G22.2110-003 Programming Languages - Fall 2012

Lecture 2

Thomas Wies

New York University

Review

Last week

- Programming Languages Overview
- Syntax and Semantics
- Grammars and Regular Expressions

High-level Programming Languages

What are the main reasons for using high-level programming languages?

High-level Programming Languages

What are the main reasons for using high-level programming languages?

- Machine independent
- Easier to use and understand

High-level Programming Languages

What are the main reasons for using high-level programming languages?

- Machine independent
- Easier to use and understand

As a language designer, the first point is straightforward to understand and implement.

High-level Programming Languages

What are the main reasons for using high-level programming languages?

- Machine independent
- Easier to use and understand

As a language designer, the first point is straightforward to understand and implement.

The second point is much less clear. It has taken many years to converge on a set of techniques that make a good programming language, and this is still evolving.

Outline

- Names and Bindings
- Lifetimes and Allocation
- Garbage Collection
- Scope

Sources:
PLP, 3.1 - 3.4, except 3.3.4 and 3.3.5

Names

What is a name?

Names

What is a name?

An *identifier*, made up of a string of characters, used to represent something else.

Names

What is a name?

An *identifier*, made up of a string of characters, used to represent something else.

What can be named?

Names

What is a name?

An *identifier*, made up of a string of characters, used to represent something else.

What can be named?

- Execution points (labels)
- Mutable variables
- Constant values
- Functions
- Types
- Type constructors (e.g., list or vector)
- Classes
- Modules/packages
- Execution points with environment (continuation)

Names

How do names make programming easier?

Names

How do names make programming easier?

Names are a key part of *abstraction*

- Abstraction reduces conceptual complexity by hiding irrelevant details
- Names for subroutines: *control abstraction*
- Names for classes: *data abstraction*

Bindings

A *binding* is an association of two things, such as:

- Names and the things they name
- A question about how to implement a feature and the answer to the question

Binding time is the time at which the association is made.

- *Language design time*: built-in features such as keywords
- *Language implementation time*: implementation dependent semantics such as bit-width of an integer
- *Program writing time*: names chosen by programmer
- *Compile time*: bindings of high-level constructs to machine code
- *Link time*: final bindings of names to addresses
- *Load time*: Physical addresses (can change during run time)
- *Run time*: bindings of variables to values, includes many bindings which change during execution

Static means before run time; *dynamic* means during run time.

Bindings

What are some advantages of early binding times?

What are some advantages of late binding times?

Bindings

What are some advantages of early binding times?

- *Efficiency*: the earlier decisions are made, the more optimizations are available to the compiler
- *Ease of implementation*: Earlier binding makes compilation easier

What are some advantages of late binding times?

Bindings

What are some advantages of early binding times?

- *Efficiency*: the earlier decisions are made, the more optimizations are available to the compiler
- *Ease of implementation*: Earlier binding makes compilation easier

What are some advantages of late binding times?

- *Flexibility*: Languages that allow postponing binding give more control to programmer
- *Polymorphic code*: Code that can be used on objects of different types is *polymorphic* - SMALLTALK is an example of a language supporting polymorphism

Bindings

What are some advantages of early binding times?

- *Efficiency*: the earlier decisions are made, the more optimizations are available to the compiler
- *Ease of implementation*: Earlier binding makes compilation easier

What are some advantages of late binding times?

- *Flexibility*: Languages that allow postponing binding give more control to programmer
- *Polymorphic code*: Code that can be used on objects of different types is *polymorphic* - SMALLTALK is an example of a language supporting polymorphism

Typically, early binding times are associated with compiled languages and late binding times with interpreted languages.

Lifetimes

The period of time between the creation and destruction of a name-to-object binding is called the binding's *lifetime*.

The time between the creation of an object and its destruction is the *object's lifetime*.

Lifetimes

The period of time between the creation and destruction of a name-to-object binding is called the binding's *lifetime*.

The time between the creation of an object and its destruction is the *object's lifetime*.

Is it possible for these to be different?

Lifetimes

The period of time between the creation and destruction of a name-to-object binding is called the binding's *lifetime*.

The time between the creation of an object and its destruction is the *object's lifetime*.

Is it possible for these to be different?

- When a variable is passed by reference, the binding of the reference variable has a shorter lifetime than that of the object being referenced
- If there are multiple pointers to an object and one of the pointers is used to delete the object, the object has a shorter lifetime than the bindings of the pointers
- A pointer to an object that has been destroyed is called a *dangling reference* and is usually a bug

Object Lifetimes

For objects residing in memory, there are typically three areas of storage, corresponding to different lifetimes:

- *Static* objects: lifetime = entire program execution
  - Globals, static variables in C
- *Stack* objects: lifetime = from the time the function or block is entered until the time it is exited
  - Local variables
- *Heap* objects: lifetime = arbitrary, not corresponding to the entrance or exit of a function or block
  - Dynamically allocated objects, e.g., with `malloc` in C or `new` in C++

Static Allocation

Static Objects

Obvious examples of static objects are global variables.

What other objects are static?

Static Allocation

Static Objects

Obvious examples of static objects are global variables.

What other objects are static?

- The actual program instructions (in machine code)
- Numeric and string literals
- Tables produced by the compiler

Static Allocation

Static Objects

Obvious examples of static objects are global variables.

What other objects are static?

- The actual program instructions (in machine code)
- Numeric and string literals
- Tables produced by the compiler

Static objects are often allocated in *read-only* memory so that attempts to change them will be reported as an error by the operating system.

Static Allocation

Static Objects

Obvious examples of static objects are global variables.

What other objects are static?

- The actual program instructions (in machine code)
- Numeric and string literals
- Tables produced by the compiler

Static objects are often allocated in *read-only* memory so that attempts to change them will be reported as an error by the operating system.

Under what conditions could local variables be allocated statically?

Static Allocation

Static Objects

Obvious examples of static objects are global variables.

What other objects are static?

- The actual program instructions (in machine code)
- Numeric and string literals
- Tables produced by the compiler

Static objects are often allocated in *read-only* memory so that attempts to change them will be reported as an error by the operating system.

Under what conditions could local variables be allocated statically?

The original FORTRAN did not support recursion, allowing local variables to be allocated statically.

Dynamic Allocation

For most languages, the amount of memory used by a program cannot be determined at compile time

- Earlier versions of FORTRAN are exceptions

Some features that require dynamic memory allocation:

- Recursion
- Pointers, explicit allocation (e.g., `new`)
- Higher order functions

Stack-Based Allocation

In languages with recursion, the natural way to allocate space for subroutine calls is on the *stack*.

This is because the lifetimes of objects belonging to subroutines follow a *last-in, first-out (LIFO)* discipline.

Each time a subroutine is called, space on the stack is allocated for the objects needed by the subroutine.

This space is called a *stack frame* or *activation record*.

Objects in the activation record may include:

- Return address
- Pointer to the stack frame of the caller (*dynamic link*)
- Arguments and return values
- Local variables
- Temporary variables
- Miscellaneous bookkeeping information

Heap-Based Allocation

Some objects may not follow a LIFO discipline:

- Space allocated with `new`.
- Space for local variables and parameters in functional languages

The lifetime of these objects may be longer than the lifetime of the subroutine in which they were created.

These are allocated on the *heap*: a section of memory set aside for such objects.

(not to be confused with the data structure for implementing priority queues – a totally different use of the word "heap")

Heaps

The heap is finite: if we allocate too many objects, we will run out of space.

Solution:

deallocate space when it is no longer needed by an object.

- *Manual deallocation*: with e.g., `free`, `delete` (C, Pascal)
- *Automatic deallocation* via garbage collection (Java, C#, Scheme, ML, Perl)
- *Semi-automatic deallocation*, using destructors (C++, Ada)
  - Automatic because the destructor is called at certain points automatically
  - Manual because the programmer writes the code for the destructor

Manual deallocation is a common source of bugs:

- Dangling references
- Memory leaks

Heaps

The heap starts out as a single block of memory.

As objects are allocated and deallocated, the heap becomes broken into smaller blocks, some in use and some not in use.

Most heap-management algorithms make use of a *free list*: a singly linked list containing blocks not in use.

- *Allocation*: a search is done to find a free block of adequate size
  - *First fit*: first available block is taken
  - *Best fit*: all blocks are searched to find the one that fits the best
- *Deallocation*: the block is put on the free list

Fragmentation is an issue that degrades performance of heaps over time:

- *Internal fragmentation* occurs when the block allocated to an object is larger than needed by the object
- *External fragmentation* occurs when unused blocks are scattered throughout memory so that there may not be enough memory in any one block to satisfy a request

Garbage Collection

Garbage collection refers to any algorithm for automatic deallocation.

Variations

- Mark/sweep
  - Variant: compacting
  - Variant: non-recursive
- Copying
  - Variant: incremental
  - Variant: generational

Garbage Collection

During garbage collection, the aim is to deallocate any object that will never be used again.

How do you know if an object will never be used again?

Garbage Collection

During garbage collection, the aim is to deallocate any object that will never be used again.

How do you know if an object will never be used again?

An approximation is the set of objects that are *live*.

An object `x` is *live* if:

- `x` is pointed to by a variable,
  - on the stack (e.g., in an activation record)
  - that is global
- there is a register (containing a temporary or intermediate value) that points to `x`, or
- there is another object on the heap (e.g., `y`) that is live and points to `x`

All live objects in the heap can be found by a graph traversal:

- Start at the *roots*: local variables on the stack, global variables, registers
- Any object not reachable from the roots is *dead* and can be reclaimed

Mark/Sweep

- Each object has an extra bit called the *mark bit*
- *Mark phase*: the collector traverses the heap and sets the mark bit of each object encountered
- *Sweep phase*: each object whose mark bit is not set goes on the free list

name	definition

GC()	`for each root pointer p do`
	`  mark(p);`
	`sweep();`

mark(p)	`if p->mark != 1 then`
	`  p->mark := 1;`
	`  for each pointer field p->x do`
	`  mark(p->x);`

sweep()	`for each object x in heap do`
	`  if x.mark = 0 then insert(x, free_list);`
	`  else x.mark := 0;`

Copying

- Heap is split into 2 parts: *FROM* space, and *TO* space
- Objects allocated in *FROM* space
- When *FROM* space is full, garbage collection begins
- During traversal, each encountered object is copied to *TO* space
- When traversal is done, all live objects are in *TO* space
- Now we flip the spaces – *FROM* space becomes *TO* space and vice-versa
- Since we are moving objects, any pointers to them must be updated: this is done by leaving a *forwarding address*

Copying

name	definition
GC()	`for each root pointer p do` `  p := traverse(p);`

traverse(p)	`if *p contains forwarding address then`
	`  p := *p;  // follow forwarding address`
	`  return p;`
	`else {`
	`  new_p := copy (p, TO_SPACE);`
	`  *p := new_p; // write forwarding address`
	`  for each pointer field p->x do`
	`  new_p->x := traverse(p->x);`
	`  return new_p;`
	`}`

Generational Garbage Collection

A variant of a copying garbage collector

Observation:

the older an object gets, the longer it is expected to stay around.

Why?

Generational Garbage Collection

A variant of a copying garbage collector

Observation:

the older an object gets, the longer it is expected to stay around.

Why?

- Many objects are very short-lived (e.g., intermediate values)
- Objects that live for a long time tend to make up central data structures in the program, and will probably be live until the end of the program

Generational Garbage Collection

A variant of a copying garbage collector

Observation:

the older an object gets, the longer it is expected to stay around.

Why?

- Many objects are very short-lived (e.g., intermediate values)
- Objects that live for a long time tend to make up central data structures in the program, and will probably be live until the end of the program

Idea:

instead of 2 heaps, use many heaps, one for each "generation"

- Younger generations collected more frequently than older generations (because younger generations will have more garbage to collect)
- When a generation is traversed, live objects are copied to the next-older generation
- When a generation fills up, we garbage collect it

Reference Counting

The problem

- We have several references to some data on the heap
- We want to release the memory when there are no more references to it
- We don't have garbage collection "built-in"

Idea:

Keep track of how many references point to the data, and free it when there are no more.

- Set reference count to 1 for newly created objects
- Increment reference count whenever we create a pointer to the object
- Decrement reference count whenever a pointer stops pointing to the object
- When an object's reference count becomes 0, we can free it

Reference Counting Example

```
template<class T>
class RefTo {
   T *p;
public:
   RefTo (T* obj = NULL) : p(obj)
     { if (p) p->refCount++; }
   RefTo (const RefTo<T>& r) : p(r.p)
     { if (p) p->refCount++; }
   ~RefTo ()
     { if (p && --p->refCount == 0) delete p; }
   RefTo<T>& operator= (const RefTo<T>&);
   ...
}
```

Reference Counting Example

```
const RefTo<T>& RefTo::operator= (const RefTo<T>& r) {
  if (r.p)
    r.p->refCount++;

  if (--p->refCount == 0)
    delete p;

  p = r.p;

  return *this;
}
```

Comparison

What are the advantages and disadvantages of mark/sweep, copying, and reference counting?

Comparison

What are the advantages and disadvantages of mark/sweep, copying, and reference counting?

- Garbage Collection: large cost, occurs rarely
- Reference Counting: small cost occurs frequently
- GC is faster overall, but RC has more consistent performance

Comparison

What are the advantages and disadvantages of mark/sweep, copying, and reference counting?

- Garbage Collection: large cost, occurs rarely
- Reference Counting: small cost occurs frequently
- GC is faster overall, but RC has more consistent performance

Costs of garbage collection

$$\begin{array}{lcl} L & = & \text{amount of storage occupied by live data} \\ M & = & \text{size of heap} \end{array}$$

- Mark/sweep: $O(L) + O(M) = O(M) \quad \text{since } M > L$
- Copying: $O(L)$
  experimental data for LISP: $L \approx 0.3 * M$

Comparison

What are the advantages and disadvantages of mark/sweep, copying, and reference counting?

- Garbage Collection: large cost, occurs rarely
- Reference Counting: small cost occurs frequently
- GC is faster overall, but RC has more consistent performance

Costs of garbage collection

$$\begin{array}{lcl} L & = & \text{amount of storage occupied by live data} \\ M & = & \text{size of heap} \end{array}$$

- Mark/sweep: $O(L) + O(M) = O(M)$ since $M > L$
- Copying: $O(L)$
  experimental data for LISP: $L \approx 0.3 * M$

Mark/Sweep costs more, but is able to use more memory.

Scope

The region of program text where a binding is active is called its *scope*.

Notice that scope is different from lifetime.

Kinds of scoping

- *Static* or *lexical*: binding of a name is determined by rules that refer only to the program text
  - Typically, the scope is the smallest block in which the variable is declared
  - Most languages use some variant of this
  - Scope can be determined at compile time
- *Dynamic*: binding of a name is given by the most recent declaration encountered during run-time
  - Used in SNOBOL, APL, some versions of LISP

Scoping example

```
var x = 1;

function f () { print x; }

function g () { var x = 10;  f(); }

function h () { var x = 100; f(); }

f(); g(); h();
```

Scoping	Output
Static	1 1 1
Dynamic	1 10 100

Static Scoping: Nested Scopes

Some languages allow nested subroutines or other kinds of nested scopes.

Typically, bindings created inside a nested scope are not available outside that scope.

On the other hand bindings at one scope typically are available inside nested scopes. The exception is if a new binding is created for a name in the nested scope.

In this case, we say that the original binding is *hidden*, and has a *hole* in its scope.

Some languages allow nested scopes to access hidden bindings by using a *qualifier* or *scope resolution operator*.

- In Ada, $A.X$ refers to the binding of X created in subroutine A, even if there is a lexically closer scope
- In C++, $A :: X$ refers to the binding of X in class A, and $:: X$ refers to the global scope

Static Scoping: Nested Subroutines

How does a nested subroutine find the right binding for an object in a outer scope?

Static Scoping: Nested Subroutines

How does a nested subroutine find the right binding for an object in a outer scope?

- Maintain a *static link* to the "parent frame"
- The parent frame is the most recent invocation of the lexically surrounding subroutine
- The sequence of static links from the current stack frame to the frame corresponding to the outermost scope is called a *static chain*

Static Scoping: Nested Subroutines

How does a nested subroutine find the right binding for an object in a outer scope?

- Maintain a *static link* to the "parent frame"
- The parent frame is the most recent invocation of the lexically surrounding subroutine
- The sequence of static links from the current stack frame to the frame corresponding to the outermost scope is called a *static chain*

Finding the right binding

- The level of nesting can be determined at compile time
- If the level of nesting is j, the compiler generates code to traverse the static chain j times to find the right stack frame

Static Scoping: Declaration Order

What is the scope of x?

```
{
  statements1;
  var x = 5;
  statements2;
}
```

- C, C++, Ada: `statements2`
- JavaScript, Modula-3: entire block
- Pascal: entire block, but not allowed to be used in `statements1`!

Declarations and Definitions

C and C++ require names to be declared before they are used.

This requires a special mechanism for recursive data types.

C and C++ solve the problem by separating *declaration* from *definition*.

- A *declaration* introduces a name and indicates the scope of the name
- A *definition* describes the object to which the name is bound

Declarations and Definitions: Example

```
struct manager;                    // Declaration
struct employee {
  struct manager* boss;
  struct employee* next_employee;
  ...
};

struct manager {                   // Definition
  struct employee* first_employee;
  ...
};
```

Redeclaration

Interpreted languages often allow *redeclaration*: creating a new binding for a name that was already given a binding in the same scope.

```
function addx(int x) { return x + 1; }

function add2(int x)
{
  x := addx(x);
  x := addx(x);
  return x;
}

function addx(int x) { return x + x; }
```

What happens if we call `add2(2)` *?*

Redeclaration

Interpreted languages often allow *redeclaration*: creating a new binding for a name that was already given a binding in the same scope.

```
function addx(int x) { return x + 1; }

function add2(int x)
{
  x := addx(x);
  x := addx(x);
  return x;
}

function addx(int x) { return x + x; }
```

What happens if we call `add2(2)` *?*

In most languages, the new definition replaces the old one in all contexts, so we would get 8.

Redeclaration

Interpreted languages often allow *redeclaration*: creating a new binding for a name that was already given a binding in the same scope.

```
function addx(int x) { return x + 1; }

function add2(int x)
{
  x := addx(x);
  x := addx(x);
  return x;
}

function addx(int x) { return x + x; }
```

What happens if we call `add2(2)` *?*

In most languages, the new definition replaces the old one in all contexts, so we would get 8.

In ML, the new meaning only applies to later uses of the name, not previous uses. ML would give 4.