



CS 381: Programming Language Fundamentals

Summer 2015

Scope and Parameter Passing
July 20, 2015

Outline

Overview

Naming and scope

Function/procedure calls

Static vs. dynamic scope

Parameter passing scheme

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 in for the bound thing

C/Java variables

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

In Haskell:

Local variables

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

Type names

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

Function parameters

```
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

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

Python locals

```
def demo():  
    x = 6  
    if x == 7:  
        x = 8  
        y = x  
    print x  
    print y
```

Haskell let

```
let x = 9  
    y = x  
    in let x = 5  
        z = y  
        in (x,y)
```

Implementing nesting scopes

Recall CS 271 approach:

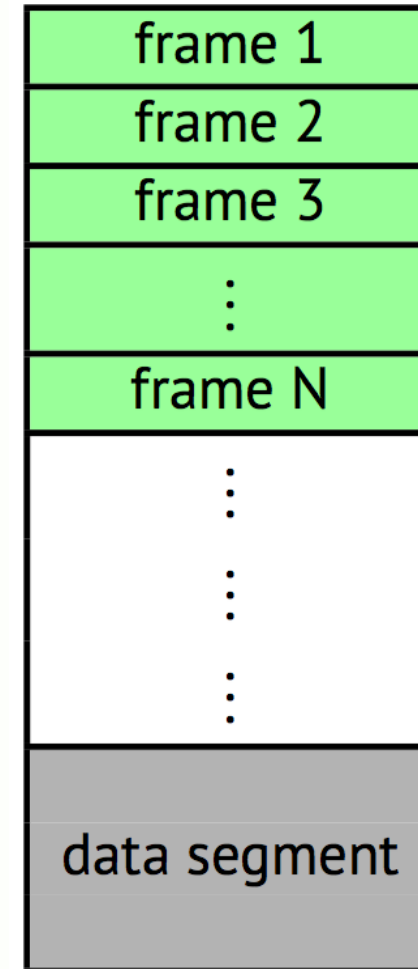
- local variables are stored in a **stack frame**
- **enter** a block: **push** a frame
- **exit** a block: **pop** a frame

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

Compare to **environments**:

```
type Env = [(Var,Val)]
```

Just a flat stack!



Outline

Overview

Naming and scope

Function/procedure calls

Static vs. dynamic scope

Parameter passing scheme

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
```

Binding parameters

A function definition contains:

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

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

Q: Where/when are the parameters **bound**?

A: At the **call site**!

```
GHCi> double 5  
10
```


References in function definitions

Three kinds of names

- **parameters**
- **local** variables
- **external** variables

Where are bindings for ...

- **parameter** and **local** names?
 - in current(ish) stack frame!
- **external** names?
 - good question!

Haskell

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

C/Java

```
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 scheme

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

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

Call site

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

Q: What is the value of **y**?

A: static scope — 8

dynamic scope — 9

Static scope

References refer to the 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 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**

Dynamic scope

References refer to the most recent binding **during execution**

Performing a function call

1. push frame with parameters onto 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

Scope of functions and parameters

What do we mean by *scope* when talking about a stack?

```
{int x;  
  {int f(int y){return y + 1};  
    x := f(1);  
  }  
}  
...
```

[]	
[x: ?]	<i>push</i>
[f: {}, x: ?]	<i>push</i>
[y: 1, f: {}, x: ?]	<i>push</i>
[f: {}, x: 2]	<i>pop</i>
[x: 2]	<i>pop</i>
[]	<i>pop</i>

Static scope of functions and parameters

Non-local variable

```
{int x;  
  x := 1;  
  {int f(int y){return y - x};  
    {int x;  
      x := 2;  
      x := f(3);  
    }  
  }  
}
```

[]	
[x: ?]	<i>push</i>
[x: 1]	
[f: {}, x: 1]	<i>push</i>
[x: ?, f: {}, x: 1]	<i>push</i>
[x: 2, f: {}, x: 1]	
[y: 3, x: 2, f: {}, x: 1]	<i>push</i>
[x: 4, f: {}, x: 1]	<i>pop</i>
[f: {}, x: 1]	<i>pop</i>
[x: 1]	<i>pop</i>
[]	<i>pop</i>

Static scope

Dynamic scope of functions and parameters



Scope.hs

Non-local variable

```
{int x;  
  x := 1;  
  {int f(int y){return y - x};  
    {int x;  
      x := 2;  
      x := f(3);  
    }  
  }  
}
```

[]	
[x: ?]	<i>push</i>
[x: 1]	
[f: {}, x: 1]	<i>push</i>
[x: ?, f: {}, x: 1]	<i>push</i>
[x: 2, f: {}, x: 1]	
[y: 3, x: 2, f: {}, x: 1]	<i>push</i>
[x: 5, f: {}, x: 1]	<i>pop</i>
[f: {}, x: 1]	<i>pop</i>
[x: 1]	<i>pop</i>
[]	<i>pop</i>

Dynamic scope

Exercises: runtime stacks

Show the development of the runtime stack under *static* and *dynamic* scoping for the execution of the following code.

```
{int y := 1;
  {int z := 0;
    {int f(int x){return x + y};
      {int g(int y){return f(2)};
        z := g(3);
        ...
      }
    }
  }
}
```

```
[ ]
[y:1]
[z:0,y:1]
[f:{},z:0,y:1]
[g:{},f:{},z:0,y:1]
[y:3,g:{},f:{},z:0,y:1]      call of g
[x:2,y:3,g:{},f:{},z:0,y:1]  call of f
[g:{},f:{},z:3,y:1]          static
[g:{},f:{},z:5,y:1]          dynamic
```

Exercises: runtime stacks

Show the development of the runtime stack under *static* and *dynamic* scoping for the execution of the following code.

```
{int z := 0;
  {int f(int x){return x + 1};
    {int g(int y){return f(y)};
      {int f(int x){return x - 1};
        z := g(3);
        ...
      }
    }
  }
}
```

```
[]
[z:0]
[f:{}, z:0]
[g:{}, f:{}, z:0]
[f:{}, g:{}, f:{}, z:0]
[y:3, f:{}, g:{}, f:{}, z:0]      call of g
[x:3, y:3, f:{}, g:{}, f:{}, z:0] call of f
[f:{}, g:{}, f:{}, z:4]          static
[f:{}, g:{}, f:{}, z:2]          dynamic
```

Outline

Overview

Naming and scope

Function/procedure calls

Static vs. dynamic scope

Parameter passing scheme

Call-by-value parameter passing

Definition

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

Call site

```
x := 4  
y := foo(3,x,x+1)
```

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

Environment: [(Var, Val)]

```
[ ("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!

Call-by-need parameter passing (a.k.a. lazy evaluation)

Definition

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

Call site

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

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

Environment: [(Var, Either Exp Val)]

Call-by-name parameter passing



Scope.hs

Definition

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

Call site

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

1. push frame with argument **addresses**

Environment: [(Var, Address)]

```
[ ("a", 2), ("b", 1), ("c", 0) ]
```

Note only plain variable references
allowed as arguments!