have (let {x=2+3; y=x+4;} in x+y) * (1+1).
I make a recursive call to handle the “let-in”. Inside that recursive call the new environment {x = 5, y = 9} is built for internal use, but not returned or passed back to the outside. The outside still uses the outside environment for “1+1”.

- E.g.

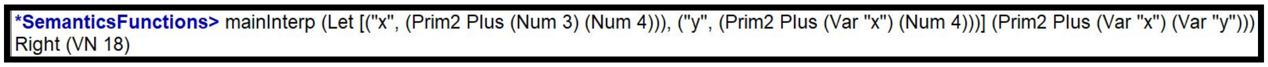


```
*SemanticsFunctions> mainInterp (Let [("x", Num 1), ("y", Num 0)] (Prim2 Plus (Var "x") (Var "y")))
Right (VN 1)
```



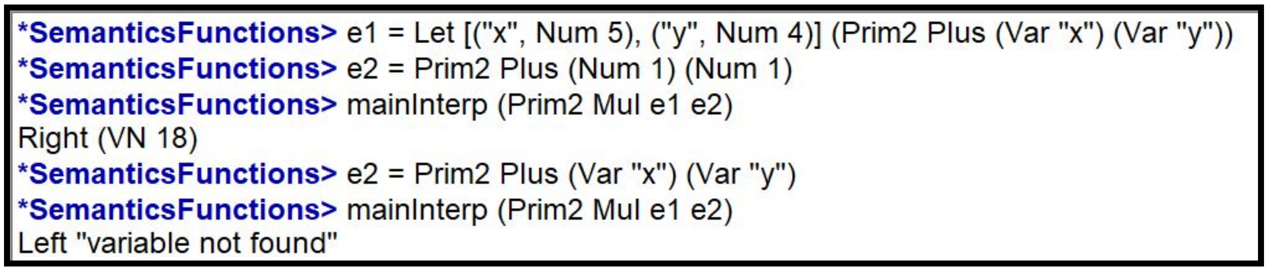
```
*SemanticsFunctions> mainInterp (Let [("x", Num 1), ("y", (Prim2 Plus (Var "x") (Num 4)))] (Prim2 Plus (Var "x") (Var "y")))
Right (VN 6)
```

Here:
x = 1
y = x + 4 = 1 + 4 = 5
x + y = 1 + 5 = 6



```
*SemanticsFunctions> mainInterp (Let [("x", (Prim2 Plus (Num 3) (Num 4))), ("y", (Prim2 Plus (Var "x") (Num 4)))] (Prim2 Plus (Var "x") (Var "y")))
Right (VN 18)
```

Here:
x = 3 + 4 = 7
y = x + 4 = 7 + 4 = 11
x + y = 7 + 11 = 18.



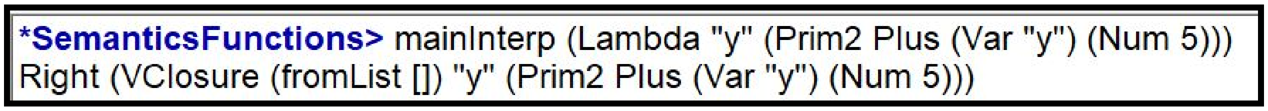
```
*SemanticsFunctions> e1 = Let [("x", Num 5), ("y", Num 4)] (Prim2 Plus (Var "x") (Var "y"))
*SemanticsFunctions> e2 = Prim2 Plus (Num 1) (Num 1)
*SemanticsFunctions> mainInterp (Prim2 Mul e1 e2)
Right (VN 18)
*SemanticsFunctions> e2 = Prim2 Plus (Var "x") (Var "y")
*SemanticsFunctions> mainInterp (Prim2 Mul e1 e2)
Left "variable not found"
```

In the first example, e1 = 5+4 = 9 and e2 = 1+1 = 2. e1 * e2 = 18.
In the second example, e2 = Prim2 Plus (Var "x") (Var "y"). However, when we do mainInterp (Prim2 Mul e1 e2), the x and y in e1 are not shown to e2. Hence, we get the error message “variable not found”.

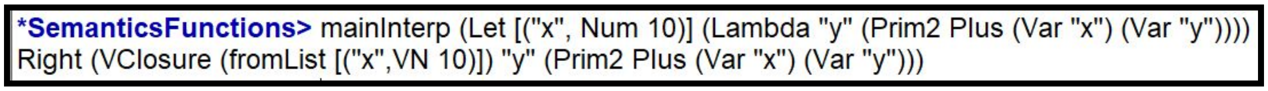
Function construction (lambda), closures:

- For simplicity, I just have a lambda construct for anonymous functions.
  If you want to define a function with a name, use lambda together with let.
- Here's the data type:
  data Expr = ...
  | Lambda String Expr -- Lambda var body
- E.g. \x -> … is represented as Lambda "x" (...).
- Suppose your lambda is **\y->x+y**.
  y is a **bound variable** and x is a **free variable**.
  This also carries to let. In **let y=x+1 in x*y**, y is a bound variable and x is a free variable.
  Bound variable of/in an expression: You can see where the variable is introduced or declared or defined.
  Free variable of/in an expression: You can't see where the variable is introduced or declared or defined. It has to come from the outside.
  E.g. A free variable like "x" is probably bound in an outer context.
- When the interpreter runs into the lambda, and if it already knows x=10 from the outer context, it needs to attach "x=10" to the lambda so it is not forgotten.
- The value after evaluating a lambda needs to remember 3 things
  1. parameter name
  2. function body
  3. the environment in scope for this lambda.
- **Definition:** The combination of "\y->x+y" plus "x=10 from an outer context" is called a **closure**. A **closure** is a record or data structure that stores an expression together with the environment for all of its free variables.
- In this lecture, my only use of closures is for lambdas, so my closure representation is specialized for that purpose only.
- Here's the data type and code:
  data Value = ...
  | VClosure (Map String Value) String Expr
  interp (Lambda v body) env = pure (VClosure env v body)

  (Map String Value) is the environment.
  String is the parameter name(s).
  Expr is the function body/expression.
- E.g.



```
*SemanticsFunctions> mainInterp (Lambda "y" (Prim2 Plus (Var "y") (Num 5)))
Right (VClosure (fromList []) "y" (Prim2 Plus (Var "y") (Num 5)))
```



```
*SemanticsFunctions> mainInterp (Let [("x", Num 10)] (Lambda "y" (Prim2 Plus (Var "x") (Var "y"))))
Right (VClosure (fromList [("x",VN 10)]) "y" (Prim2 Plus (Var "x") (Var "y")))
```

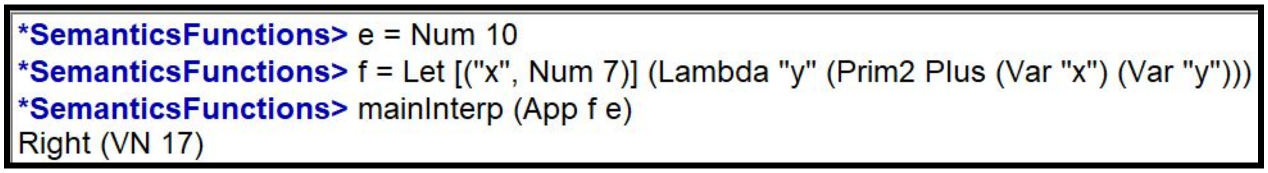
This is equivalent to let {x = 10;} in \y -> x + y

Function application:

- Here's the data type:
```
data Expr = ...
        | App Expr Expr            -- App func param
```
- There is a decision to make about the semantics of function application, namely which evaluation strategy should be used: call by value, lazy evaluation, or call by name. Here, I will do call by value.
  Note: All of them require you to evaluate the function until you get a function closure, sooner or later.
- Since I do call by value, I evaluate the parameter until I get a value. Then, I plug the value in. To plug the value in, just like evaluating let, I can first extend the environment to bind the parameter name to the parameter value, then it makes sense to evaluate the function body under that environment.
- Here's the code:
```
interp (App f e) env =
   interp f env
   >>= \c -> case c of
     VClosure fEnv v body ->
        interp e env
        >>= \eVal -> let bEnv = Map.insert v eVal fEnv  -- fEnv, not env
                  in interp body bEnv
        -- E.g.
        --    (\y -> 10+y) 17
        -- -> 10 + y     (but with y=17 in environment)
        --
```

  E.g. for (let x=7 in \y -> x+y) 10:
  1. interp on the function gives: VClosure {x = VN 7} "y" (x+y)
  2. interp on the "10" gives: VN 10
  3. Now do: interp (x+y) {x = VN 7, y = VN 10}

  E.g.



```
*SemanticsFunctions> e = Num 10
*SemanticsFunctions> f = Let [("x", Num 7)] (Lambda "y" (Prim2 Plus (Var "x") (Var "y")))
*SemanticsFunctions> mainInterp (App f e)
Right (VN 17)
```

- **Dynamic scoping** is when a variable name refers to whoever has that name at the time of evaluation. We shouldn't do this.
  E.g.
  let {x=10; f = \y->x+y;} in
  let {x=5;} in
  f0 → 5+0 (Here, x=5 is used instead of x=10)
- **Lexical/Static scoping** is when a variable name refers to whoever has that name in the code location. We should do this.

Recursion:

- The trick is to take an extra parameter for a function to be called. Then, call a function with itself as a parameter.
- Here's an example of doing factorial recursively.
  Here's a trace of **let mkFac = \f -> \n -> if n=0 then 1 else n * (f f) (n-1) in mkFac mkFac 2**

```
  mkFac mkFac 2
→ (\f -> \n -> if n=0 then 1 else n * (f f) (n-1)) mkFac 2
→ (\n -> if n=0 then 1 else n * (mkFac mkFac) (n-1)) 2
→ if 2=0 then 1 else 2 * (mkFac mkFac) (2-1)
→ 2 * (mkFac mkFac) (2-1)
→ 2 * (\f -> \n -> ...) mkFac (2-1)
→ 2 * (\n -> if n=0 then 1 else n * (mkFac mkFac) (n-1)) (2-1)
→ 2 * (\n -> if n=0 then 1 else n * (mkFac mkFac) (n-1)) 1
→ 2 * if 1=0 then 1 else 1 * (mkFac mkFac) (1-1)
→ 2 * 1 * mkFac mkFac (1-1)
→ 2 * 1 * (\n -> if n=0 then 1 else n * mkFac mkFac (n-1)) (1-1)
→ 2 * 1 * (\n -> if n=0 then 1 else n * mkFac mkFac (n-1)) 0
→ 2 * 1 * if 0=0 then 1 else 0 * ...
→ 2 * 1 * 1
```