Scope and Parameter Passing
Outline

Overview
  Naming and scope
  Function/procedure calls

Static vs. dynamic scope

Parameter passing schemes
Review of naming

Most languages provide a way to **name** and **reuse** stuff.

**Naming concepts**

- **declaration**: introduce a new name
- **binding**: associate a name with a thing
- **reference**: use the name to stand for the bound thing

**C/Java variables**

```java
int x; int y;
x = slow(42);
y = x + x + x;
```

**In Haskell:**

**Local variables**

```haskell
let x = slow 42
    in x + x + x
```

**Type names**

```haskell
type Radius = Float
data Shape = Circle Radius
```

**Function parameters**

```haskell
area r = pi * r * r
```
Scope

Every name has a **scope**
The parts of the program where that name can be referenced

**Block**: shared scope of a group of declared names

**Shadowing**: when a declaration in an inner block temporarily hides a name in an outer block

### C blocks
```c
{ int x;
  int y;
  x = 2;
  if (x == 3) {
    int x = 4;
    int z = 5;
    y = x;
  }
  print(x);
}
```

### Python locals
```python
def demo():
x = 6
if x == 7:
x = 8
y = x
print x
print y
```

### Haskell **let**
```haskell
let x = 9
    y = x
in let x = 5
    z = y
in (x,y)
```
Implementing nested scopes

Recall CS 271 approach:
- local variables are stored in a **stack frame**
- **enter** a block: **push** a frame
- **exit** a block: **pop** a frame

```
type Frame = [(Var, Val)]
type Stack = [Frame]
```

Compare with **environments**:
```
type Env = [(Var, Val)]
```

Just a flat stack!
Outline

Overview

Naming and scope

Function/procedure calls

Static vs. dynamic scope

Parameter passing schemes
Function/procedure declarations

Function definitions declare names in two scopes

1. the **function name**: in the file/module
2. the **argument names** (parameters): in the function body

Example: Haskell

```
triple :: Int -> Int
triple y = double y + y

double :: Int -> Int
double x = x + x

perimeter :: Int -> Int -> Int
perimeter x y = double x + double y
```
A function definition contains:

- the **declaration** of the parameters
- **references** to the parameters

Q: Where/when are the parameters **bound**?
A: At the **call site**!
References in function definitions

Three kinds of variable names
- parameters
- local variables
- external variables

Where are bindings for …
- parameter and local names?
  - in current(ish) stack frame!
- external names?
  - good question!

Haskell
```haskell
area :: Float -> Float
area d = let r = d / 2
         in pi * r * r
```

C/Java
```c
float area(float d) {
    float r = d / 2;
    return pi * r * r;
}
```
Outline

Overview
  Naming and scope
  Function/procedure calls

Static vs. dynamic scope

Parameter passing schemes
Static vs. dynamic scope

**Static scope:** external names refer to variables that are visible at **definition**

**Dynamic scope:** external names refer to variables that are visible at **call site**

---

**Definition**

```java
int x = 3;
...
int baz(int a) {
    int b = x+a;
    return b;
}
```

**Call site**

```java
int x = 4;
...
int y = baz(5);
```

Q: What is the value of `y`?

- **static scope:** 8
- **dynamic scope:** 9
Dynamic scope

References refer to most recent binding during execution

Performing a function call

1. push frame with parameters onto the stack
2. run function body, save return value
3. pop frame from stack and resume executing

Tradeoffs:

- supports ad-hoc extensibility
- all names are part of the public interface
  - risk of name collision and unintended behavior
  - bad modularity – hard to refactor and understand
Static scope

References refer to most recent binding *in the source code*

Performing a function call

1. save current stack, restore function’s stack
2. push frame with parameters onto the stack
3. run function body, save return value
4. restore saved stack and resume executing

Tradeoffs:

- names are not part of the public interface
  - no risk of name collision – more predictable behavior
  - improved modularity – can change names without breaking clients
- only supports planned extensibility
Closures

**Closure** = function + its environment (stack)

Needed to implement static scoping!
Outline

Overview
   Naming and scope
   Function/procedure calls

Static vs. dynamic scope

Parameter passing schemes
Call-by-value parameter passing

**Definition**
```python
def foo(a, b, c):
    a := b + 1
    c := a - b
    return c
```

**Call site**
```python
x := 4
y := foo(3, x, x + 1)
```

1. evaluate argument expressions
2. push frame with argument values

**Environment:**
```
[('a', 3), ('b', 4), ('c', 5)]
```
Call-by-name parameter passing

Definition

def foo(a,b,c):
    if a > 0 then
        a := a + b
    else
        a := a + c
    return a

Call site

x := 5
y := 0
foo(x,x+y,x/y)

1. push frame with argument expressions

Environment: [(Var,Exp)]

[("a", Ref "x"),
 ("b", Add (Ref "x") (Ref "y")),
 ("c", Div (Ref "x") (Ref "y"))]

This simple approach only works with dynamic scoping – why?

What happens if an argument has a side effect?
Call-by-need parameter passing (a.k.a. lazy evaluation)

Idea: Use call-by-name, but **remember** the value of any argument we evaluate

- only evaluate argument if needed, but evaluate each at most once
- best aspects of call-by-value and call-by-name!

**Definition**

```python
def triple(x,y):
    if x > 0 then
        z := x + x + x
    else
        z := y + y + y
    return z
```

**Call site**

```python
triple (slow(42), crash())
```

1. push frame with argument expressions
2. replace expressions by values as evaluated

**Environment:** `[((Var, Either Exp Val)]`
Call-by-reference parameter passing

Only relevant in languages with assignment

- use a "store" to simulate memory

Definition

```
def foo(a, b, c):
a := b+5
c := a-b
```

Call site

```
x := 2
y := 3
z := 4
foo(x, y, z)
```

Note: only plain variable references allowed as arguments!

1. push frame with argument addresses

```
Environment: [(Var, Addr)]
[("a", 2), ("b", 1), ("c", 0)]
```

Parameter passing schemes