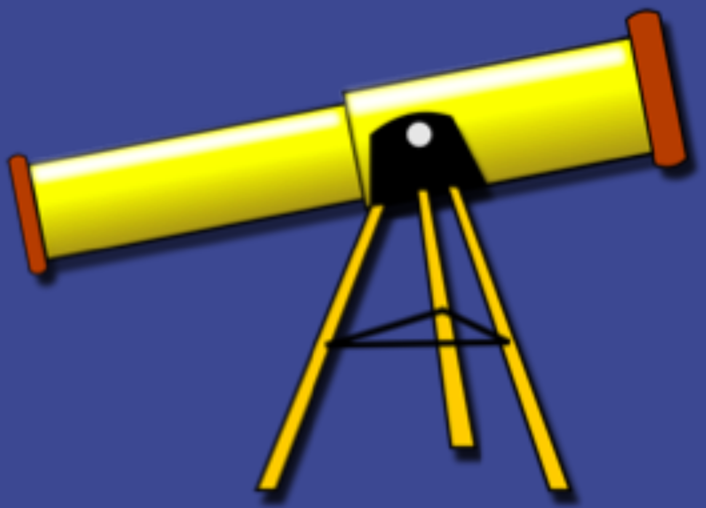
CS 152

Programming Paradigms

Basic Semantics
Attributes, Bindings & Scope



Today

- Homework 6 Hints
- Basic Semantics
  - Identifiers & Attributes
  - Binding
  - Scope

Course Learning Outcomes

7. Understand variable scoping and lifetimes.
8. Write interpreters for simple languages that involve arithmetic expressions, bindings of values to names, and function calls.

Homework 6

- *takeWhile*
- *takeRest*
- *read*: be careful with numbers that start or end with a decimal point. They are not supported in Haskell
  - read .5 => error
  - read 1. => error
- They are supported in Scheme: (+ .5 1.) => 1.5
- You need to support them in MUFL 1.0

Semantics - how to

- Syntax: what the language constructs look like
- Semantics: what the language constructs actually do
- Specifying semantics is more difficult than specifying syntax
- Several ways to specify semantics:
  - Language reference manual
  - Defining a translator
  - Formal definition

Specifying Semantics: Reference Manual

- Most common way to specify semantics
- Suffers from a lack of precision inherent in natural language descriptions
- May have omissions and ambiguities

Specifying Semantics: Defining a translator

- Questions about a language can be answered by experimentation
- Questions about program behavior cannot be answered in advance
- Bugs and machine dependencies in the translator may become part of the language semantics, possibly unintentionally
- May not be portable to all machines
- May not be generally available

Specifying Semantics: Formal definition

- Formal mathematical methods: precise, but are also complex and abstract, require study to understand
- Denotational semantics: define the meaning of a language by supplying a valuation function for each construct. The valuation function for a construct is defined in terms of the valuation functions for the sub-constructs
- Example: How do we evaluate: a + 3?

  eval ((expr1 + expr 2), env) = eval ((expr1), env) + eval ((expr2), env)

Semantics: What's in a name?

- We use names (or identifiers) to denote language entities or constructs
- Fundamental step in describing the semantics of a programming language is to describe the naming conventions for identifiers

Semantics: Location & Value

- Most languages include concepts of location and value
  - Value: any storable quantities
  - Location: place where value can be stored; usually a relative location

Attributes

Attributes are the properties that determine the meaning of the corresponding identifier.

const int n = 5;

- Identifier: n
- Attributes?
  - data type: int
  - value: 5
  - what else?
  - role: constant

Attributes

int x;

- Identifier: x
- Attributes?
  - data type: int
  - what else?
  - role: variable

Attributes for variables and constants include data type and value

Attributes

```
double f (int n) {
    …
}
```

- Identifier: f
- Attributes?
  - parameters: number, names, data type
  - return value: data type
  - function body/code
  - role: function

Attributes

Assignment statements also associate attributes with identifiers

x = 2;

Associate attribute "value: 2" with variable x.

Binding

- Binding: process of associating an attribute with a name/identifier
- Bindings are created by:
  - explicit declarations
  - by assignment statements or let expressions
  - by binding parameters to arguments in a function call.
- Binding time: the time when an attribute is computed and bound to a name

Categories of Binding

- Two categories of binding:
  - Static binding: occurs prior to execution
  - Dynamic binding: occurs during execution
- Static attribute: an attribute that is bound statically (prior to execution)
- Dynamic attribute: an attribute that is bound dynamically (during execution)

Static Attributes

Static attributes can be bound:

- during translation
- during linking
- during loading of the program
- prior to translation time
    - predefined identifiers: specified by the language definition

Dynamic Attributes

Dynamic attributes can be bound at different times during execution

- entry or exit from a function/procedure
- entry or exit from the program

Languages and Bindings

- Languages differ substantially in which attributes are bound statically or dynamically
- Functional languages tend to have more dynamic binding than imperative languages

Symbol Table

- A translator creates a data structure to maintain bindings
- Symbol table: a mapping from names to attributes

Symbol Table

$$\text{Names} \longrightarrow \text{Attributes}$$

Lexical, Syntax and Semantic Analysis

The parsing phase of translation includes three types of analysis:

- Lexical analysis: determines whether a string of characters represents a token
- Syntax analysis: determines whether a sequence of tokens represents a valid phrase in the grammar
- Static semantic analysis: establishes attributes of names in declarations and ensures that the use of these names conforms to their declared attributes

Explicit vs Implicit Bindings

Declaration:

int x;

Data type binding of x is explicit.

Location binding of x is implicit.

Explicit vs Implicit Binding

Python – no declarations:

x = 5

Value binding?

A. Explicit
B. Implicit

Explicit vs Implicit Binding

Python:

x = 5

Data type binding?

A. Explicit
B. Implicit

Explicit vs Implicit Binding

Python:

x = 5

Location binding?

A. Explicit
B. Implicit

Scope

- Scope of a binding: region of the program where the binding is valid/visible. The scope of an identifier refers to the part of the program where that identifier is visible, where it can be accessed.
- Levels of scope: block, function, module, …

Static vs Dynamic Scope

- Static/lexical scope: the bindings can be determined statically by reading the source code.
  - The meaning of an identifier can be determined by reading the source code to see where that identifier is defined

- Dynamic scope: the bindings are determined during execution and they depend on the runtime context

Scope

- Most modern languages use lexical/static scoping
- Early dialects of Lisp were dynamically scoped
- Modern dialects, including Scheme and Common Lisp, are lexically (statically) scoped

Example: Scope in Python

Python uses the location of the assignment (binding) to associate a variable with a scope.

The place where we assign a variable determines its scope of visibility. We can only access a variable in our scope after it has been assigned a value.

```
print(grade)
grade = 100
```

Error

```
grade = 100
print(grade)
```

100

Example: Function Scope in Python

The variables defined inside a function definition can only be seen by the code in that function definition. We cannot refer to these variables from outside the function.

```
def grader():
    grade = 100
    print(grade)

grader()
print(grade)
```

What does the last statement print?

A. 100

B. An error

Example: Function Scope in Python

```
grade = 70

def grader():
    print(grade)  # prints 70

grader()
```

Because the variable *grade* is assigned in the global scope (that is in the outermost code), the function *grader* has read access to it.

Example: Function Scope in Python

```
grade = 70

def grader():
    grade = 100 # new variable

grader()
```

An assignment statement inside *grader* creates a new local variable.
It does not modify the global variable

Summary: Lexical Scope in Python

- Parameters defined inside a function definition can be seen by the code in that function definition (including nested functions.)
- The scope of a variable in a function extends from the assignment statement until the end of the function (no hoisting).
- Variables defined in the outer statements outside any function are global variables. They can be read inside the function but in general they cannot be modified inside the function.
- An assignment statement to a variable inside a function usually creates a new local variable – even if a variable with the same name exists in the outer scope.

Example: Function Scope in Python

```
grade = 70

def grader():
    grade = 100
    print(grade)


print(grade)
grader()
print(grade)
```

What does the last statement print?

A. 100
B. 70
C. An error

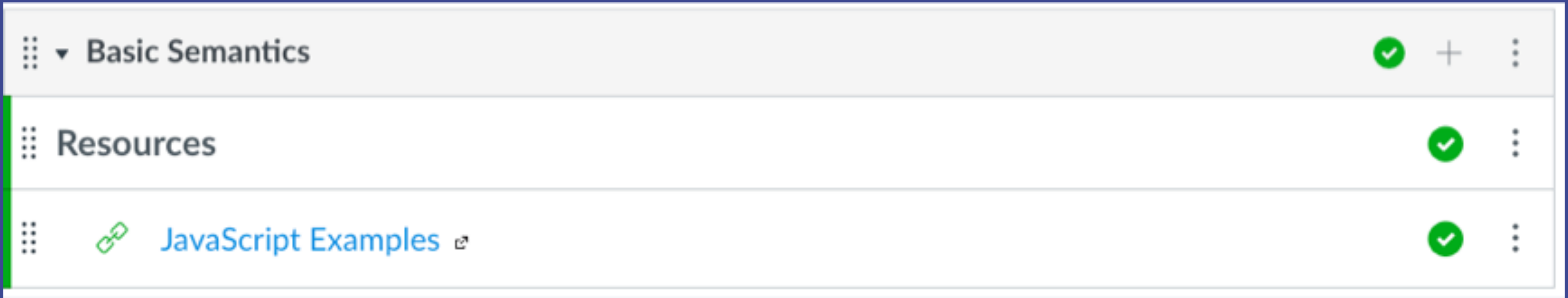
Example: Function Scope in Python

```
grade = 70

def grader():
    grade = 100
    print(grade)


print(grade)
grader()
print(grade)
```

JavaScript Examples



Example: Function Scope in JavaScript

We also have function scope in JavaScript when the variables are declared with the var keyword.

```
function grader() {
  var grade = 100;
  console.log(grade);
}
grader();
console.log(grade);
```

What does the last statement print?

A. 100
B. A Reference Error

Example: Function Scope in JavaScript

```
var grade = 70;
function grader() {
  var grade = 100;
  console.log(grade);
}

console.log(grade);
grader();
console.log(grade);
```

What does the last statement print?

A. 100
B. 70
C. A Reference Error

Example: Function Scope in JavaScript

```
var grade = 70;
function grader() {
  var grade = 100;
  console.log(grade);
}

console.log(grade);
grader();
console.log(grade);
```

Example: Hoisting in JavaScript

All variables declared with the *var* keyword within a function are visible throughout the body of the function.

Variable declarations are "hoisted" (moved up) to the top of the function.

Only the declaration is hoisted

The assignment is not.

Example: Hoisting in JavaScript

Only the declaration is hoisted
The assignment is not.

```
var a = 5;  // var declaration in JavaScript
```

Equivalent to:

```
var a;   // declaration
a = 5;  // assignment
```

Example: Hoisting in JavaScript

Accessing a variable that has not been declared or initialized results in a reference error

```
function grader() {
  console.log(grade);
}
grader()
```

What does this statement print?

A. undefined
B. A Reference Error

Example: Hoisting in JavaScript

- When a variable is declared in JavaScript, it is initialized to the value *undefined*.

```
function grader() {
  var grade;
  console.log(grade);
}
grader()
```

What does this statement print?

A. undefined
B. A Reference Error

Example: Hoisting in JavaScript

All variables declared with the *var* keyword within a function are visible throughout the body of the function.

Only the declaration is hoisted. The assignment is not.

```
function grader() {
  console.log(grade);
  var grade = 100;
  console.log(grade);
}
grader()
```

What does this statement print?

A. undefined
B. A Reference Error
C. 100

Example: Hoisting in JavaScript

The following implementations are equivalent in JavaScript

```
function grader() {
  console.log(grade);
  var grade = 100;
  console.log(grade);
}
grader()
```

```
function grader() {
  var grade;
  console.log(grade);
  grade = 100;
  console.log(grade);
}
grader()
```

Example: Block Scope in JavaScript

The *let* statement in JavaScript declares a block scope local variable, optionally initializing it to a value.

```
function grader(){
  var grade = 90;
  if (grade > 85) {
    let grade = 100;
    console.log(grade);
  }
  console.log(grade);
}
grader();
```

Example: Block Scope in JavaScript

```
function grader(){
  var grade = 90;
  if (grade > 85) {
    let grade = 100;
    console.log(grade);
  }
  console.log(grade);
}
grader();
```

What does this statement print?

A. undefined
B. A Reference Error
C. 100
D. 90
E. 85

Example: Block Scope in JavaScript

```
function grader(){
  var grade = 90;
  if (grade > 85) {
    let grade = 100;
    console.log(grade);
  }
  console.log(grade);
}
grader();
```

What does this statement print?

A. undefined
B. A Reference Error
C. 100
D. 90
E. 85

Example: Block Scope in JavaScript

```
function grader(){
  var grade = 90;
  if (grade > 85) {
    let grade = 100;
    console.log(grade);
  }
  console.log(grade);
}
grader();
```

Block Scope in Scheme

The special form *let* in Scheme introduces a list of local variables for use within its body:

```
(let
    (
      (variable1 value1)
      (variable2 value2)
       …
    )
    body)
```

Block Scope in Scheme

```
(let
    (
      (x 1)
      (y 2)
      (z 3)
     )
   (+ x y z))
```

```
(let
    (
      (variable1 value1)
      (variable2 value2)
       …
    )
    body)
```

Block Scope in Scheme

```
(define x 10)
(let
  (
    (x 1)
    (y 2)
    (z 3)
  )
 (+ x y z))
```

What does the let expression evaluate to?

Block Scope in Scheme

```
(define x 10)
(let
   (
     (x 1)
     (y 2)
     (z 3)
    )
  (+ x y z))
x
```

What does x evaluate to?

Block Scope in Scheme

```
(define x 10)
(let
   (
     (x 1)
     (y 2)
     (z 3)
    )
  (+ x y z))
y
```

What does y evaluate to?

A. 2
B. undefined

The Symbol Table

- Symbol table: a data structure that maps names to attributes
- Must support insertion, lookup, and deletion of names with associated attributes
- A lexically scoped, block-structured language requires a stack-like data structure
  - On block entry, all declarations of that block are processed and bindings added to symbol table
  - On block exit, bindings are removed, restoring any previous bindings that may have existed

Block Scope in Scheme

The special form let in Scheme introduces a list of local variables for use within its body:

```
(let
  (
    (variable1 value1)
    (variable2 value2)
    …
  )
  body)
```

Symbol Table

```
(define x 10)
(let
    (
      (x 1)
      (y 2)
      (z 3)
    )
  (+ x y z))
x
```

Name	Bindings
x	(integer 10)

On block entry, all declarations of that block are processed and bindings added to symbol table

Symbol Table

```
(define x 10)
(let
    (
      (x 1)
      (y 2)
      (z 3)
     )
   (+ x y z))
x
```

Name	Bindings
x	(integer 1) (integer 10)

On block entry, all declarations of that block are processed and bindings added to symbol table

Symbol Table

```
(define x 10)
(let
    (
      (x 1)
      (y 2)
      (z 3)
     )
   (+ x y z))
x
```

Name	Bindings
x	(integer 1) (integer 10)
y	(integer 2)

On block entry, all declarations of that block are processed and bindings added to symbol table

Symbol Table

```
(define x 10)
(let
    (
      (x 1)
      (y 2)
      (z 3)
    )
  (+ x y z))
x
```

Name	Bindings
x	(integer 1) (integer 10)
y	(integer 2)
z	(integer 3)

On block entry, all declarations of that block are processed and bindings added to symbol table

Symbol Table

```
(define x 10)
(let
    (
      (x 1)
      (y 2)
      (z 3)
     )
   (+ x y z))        6
x
```

Name	Bindings
x	(integer 1) (integer 10)
y	(integer 2)
z	(integer 3)

Symbol Table

```
(define x 10)
(let
    (
      (x 1)
      (y 2)
      (z 3)
     )
   (+ x y z))
x
```

Name	Bindings
x	(integer 10)

On block exit, bindings are removed, restoring any previous bindings that may have existed.