Naming Issues: Example 1

We need to know what thing a *name* refers to in our programs.

Overview

```
Consider, in Perl:
$x=1;
sub foo() { $x = 5; }
sub bar() { local $x = 2; foo(); print $x,"\n"; }
bar();
```

What gets printed for x?

SI 413 (USNA)

Overview

Unit 6

Naming Issues: Example 2

We need to know what thing a *name* refers to in our programs.

Consider, in Scheme:

```
(define x 1)
(define (foo x)
  (lambda () (display x)))
((foo 5))
(display x)
```

What gets printed for x?

SI 413 (USNA)

Unit 6

Fall 2023

2 / 27

3/27

Fall 2023

1/27

Naming Issues: Example 3

We need to know what thing a *name* refers to in our programs.

Overview

```
Consider, in C++:
char* foo() {
    char s[20];
    cin >> s;
    return s;
}
int bar (char* x) { cout << x << endl; }
int main() { bar(foo()); }
What gets printed for x?
</pre>
```

Dverview Basic terminology Name: A reference to something in our program Binding: An attachment of a value to a name Scope: The part of code where a binding is active Referencing Environment: The set of active bindings at the point of an expression Allocation: Setting aside space for an object Lifetime: The time when an object is in memory

Unit 6

Overview

Fall 2023 4 / 27

Options

SI 413 (USNA)

Scoping Single Global Scope Just one symbol table

- Dynamic Scope Stacks of scopes, depends on *run-time* behavior
- Lexical Scope Scope is based on the syntactical (lexical) structure of the code.

Allocation

- Static Allocation Allocation fixed at compile-time
- Stack Allocation Follows function calls
- Heap Allocation Done at run-time, as objects are created and destroyed

SI 413 (USNA)

Unit 6

Fall 2023 5 / 27

6 / 27

Allocation Static Allocation The storage for some objects can be fixed at compile-time. Then our program can access them *really quickly*! Examples: • Global variables • Literals (e.g. "a string") • *Everything* in Fortran 77? SI 413 (USNA) Unit 6 Fall 2023

| | Allocation | | |
|---|--|----------------------|--------|
| Stack Allocation | | | |
| The run-time stack is usuall Includes local variables, argu | - | | |
| Example: What does the sta int g(int x) { return | | C program? | |
| int g(int x) (ietuii | | | |
| <pre>int f(int y) { int x = 3 + g(y); return x; }</pre> | | | |
| <pre>int main() { int n = 5; f(n);</pre> | | | |
| } | | | |
| | | | |
| SI 413 (USNA) | Unit 6 | Fall 2023 | 7 / 27 |
| | | | |
| Heap Allocation The heap refers to a pile of typically used for <i>run-time m</i> This is the <i>slowest</i> kind of a Compilers/interpreters provi lots of heap-allocated storag Otherwise the segfault mons | nemory allocation. Allocation because it ding garbage collect ge. | happens at run-time. | |
| SI 413 (USNA) | Unit 6 | Fall 2023 | 8 / 27 |
| | | | |
| | Scoping Intro | | |
| Single Global Scope | | | |
| Why not just have every ins (Compiler writing would be | | d to the same object | ? |
| | | | |
| | | | |
| | | | |
| | | | |

Unit 6

Scoping Intro What is a scope? Certain language structures create a *new scope*. For example: int temp = 5;// Sorts a two-element array. void twosort(int A[]) { if (A[0] > A[1]) { int temp = A[0]; A[0] = A[1];A[1] = temp;} } int main() { int arr[] = $\{2, 1\};$ twosort(arr); cout << temp; // Prints 5, even with dynamic scoping!</pre> } SI 413 (USNA) Unit 6 Fall 2023 10 / 27 Scoping Intro Nested Scopes In C++, nested scopes are made using curly braces ($\{ and \}$). The scope resolution operator :: allows jumping between scopes manually. In most languages, function bodies are a nested scope. Often, control structure blocks are also (e.g. for, if, etc.)

Lexical scoping follows the nesting of scopes in the actual source code (as it is parsed).

Dynamic scoping follows the nesting of scopes as the program is executed.

SI 413 (USNA)

Unit 6

Fall 2023 11 / 27

Scoping Intro

Declaration Order

In many languages, variables must be *declared* before they are used. (Otherwise, the first use constitutes a declaration.)

In C/C++, the scope of a name starts at its declaration and goes to the end of the scope. Every name must be declared before its first use, because names are *resolved* as they are encountered.

 $\mathsf{C}{++}$ and Java make an exception for names in $\mathit{class\ scope}.$ Scheme doesn't resolve names until they are evaulated.

Declaration Order and Mutual Recursion

Consider the following familar code:

```
void exp() { atom(); exptail(); }
void atom() {
   switch(peek()) {
    case LP: match(LP); exp(); match(RP); break;
    // ...
   }
}
```

Scoping Intro

Mutual recursion in C/C++ requires forward declarations, i.e., function prototypes.

These wouldn't be needed within a class definition or in Scheme. C# and Pascal solve the problem in a different way...

Unit 6 Fall 2023 13 / 27

Dynamic Scope

Dynamic vs. Lexical Scope

Dynamic Scope

SI 413 (USNA)

- Bindings determined by most recent declaration (at run time)
- The same name can refer to many different bindings!
- Examples:

Lexical Scope

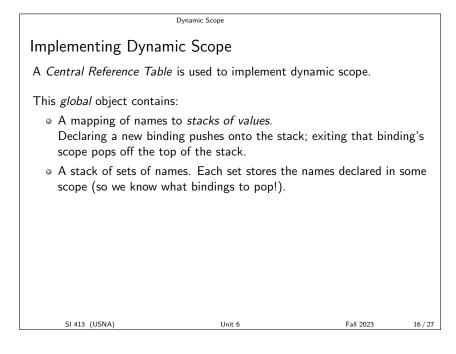
- Bindings determined from lexical structure at compile-time
- The same name always refers to the same binding.
- More common in "mature" languages
- Examples:

SI 413 (USNA)

Unit 6

Fall 2023 14 / 27

Dynamic Scope Dynamic vs. Lexical Example int x = 10;int foo(int y) { x = y + 5;print(x); } int main() { int x = 8;foo(9); print(x); } How does the behavior differ between a dynamic or lexically scoped language? SI 413 (USNA) Fall 2023 15 / 27 Unit 6



Dynamic Scope

Example: Central Reference Tables with Lambdas { new x := 0;new i := -1; new g := lambda z { ret := i; }; new f := lambda p { new i := x; if (i > 0) { ret := p@0; } else { x := x + 1; i := 3; ret := f@g; } }; write f@(lambda y {ret := 0}); } What gets printed by this (dynamically-scoped) SPL program? SI 413 (USNA) Unit 6 Fall 2023

Lexical Scope

Lexical Scope Tree

Name resolution in lexical scoping follows the scope tree:

- Every (nested) scope is a node in the tree.
- The root node is the global scope
- Nodes contain names defined in that scope.
- To determine active bindings, follow the tree up from the current scope until you see the name!

Example (program on previous slide):

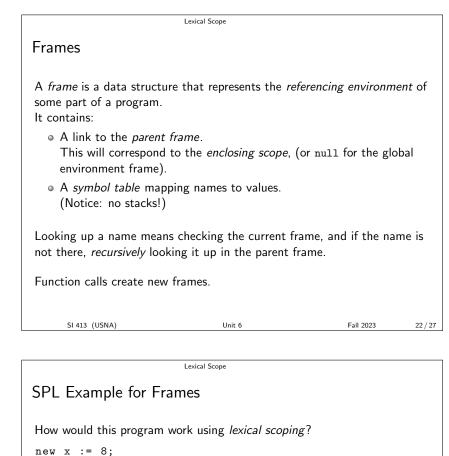
17 / 27

| | Lexical Scope | | |
|--|-----------------------------------|--------------------|---------|
| Reminder: The class | | | |
| | | | |
| | | | |
| Recall that functions in a | programming language c | can be: | |
| • Third class: Never t | reated like variables | | |
| • Second class: Passe | ed as parameters to other | functions | |
| • First class: Also ret | urned from a function an | d assigned to a va | riable. |
| | | | |
| | | | |
| | | | |
| SI 413 (USNA) | Unit 6 | Fall 2023 | 19 / 27 |
| · · · · | | | |
| | Lexical Scope | | |
| Implementing Lexica | I Scope | | |
| | | | |
| With <i>lexical scoping</i> , rules functions have more flexit | | omplicated when | |
| Third-class functions Can use "static links" | : " into the function call st | tack | |
| Second-class function Can use "dynamic lingular | ns: nks" into the function cal | ll stack | |
| • First-class functions: Must use Frames | | | |
| | | | |
| SI 413 (USNA) | Unit 6 | Fall 2023 | 20 / 27 |

Lexical Scope

Lexical Scope with 1st-Class Functions What happens here? { new f := lambda x { new g := lambda y { ret := x * y; }; ret := g; }; new h := f@2; write h@3; } Where are the non-local references stored?

21 / 27



new f := lambda n {
 write n + x;
};
{ new x := 10;
 write f@2;
}

• How do frames compare with activation records on the stack?

• Can we use frames for *dynamic* scoping?

SI 413 (USNA)

Closures

Unit 6

Lexical Scope

Fall 2023 23 / 27

How are functions represented as values (i.e., first-class)? With a *closure*! Recall that a closure is a function definition plus its referencing

environment. In the frame model, we represent this as a pair of:

- The function definition (parameters and body)
- A link to the frame where the function was defined

Example with closures

Draw out the frames and closures in a Scheme program using our stacks:

Lexical Scope

```
(define (make-stack)
 (define stack '())
 (lambda (arg)
   (if (eq? arg 'pop)
       (let ((popped (car stack)))
            (set! stack (cdr stack))
                popped)
        (set! stack (cons arg stack)))))
(define s (make-stack))
(s 5)
(s 20)
(s 'pop)
SI 413 (USNA) Unit 6 Fall 2023
```

Lexical Scope

Class outcomes

You should know:

- The meaning of terms like binding and scope
- The trade-offs involved in storage allocation
- The trade-offs involved in scoping rules
- The motivation behind declare-before-use rules, and their effect on mutual recursion.
- Why some languages restrict functions to 3rd-class or 2nd-class
- What non-local references are, and what kind of headaches they create
- How memory for local variables is allocated when in lexical scoping with first-class functions
- Why first class functions require different allocation rules
- What is meant by closure, referencing environment, and frame.

| SI 41 | 3 (USNA) |
|-------|----------|
| | |

Unit 6

Fall 2023 26 / 27

25 / 27

Lexical Scope

Class outcomes

You should be able to:

- \bullet Show how variables are allocated in C++, Java, and Scheme.
- Draw out activation records on a run-time stack.
- Determine the run-time bindings in a program using dynamic and lexical scoping.
- Draw the state of the Central Reference Table at any point in running a dynamically-scoped program
- Draw the tree of nested scopes for a lexically-scoped program.
- Trace the run of a lexically-scoped program.
- Draw the frames and closures in a program run using lexical or dynamic scoping

27 / 27