CSE 131A
Lecture 15

Bindings and environments
Parameter Matching

Scott B. Baden
Department of Computer Science & Engineering
University of California, San Diego
Winter 2007

FLWR expressions

- In Onyx, a FLWR expression has the form:
  (ForClause | LetClause)+ WhereClause? ReturnStmt
- The result is a a sequence
- Let clauses bind single variables to values
- For clauses iteratively bind single variables to values in a sequence of values
- The return clause specifies an expression to evaluate in the context of current bindings
- A where clause "gates" the evaluation of the return clause
- The result of a FLWR expression is a sequence

For and let clauses

- A let clause can also have multiple variable bindings separated by commas; this is equivalent to a sequence of single variable let clauses

```
let $k := 12, $n := $k, $n:=$n+2
let $k := 12
   let $n := $k
      let $n := $n+2
```

- A for clause can have multiple variable bindings separated by commas; this is equivalent to a sequence of for clauses

For and let clauses

- The FLWR expression

```
let $n := 3 for $i in (3,4), $j in ($i,7)
let $k := 12, $n:=$n+2 where $j < $n return $i * $j
```

is equivalent to

```
let $n := 3
    for $i in (3,4)
        for $j in ($i,7)
            let $k := 12
                let $n := $n+2
                    where $j < $n
                        return $i * $j
```

Bindings in let clauses

- A let clause evaluates the expression on the right hand side of the := in the context of the current bindings..
- … and binds the variable on the left hand side to that value

```
let  $a  :=  3
   let $b := $a + 7
      return 2*$b
```

- This binding has scope until the end of the FLWR expression in which the let clause appears…
- .. but may be hidden by a later binding of the same name let $a := 6

Bindings in for clauses

- A for clause first evaluates the expression on the right hand side of the in in the context of the current bindings
- This generally evaluates to a sequence..
- … the variable on the left hand side is iteratively bound to the values in that sequence
- Each binding has a scope until the end of the FLWR expression in which the for clause appears
- Again, it may be hidden by a later binding of the same name

Evaluating FLWR expressions

- The Onyx AST representation provides separate node types for for and Let expressions
- This simplifies the evaluation procedure
- First, create an empty receptable to hold the result of the evaluation
- Then process as follows

Processing FLWR clauses: let clause

- If the clause to be processed is a let clause…
- Evaluate the expression on the right hand side of the := in the context of the current bindings

```
let $a := 3
```

- Bind the variable on the left hand side to that value
- (Recursively) evaluate the expression in the scope of the new binding
- Pop the binding from the symbol table stack and return

Evaluating a let clause

```
let $a := 4
  let $b := 3
    return $a+$b
```

```
<onyx.ast.Query>
  <onyx.ast.ExprList>
    <onyx.ast.LetExpression name="$a">
      <onyx.ast.Constant datatype="onyx.types.Integer">4</onyx.ast.Constant>
      <onyx.ast.LetExpression name="$b">
        <onyx.ast.Constant datatype="onyx.types.Integer">3</onyx.ast.Constant>
        <onyx.ast.Operator name="op:numeric-add">
          <onyx.ast.Variable>$a</onyx.ast.Variable>
          <onyx.ast.Variable>$b</onyx.ast.Variable>
        </onyx.ast.Operator>
      </onyx.ast.LetExpression>
    </onyx.ast.LetExpression>
  </onyx.ast.ExprList>
</onyx.ast.Query>
```

Processing FLWR clauses: for clause

- If the clause to be processed is a for clause:
- Evaluate the expression on the right hand side of the in in the context of the current bindings. This will evaluate to a sequence for $i in (2 to 7) where $i > 3 return $i
- For each value in that sequence:
  - Bind the variable on the LHS to that value
  - (Recursively) evaluate the body in the context of the new binding
  - Pop this binding from the symbol table stack
- When done with the sequence, return from the for clause

Processing FLWR clauses: where clause

- These are converted to IfThenElse expressions

```
let $a := 2, $b :=7, $c := 3
   for $i in ($a to $b) where $c > 3 return $i
<onyx.ast.ForExpression name="$i">
    <onyx.ast.Operator name="op:to">
        <onyx.ast.Variable>$a</onyx.ast.Variable>
        <onyx.ast.Variable>$b</onyx.ast.Variable>
    </onyx.ast.Operator>
    <onyx.ast.IfThenElseExpr normalized="where">
        <onyx.ast.Operator name="op:greater-than">
            <onyx.ast.Variable>$i</onyx.ast.Variable>
            <onyx.ast.Variable>$c</onyx.ast.Variable>
        </onyx.ast.Operator>
        <onyx.ast.Variable>$i</onyx.ast.Variable>
        <onyx.ast.ExprList/>
    </onyx.ast.IfThenElseExpr>
</onyx.ast.ForExpression>
```

Processing FLWR clauses: return results

- Evaluate the return body expression in the context of the current bindings
- Add this value to the result value list and return
- Remember that sequences must be flattened

Evaluating a FLWR

- So this FLWR expression:

```
let $n := 3
   for $i in (3,4)
      for $j in ($i,7)
         let $k := 12
            let $n := $n+2
               where $j < $n
                  return $i * $j
```

- Evaluates to the sequence `9 16`

How does this work?

- Some helper functions

```
declare function C($i) {concat(string($i),",")};
declare function Bracket($s) {concat(concat("[",$s),"]")};
let $n := 3
for $i in (3,4)
for $j in ($i,7)
let $n := $n+2
return Bracket(concat(concat(C($i),C($j)),string($n)))
```

- Returns the result

<Result>[3,3,5] [3,7,5] [4,4,5] [4,7,5]</Result>

Variables bound in for and let clauses of FLWR expressions

- Any variable bound in a for or let clause is in scope until the end of the FLWR expression in which it is bound
- If the variable name used in the binding was already bound in the current scope, the variable is bound again with a new value
  (we push its value onto the stack for that variable's symbol table entry)
- The variable goes out of scope once the FLWR expression that first bound the variable ends.

Evaluating a FLWR with flattening

- So this FLWR expression:

```
for $i in (3,4)
   for $j in ($i,7)
      let $k := 12
         return ($i, $j, $k)
```

- Has the result value sequence
  ((3,3,12),(3,7,12),(4,4,12),(4,7,12))

- which flattens to (3,3,12,3,7,12,4,4,12,4,7,12)
- which prints as

  3 3 12 3 7 12 4 4 12 4 7 12

Objects and Types

- Every Onyx expression has a value (if evaluating it does not raise an error)
- ...And an Onyx program is just a sequence of expressions
- So an Onyx program has a value: the sequence of values of those expressions that make up the program
- Onyx is a strongly typed language, so every Onyx value has two attributes:
  - the type of the value
  - the "value" of the value
- Values may be bound to variables for convenience: we just use the variable in the program instead of the larger expression

The role of binding

- The purpose of a binding is to establish a connection between an identifier and a value

```
let $i := 3
```

- This binds $i to the value 3 of type Integer
- Languages like C and Java bind storage locations to variable names
- Assignment modifies that bound storage location

```
int i = 3;
```

- In Onyx, a variable is just a synonym for a value, and there is no updatable storage
- Binding simply associates a name with a value

Declarations

- In languages like C and Java, the type of a variable is established in a type declaration, and variable type can be determined at compile time:

```
int i = 3;
```

- In Onyx, type declarations maybe used in global variable or function declarations, but not in let or for expressions

```
declare variable $a as onyx.types.Integer {2};

declare function f($x as onyx.types.Integer) as
                     onyx.types.String
                     {string($x) + "string"};
```

- A lot of variable typing is done dynamically, at evaluation time

The evaluation environment

- An ***evaluation environment*** provides the context for evaluating expressions
- This can include:
  - the bindings of variables
  - the definitions of functions
  - the definitions of types
  - the contents of modules or packages, etc.
- When a name is declared (e.g. when a variable is bound), put its information in the environment
- When the “meaning” of a name is needed, look it up in the environment

More about environments

- Each time we declare a new identifier, we augment the environment
- let $i := 3

i	int	3

More about environments

- Each time we declare a new identifier, we augment the environment

i	int	3
j	int	8

- let $i := 3,
  $j := 8

How about ‘for?’

- We can think of for as inheriting the environment of the enclosing context, that is what was defined before we encountered the for
- Each iteration of for defines a new environment obtained by…
  - augmenting the inherited environment with a binding of the induction variable ..
  - to one of the values defined in the iteration sequence
- Do this for this for each value in the range

```
let $a := 3, $b := 5
   for $i in $a to $b
      return $i
```

Evaluating for

- Consider the following

```
let $a := 3, $b := 5
for $i in $a to $b
   return $i
```

- Initial environment

a	b	
3	5	

Evaluating for

- Consider the following

```
let $a := 3, $b := 5
for $i in $a to $b
   return $i
```

- Iteration 3

a	b	i
3	5	3

Evaluating for

- Consider the following

```
let $a := 3, $b := 5
for $i in $a to $b
   return $i
```

- Iteration 4

a	b	i
3	5	4

Evaluating for

- Consider the following

```
let $a := 3, $b := 5
for $i in $a to $b
   return $i
```

- Iteration 5

a	b	i
3	5	5

Evaluating for

- Consider the following

```
let $a := 3, $b := 5
for $i in $a to $b
   return $i
```

- Exit the for loop

a	b	i
3	5	5

The environment and language rules

- What data structures are best for implementing the evaluation environment?
- It depends on the language's rules for the use of the names in question
- It is important to distinguish between *static* (also known as *lexical*) and *dynamic* rules

Static and dynamic language rules

- Static (lexical) rules: Rules for uses of symbols that can be enforced by the compiler strictly by inspecting the structure of the program, without running it
- Dynamic rules: Rules for uses of symbols that are enforced when the program actually runs or is evaluated

Static and dynamic concepts

Static Concept	**Dynamic Concept**
Defining a function	Calling a function
Declaring (introducing) a variable	Binding a variable to a value
Lexical scope of a variable	Lifetime of a variable binding

The structure of the environment

- To decide how to structure the environment, consider these questions
- Can function declarations contain nested function declarations ?
- Can function declarations contain calls to the function being declared?
- Are function arguments passed by reference or value?
- Can a function definition contain references to names not declared in the function? If so what is their meaning?
- What happens to names declared in a function when the function returns?
- Can a block contain references to names not declared in the block? If so what is their meaning?
- What happens to names declared in a block when control exits the block?

Nested structure and stacks

- If the language permits nested name declarations, a stack is an appropriate data structure for maintaining the environment for those names
- A piece of a program that can contain its own name declarations is called a *block*
- In C/Java, a block is delimited by {}'s; in Pascal or Ada, by begin/end; in Onyx, by `for/return` or `let/return`
- Blocks can be nested, and can contain nested name declarations

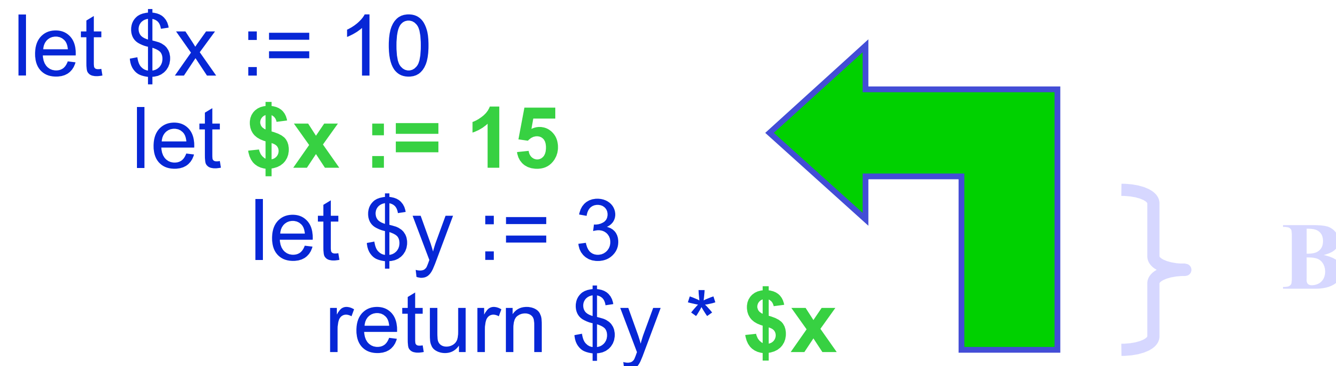
Properties of blocks

- Two blocks are either nested one within the other, or don't intersect at all
- Most-closely-nested rule for static scope:
- The scope of a declaration in a block is from the declaration to the end of the block

```
let $a := 3, $b := 5+$a
    return $a * $b
```

Most-closely-nested rule

- If a name x is not declared in a block B, then an occurrence of x in B is in the scope of a declaration of x in a block B′ that is more closely nested around B than any other block containing a declaration of x



Blocks and stacks

- The most-closely-nested rule is implemented with simple stack traversal
- A variable can be bound (pushed) when its declaration is encountered, and unbound (popped) when execution exits the block

Functions and blocks

- The body of a function definition is like a named block with a special set of variable declarations: the formal parameters
- We can also look as a block as an anonymous function without any parameters
- A parameter can be bound (pushed) when the function is called, and unbound (popped) when the function returns
- How do we distinguish a function call from a `let` or `for` ?

Free variables

- An occurrence of a name is *free* if it is not in a static scope of a declaration of that name
- In this Onyx example, the variable $x in the body of the function definition, and the variable $z in the return clause are free:

```
declare function g($a,$b) { $a + $x + $b };
let $a:=3, $y:=4, $x:=5
     return g($z,$a)
```

- What to do about that?

Static vs. dynamic scoping

- In a statically scoped language, with no global variables, a free variable is an error
  "variable not declared"
- However some languages permit ***dynamic scoping***
- An invocation of a function *inherits existing bindings* from the calling context; these can provide bindings for free variables in the function body
- With static scoping, an invocation of a function does not inherit bindings from the calling context and free variables are not allowed
- What is the result of the call f(4) ?

```
declare function f($x) { $x + $i };
let $i := 3 return f(4)
```

Implications of scoping

- With static scoping, the function always returns the same value for the same argument list
- With dynamic scoping, the function's behavior is sensitive to the environment
- Onyx implements static scoping

Free variables without dynamic binding

- In this example:

```
declare function g($a,$b) { $a + $x + $b };
let $a:=3, $y:=4, $x:=5
    return g($z,$a)
```

- The occurrence of $z in the FLWR expression is free, and cannot dynamically inherit a binding from anywhere; this is a problem

Free variables with dynamic binding

- Now in this example:

```
declare function g($a,$b) { $a + $x + $b }
let $a:=3, $y:=4, $x:=5
   return g($y,$a)
```

- The occurrence of $x in the function body is free, but it can dynamically inherit a binding when the function is called, if that is allowed by the language
- As expected, it should inherit the most recent binding, if there are multiple nested contexts

Free variables with dynamic binding

```
declare function g($a,$b) { $a + $x + $b };
let $a:=3, $y:=4, $x:=5 return g($y,$a)
```

x	**5**
y	**4**
a	**3**

- When the call g($y,$a) is evaluated:
  - bindings for $a, $y, $x have been pushed on the stack by the let-clauses
  - new bindings for the parameters `$a, $b` are pushed on the stack to prepare for the function call
  - when the body of the function is evaluated, the statically free variable `$x` is bound (to value 5; `$a` is bound to 4 and `$b` to 3)

Free variables with dynamic binding

```
declare function g($a,$b) { $a + $x + $b };
let $a:=3, $y:=4, $x:=5 return g($y,$a)
```

b	3
a	4
x	5
y	4
~~a~~	~~3~~

- When the call g($y,$a) is evaluated:
  - bindings for $a, $y, $x have been pushed on the stack by the let-clauses
  - new bindings for the parameters $a and $b are pushed on the stack to prepare for the function call
  - when the body of the function is evaluated, the statically free variable **$x** is bound (to value 5; **$a** is bound to 4 and **$b** to 3)

Free variables with dynamic binding

```
declare function g($a,$b) { $a + $x + $b };
let $a:=3, $y:=4, $x:=5 return g($y,$a)
```

b	3
a	4
x	5
y	4
~~a~~	~~3~~

- When the call g($y,$a) is evaluated:
  - bindings for $a, $y, $x have been pushed on the stack by the let-clauses
  - new bindings for the parameters $a, $b are pushed on the stack to prepare for the function call
  - when the body of the function is evaluated, the statically free variable $x is bound (to value 5; $a is bound to 4 and $b to 3)

Implementing lexical scoping

- The compiler can statically detect free variables and flag them as errors at compile time if required
- To prevent dynamic binding without static checks, just block the inheritance of bindings across function calls
- For example: When processing a function call, first push a special marker *START_FUNCTION_SCOPE* on the symbol table stack
- To resolve variable references, do not cross this marker when scanning the stack

b	3
a	4
x	5
y	4
a	3

Global variables

- Onyx supports global variable defintions

```
declare variable $z as onyx.types.Integer { 3 };
declare function g($a,$b) { $a + $z + $b };
let $a:=3, $y:=4, $x:=5 return g($y,$a)
```

- What is the result?

Types

- A type is:
  - a specification of a set of possible values
  - together with a specification of a set of operations on those values
- For example, the int primitive type in Java can be defined as
  - the set of 32-bit signed integer values
  - the operations +, -, *, /, %, etc.
- And a type usually has a name that identifies it uniquely: int, java.lang.String, onyx.types.Decimal, etc.

Type systems

- A ***type system*** is a collection of rules for assigning type expressions to various parts of a program
- A ***type expression*** is an expression that refers to a type
  - float, pointer to float, array of float, etc.
- A ***type checker*** implements a type system by annotating a program type expressions and testing if these annotations are permitted
- This can be done at compile time (static) or at run time (dynamic)

Type checking

- A *type checker* verifies that the way a construct is used is consistent with its type
- Examples
  - In Java, switch case labels can only be of primitive integer type
  - In Onyx, the to operator takes integer arguments only
- Type information is also needed for code generation or interpretation: which version of an overloaded function to call can depend on types of actual arguments in the call

Strong typing

- We say that a language is *strongly typed* if all type checks can be done statically
- That is, in a strongly typed language, if a program passes compile time type checks then it is guaranteed not to have type errors when run
- It is not clear that any interesting languages are really strongly typed in this sense

Practical definition of strong typing

- A language is strongly typed if all expressions have a definite type, and use of inappropriate types raises exceptions either at compile time or runtime
- In this sense, many languages, such as Java and Onyx, are strongly typed
- Such languages will try to do static type checking as much as possible, while relying on runtime type checking as necessary