Names, Scope, and Bindings

Stephen A. Edwards

Columbia University

Fall 2008

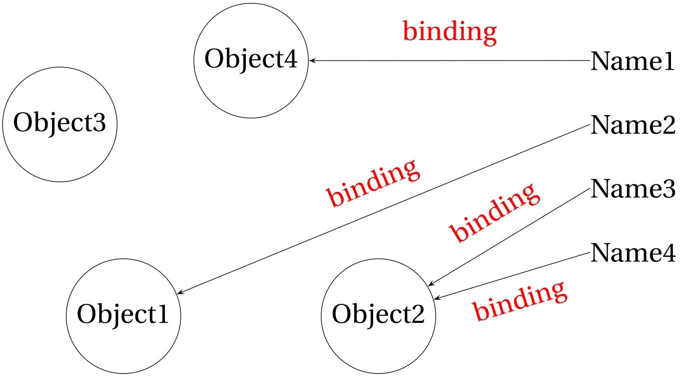
What's In a Name?

Name: way to refer to something else

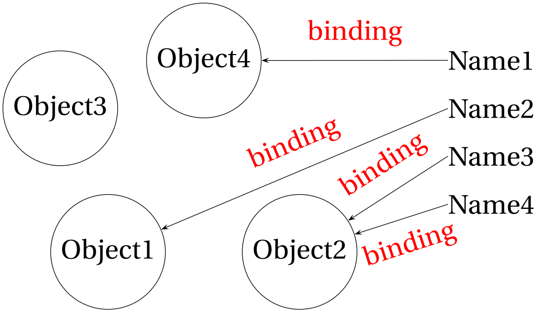
variables, functions, namespaces, objects, types

```
if ( a < 3 ) {
  int bar = baz(a + 2);
  int a = 10;
}
```

Names, Objects, and Bindings



Names, Objects, and Bindings



When are objects created and destroyed?

When are names created and destroyed?

When are bindings created and destroyed?

Part I

Object Lifetimes

Object Lifetimes

The objects considered here are regions in memory.

Three principal storage allocation mechanisms:

1. Static
   Objects created when program is compliled, persists throughout run
2. Stack
   Objects created/destroyed in last-in, first-out order. Usually associated with function calls.
3. Heap
   Objects created/deleted in any order, possibly with automatic garbage collection.

Static Objects

```
class Example {
  public static final int a = 3;

  public void hello() {
    System.out.println("Hello");
  }
}
```

Static class variable

Code for hello method

String constant “hello”

Information about Example class.

Static Objects

Advantages:

- Zero-cost memory management
- Often faster access (address a constant)
- No out-of-memory danger

Disadvantages:

- Size and number must be known beforehand
- Wasteful if sharing is possible

Stack-Allocated Objects

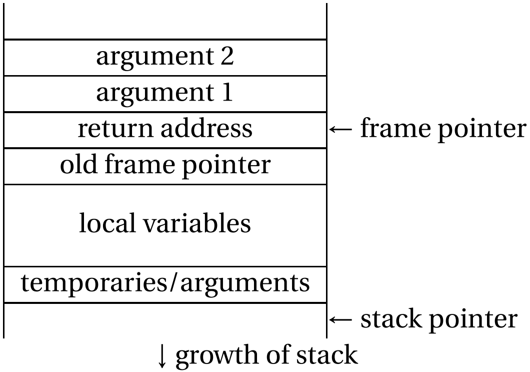
Natural for supporting recursion.

Idea: some objects persist from when a procedure is called to when it returns.

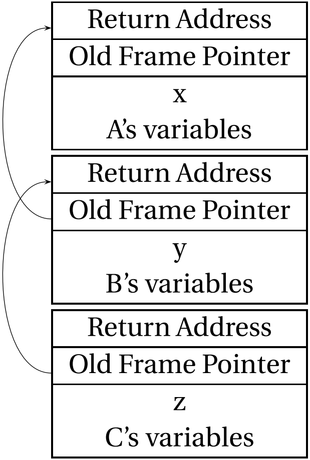
Naturally implemented with a stack: linear array of memory that grows and shrinks at only one boundary.

Each invocation of a procedure gets its own *frame* (*activation record*) where it stores its own local variables and bookkeeping information.

Activation Records



Activation Records




```
int A() {
  int x;
  B();
}

int B() {
  int y;
  C();
}

int C() {
  int z;
}
```

Stack-Based Langauges

The FORTH language is stack-based. Very easy to implement cheaply on small processors.

The PostScript language is also stack-based.

Programs are written in Reverse Polish Notation:

```
2 3 * 4 5 * + .  ( . is print top-of-stack)
26 OK
```

FORTH

```
: CHANGE     0       ;
: QUARTERS 25 * + ;
: DIMES     10 * + ;
: NICKELS    5 * + ;
: PENNIES        + ;
: INTO 25 /MOD CR . ." QUARTERS"
       10 /MOD CR . ." DIMES"
        5 /MOD CR . ." NICKELS"
               CR . ." PENNIES" ;
CHANGE 3 QUARTERS 6 DIMES 10 NICKELS
112 PENNIES INTO
11 QUARTERS
2 DIMES
0 NICKELS
2 PENNIES
```

FORTH

Definitions are stored on a stack. FORGET discards the given definition and all that came after.

```
: FOO ." Stephen" ;
: BAR ." Nina" ;
: FOO ." Edwards" ;
FOO Edwards
BAR Nina
FORGET FOO   ( Forgets most-recent FOO)
FOO Stephen
BAR Nina
FORGET FOO   ( Forgets FOO and BAR)
FOO FOO ?
BAR BAR ?
```

Heap-Allocated Storage

Static works when you know everything beforehand and always need it.

Stack enables, but also requires, recursive behavior.

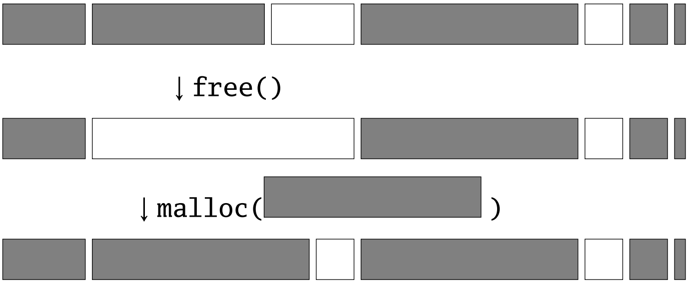
A *heap* is a region of memory where blocks can be allocated and deallocated in any order.

(These heaps are different than those in, e.g., heapsort)

Dynamic Storage Allocation in C

```
struct point {
   int x, y;
};

int play_with_points(int n)
{
  int i;
  struct point *points;

  points = malloc(n * sizeof(struct point));

  for ( i = 0 ; i < n ; i++ ) {
    points[i].x = random();
    points[i].y = random();
  }

  /* do something with the array */

  free(points);
}
```

Dynamic Storage Allocation



Dynamic Storage Allocation

Rules:

- Each allocated block contiguous (no holes)
- Blocks stay fixed once allocated

`malloc()`

- Find an area large enough for requested block
- Mark memory as allocated

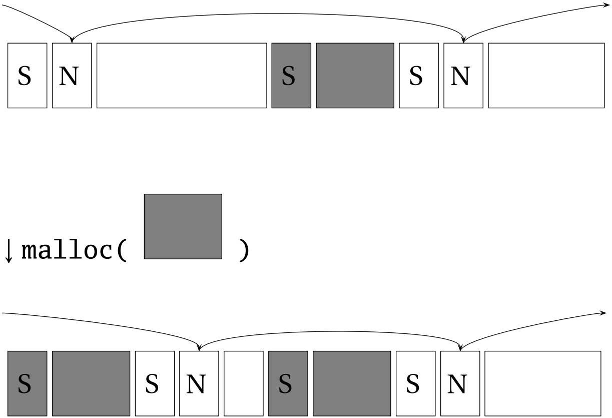
`free()`

- Mark the block as unallocated

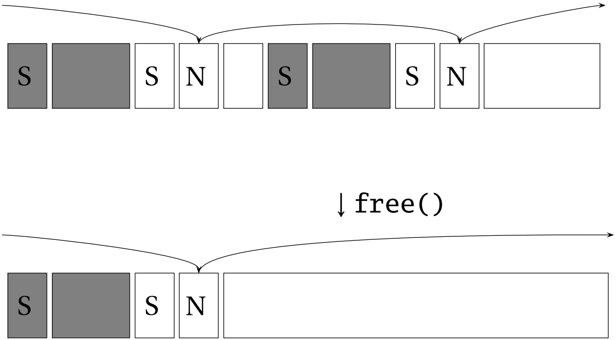
Simple Dynamic Storage Allocation

- Maintaining information about free memory
  - Simplest: Linked list
- The algorithm for locating a suitable block
  - Simplest: First-fit
- The algorithm for freeing an allocated block
  - Simplest: Coalesce adjacent free blocks

Dynamic Storage Allocation



Simple Dynamic Storage Allocation



Dynamic Storage Allocation

Many, many other approaches.

Other “fit” algorithms

Segregation of objects by size

More clever data structures

Heap Variants

Memory pools: Differently-managed heap areas

Stack-based pool: only free whole pool at once

- Nice for build-once data structures

Single-size-object pool:

- Fit, allocation, etc. much faster
- Good for object-oriented programs

Fragmentation

`malloc(`  `)` seven times give

`free()` four times gives

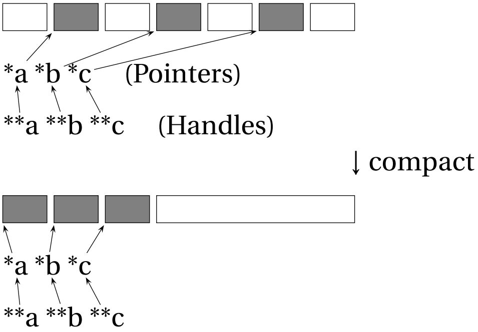
`malloc(`  `)` ?

Need more memory; can't use fragmented memory.

Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through “handles.”



The original Macintosh did this to save memory.

Automatic Garbage Collection

Remove the need for explicit deallocation.

System periodically identifies reachable memory and frees unreachable memory.

Reference counting one approach.

Mark-and-sweep another: cures fragmentation.

Used in Java, O'Caml, other functional languages, etc.

Automatic Garbage Collection

Challenges:

How do you identify all reachable memory?

(Start from program variables, walk all data structures.)

Circular structures defy reference counting:

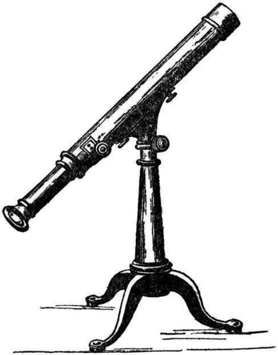
Neither is reachable, yet both have non-zero reference counts.

Garbage collectors often conservative: don't try to collect everything, just that which is definitely garbage.

Part II

Scope

When are names created, visible, and destroyed?



Scope

The scope of a name is the textual region in the program in which the binding is active.

Static scoping: active names only a function of program text.

Dynamic scoping: active names a function of run-time behavior.

Scope: Why Bother?

Scope is not necessary. Languages such as assembly have exactly one scope: the whole program.

Reason: Information hiding and modularity.

Goal of any language is to make the programmer's job simpler.

One way: keep things isolated.

Make each thing only affect a limited area.

Make it hard to break something far away.

Basic Static Scope

Usually, a name begins life where it is declared and ends at the end of its block.

```
void foo()
{

  int k;
}
```

Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

```
void foo()
{
  int x;

  while ( a < 10 ) {
    int x;

  }


}
```

Static Scoping in Java

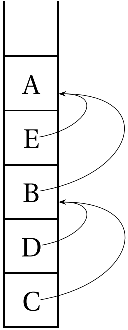
```
public void example() {
  // x, y, z not visible

  int x;
  // x visible

  for ( int y = 1 ; y < 10 ; y++ ) {
    // x, y visible

    int z;
    // x, y, z visible
  }

  // x visible
}
```

Nested Subroutines in Pascal

```
procedure mergesort;
var N : integer;

  procedure split;
  var I : integer;
  begin
  ...
  end

  procedure merge;
  var J : integer;
  begin
  ...
  end

begin
...
end
```

Nested Subroutines in Pascal

```
procedure A;
  procedure B;
    procedure C;
    begin
    ...
    end

    procedure D;
    begin
    C
    end

  begin
  D
  end

  procedure E;
  begin
  B
  end

begin
E
end
```



Scope in O'Caml

The `let` construct defines scopes:

```
let x = 8 in



;;
```

`let rec` extends the scope of the name into its definition, which is useful for recursive functions:

```
let rec fib i =
  if i < 1 then 1 else
    fib (i - 1) + fib (i - 2)
in
  fib 5;;
```

Scope in O'Caml

Scopes can nest to produce holes

```
let x = 8 in
   (let x = x + 2 in
    x + 2),
x
```

This returns the pair (12, 8).

Scope in O'Caml

Mutual recursion requires the use of the and keyword:

```
let rec fac n =
    if n < 2 then
      1
    else
      n * fac1 n
and fac1 n = fac (n - 1)
in
fac 5;;
```

Nested Functions in O'Caml

Static (lexical) scope like Pascal

```
let a = 3 in
let f1 x = a + x in
let a = 4 in
let f2 x = f1 x in
f2 3
```

Dynamic Definitions in TeX

```
% \x, \y undefined
{
  % \x, \y undefined
  \def \x 1
  % \x defined, \y undefined

  \ifnum \a < 5
    \def \y 2
  \fi

  % \x defined, \y may be undefined
}
% \x, \y undefined
```

Static vs. Dynamic Scope

```
program example;
var a : integer; (* Outer a *)

  procedure seta;
  begin
    a := 1  (* Which a does this change? *)
  end

  procedure locala;
  var a : integer; (* Inner a *)
  begin
    seta
  end

begin
  a := 2;
  if (readln() = 'b')
    locala
  else
    seta;
  writeln(a)
end
```

Static vs. Dynamic Scope

Most languages now use static scoping.

Easier to understand, harder to break programs.

Advantage of dynamic scoping: ability to change environment.

A way to surreptitiously pass additional parameters.

Application of Dynamic Scoping

```
program messages;
var message : string;

  procedure complain;
  begin
    writeln(message);
  end

  procedure problem1;
  var message : string;
  begin
    message := 'Out of memory';
    complain
  end

  procedure problem2;
  var message : string;
  begin
    message := 'Out of time';
    complain
  end
```

Forward Declarations

Languages such as C, C++, and Pascal require *forward declarations* for mutually-recursive references.

```
int foo(void);
int bar() { ... foo(); ... }
int foo() { ... bar(); ... }
```

Partial side-effect of compiler implementations. Allows single-pass compilation.

Open vs. Closed Scopes

An *open scope* begins life including the symbols in its outer scope.

Example: blocks in Java

```
{
  int x;
  for (;;) {
    /* x visible here */
  }
}
```

A *closed scope* begins life devoid of symbols.

Example: structures in C.

```
struct foo {
  int x;
  float y;
}
```

Part III

Overloading

What if there is more than one object for a name?

Overloading versus Aliases

Overloading: two objects, one name

Alias: one object, two names

In C++,

```
int foo(int x) { ... }
int foo(float x) { ... } // foo overloaded

void bar()
{
  int x, *y;
  y = &x;  // Two names for x: x and *y
}
```

Examples of Overloading

Most languages overload arithmetic operators:

```
1 + 2           // Integer operation
3.1415 + 3e-4   // Floating-point operation
```

Resolved by checking the *type* of the operands.

Context must provide enough hints to resolve the ambiguity.

Function Name Overloading

C++ and Java allow functions/methods to be overloaded.

```
int   foo();
int   foo(int a);   // OK: different # of args
float foo();       // Error: only return type
int   foo(float a); // OK: different arg types
```

Useful when doing the same thing many different ways:

```
int add(int a, int b);
float add(float a, float b);

void print(int a);
void print(float a);
void print(char *s);
```

Function Overloading in C++

Complex rules because of *promotions*:

```
int i;
long int l;
l + i
```

Integer promoted to long integer to do addition.

```
3.14159 + 2
```

Integer is promoted to double; addition is done as double.

Function Overloading in C++

1. Match trying trivial conversions
   `int a[]` to `int *a`, *T* to *const T*, etc.
2. Match trying promotions
   `bool` to `int`, `float` to `double`, etc.
3. Match using standard conversions
   `int` to `double`, `double` to `int`
4. Match using user-defined conversions
   `operator int() const { return v; }`
5. Match using the elipsis `...`

Two matches at the same (lowest) level is ambiguous.

Part IV

Symbol Tables

How does a compiler implement scope rules?

Symbol Tables

Basic mechanism for relating symbols to their definitions in a compiler.

Eventually need to know many things about a symbol:

- Whether it is defined in the current scope. “Undefined symbol”
- Whether its defined type matches its use. `1 + "hello"`
- Where its object is stored (statically allocated, on stack).

Symbol Tables

Implemented as a collection of dictionaries in which each symbol is placed.

Two operations: insert adds a binding to a table and lookup locates the binding for a name.

Symbol tables are created and filled, but never destroyed.

Symbol Tables

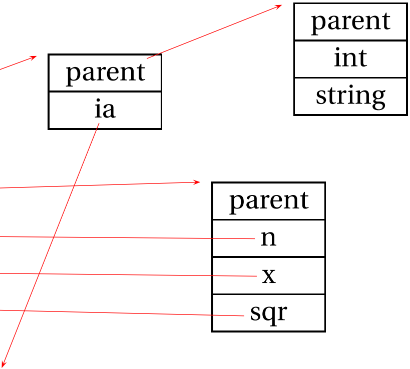
There are three namespaces in the Tiger functional language:

- functions and variables
- types
- record names

How many namespaces are there in Java?

How many namespaces are there in your language?

Symbol Tables in Tiger




```
let
  var n := 8
  var x := 3
  function sqr(a:int)
      = a * a
  type ia = array of int
in
  n := sqr(x)
end
```

Implementing Symbol Tables

Many different ways:

- linked-list
- hash table
- binary tree

Hash tables are faster, but linked lists are good enough for simple compilers.

Symbol Table Lookup

Basic operation is to find the entry for a given symbol.

In many implementation, each symbol table is a scope.

Each symbol table has a pointer to its parent scope.

Lookup: if symbol in current table, return it, otherwise look in parent.

Static Semantic Checking

Main application of symbol tables.

A taste of things to come:

Enter each declaration into its symbol table.

Check that each symbol used is actually defined in the symbol table.

Check its type. . . (next time)

Part V

Binding Time

When are bindings created and destroyed?

Binding Time

When a name is connected to an object.

Bound when	**Examples**
language designed	`if` `else`
language implemented	data widths
Program written	`foo` `bar`
compiled	static addresses, code
linked	relative addresses
loaded	shared objects
run	heap-allocated objects

Binding Time and Efficiency

Earlier binding time ⇒ more efficiency, less flexibility

Compiled code more efficient than interpreted because most decisions about what to execute made beforehand.

```
switch (statement) {

case add:
   r = a + b;
   break;

case sub:
   r = a - b;
   break;

   /* ... */
}
```

```
add %o1, %o2, %o3
```

Binding Time and Efficiency

Dynamic method dispatch in OO languages:

```
class Box : Shape {
  public void draw() { ... }
}

class Circle : Shape {
  public void draw() { ... }
}

Shape s;
s.draw();  /* Bound at run time */
```

Binding Time and Efficiency

Interpreters better if language has the ability to create new programs on-the-fly.

Example: Ousterhout's Tcl language.

Scripting language originally interpreted, later byte-compiled.

Everything's a string.

```
set a 1
set b 2
puts "$a + $b = [expr $a + $b]"
```

Binding Time and Efficiency

Tcl's `eval` runs its argument as a command.

Can be used to build new control structures.

```
proc ifforall {list pred ifstmt} {
  foreach i $list {
    if [expr $pred] { eval $ifstmt }
  }
}

ifforall {0 1 2} {$i % 2 == 0} {
  puts "$i even"
}

0 even
2 even
```

Part VI

Binding Reference Environments

What happens when you take a snapshot of a subroutine?

References to Subroutines

In many languages, you can create a reference to a subroutine and call it later. E.g., in C,

```
int foo(int x, int y) { /* ... */ }

void bar()
{
  int (*f)(int, int) = foo;

  (*f)(2, 3); /* invoke foo */
}
```

Where does its environment come from?

References to Subroutines

C is simple: no function nesting; only environment is the omnipresent global one. But what if there were?

```
typedef int (*ifunc)();

ifunc foo() {
  int a = 1;

  int bar() { return a; } /* this is not C */

  return bar;
}

int main() {
  ifunc f = foo();  /* returns bar */
  return (*f)();    /* call bar. a? */
}
```