CS655 class notes

Raphael Finkel

December 8, 2021

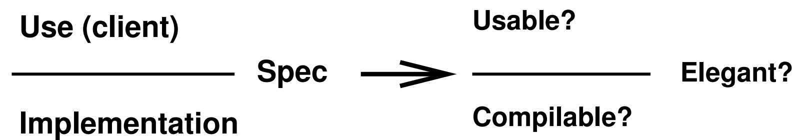
1 Intro

Class 1, 8/23/2021

- Handout 1 — My names
  - Mr. / Dr. / Professor / —
  - Raphael / Rafi / Refoyl
  - Finkel / Goldstein
- Extra 5 minutes on every class? What is a good ending time?
- Plagiarism — read aloud from handout 1
- Assignments on web and in handout 1.
- E-mail list: cs655001@cs.uky.edu; instructor uses to reach students.
- All students will have MultiLab accounts, although you may use any computer you like to do assignments. But your programs must run on MultiLab computers, because that's how they will be graded.
- textbook — all homework comes from here
- Oral assignments are end-of-chapter assignments. Assignment for Chapter 1 exercises (Friday).

2 Software tools

A programming language is an example of a **software tool**.



3 McLennan's Principles (elicit first)

4 Algol-like languages: review

- First generation: Fortran
  - constructs based on hardware
  - lexical: linear list of statements
  - control: sequence, **`goto`**, **`do`**, subroutines using reference parameter mode
  - data: arithmetic, including complex; arrays with a 3-d limit
  - name: separate scopes; common area
- Second generation: Algol 60
  - lexical: free format; keywords
  - control: nested; **`if`**, **`while`**, **`for`**, but baroque; subroutines with value and name parameter modes.
  - data: generalized arrays, but no complex
  - name: nested scopes with inheritance and local override
- Third generation: Pascal (return to simplicity)
  - data: user-defined types; records, enumerations, pointers
  - control: subroutines with value and reference parameter modes; **`case`** statement
- Fourth generation: Ada (abstract data types)
  - lexical: bracketed syntax
  - name: modules with controlled export; generic modules
  - control: concurrency with rendezvous
- Fifth generation: Other directions

  - dataflow
  - functional. We will study ML and Lisp.
  - object-oriented. We will study Smalltalk.
  - declarative (logic). We will study Prolog.

5 Theme: binding time

- Class 2, 8/25/2021
- There is a range from early to late.
  - language-definition time (example: the fact that constants exist)
  - compile time (example: values of constants in Pascal) We call compile-time bindings **static**.
  - link time (example: version of `printf` in C)
  - elaboration time (example: value of **`final int`** in Java)
  - statement-execution time (example: value of **`int`** variable) We call execution-time bindings **dynamic**.
- Early binding is most efficient.
- Late binding is most capable.

6 Block structure

- Adequately difficult example: book 24:21
- Class 3, 8/27/2021: Discussion of questions from Chapter 1.
- Class 4, 8/30/2021
- Introduced in Algol.
- A block is a nestable name scope.
- Identifiers can be local, nonlocal, or global with respect to a block.
- Nonlocal identifiers: the language must define whether to
  - inherit (typically allowed if there is no conflict)
  - override (typically true if there is a conflict)
  - require explicit import and export

- At **elaboration time**, constants get values, dynamic-sized types are bound to their size, space is allocated for variables.
- Definition: the **non-local referencing environment (NLRE)** of a procedure or block of code is the binding of non-local identifiers (typically variables, but also constants, types, procedures, and labels) to values.
- **Deep binding**: The NLRE of P is determined (bound) at the time that P is elaborated (and is the RE of the elaborating scope).
- **Shallow binding**: The NLRE of P is determined at the time that P is invoked (and is the RE of the calling scope).

7 Imperative languages

- Class 5, 9/1/2021
- Imperative languages involve statements that modify the current state by changing the values of variables.
- A **variable** is an identifier bound (usually statically) to a type, having a value that can change over time. The **L-value** of a variable is the use of a variable on the left side of an assignment (think of "address"); the **R-value** of a variable is its use on the right side (think of "current value").
- A **type** is a set of values, associated (mostly statically) with operations defined on those values. Type **conversion** means expressing a value of one type as a value of another type.
  - **coercion:** implicit conversion
  - **cast:** explicit conversion
  - **non-converting cast:** rarely needed. **`qua`** operator of Wisconsin Modula, **`reinterpret_cast<>`** of C++.
- An **operation** is a function or an operator symbol as shorthand. It can be heterogeneous.
  - operators have **arity** (example: unary, binary), precedence, associativity
  - operators may be infix (+), prefix (unary −), postfix (−>)
  - operators may have short-circuit (lazy) semantics

- An operation is **overloaded** if its identifier or operator symbol has multiple visible definitions. Overloading is resolved (usually statically) by arity, operand types, and return type. Overloading resolution can be exponentially expensive. For instance, say we have four versions of +, depending on whether they take integers/floats and whether they return integers/floats. Then how do you resolve (a + b) + (c + d)?
- A **primitive type** (or **basic type**) has no separately accessible components. Examples: integer, character, real, Boolean.
- A **structured type** has separately accessible components. Examples: pointer (dereference), record (field select), array (subscript), disjoint union (variant select). An **associative array** is an array whose index type is string.
- A **constant** is like a variable, but it has no L-value and an unchanging R-value. In Java, it's denoted by the modifier `final`.

8 Iterators

- Iterators allow us to generalize `for` loops.
  - The control variable of the `for` loop ranges over a set of values generated piecemeal by an **iterator**. book 39:9-10.
  - The iterator is like a procedure, taking parameters and returning values of a specified type.
  - The iterator uses a `yield` statement to return a value, but it maintains its RE (and its program counter) in order to continue on demand from the `for` loop.
  - A useful language-supplied iterator is `int upto(low, high)`, which yields all the values in the specified range.
- Iterators are especially useful for generating combinatorial structures.
  - Algorithm for generating all binary trees of n nodes: book 41:11
  - Same thing in Python, using "generators":

```
def binGen(size):
   if size > 0:
      for root in range(size):
         for left in binGen(root):
            for right in binGen(size - root - 1):
               yield("cons(" + left + "," + right + ")")
   else:
      yield "-"

for aTree in binGen(3):
   print(aTree)
```

- Class 6, 9/3/2021
- Trace of binGen(3).

- Another example: yield all nodes in a tree (in pseudo-Python)

```
def treeNodes(tree):
  if tree != null:
    for element in treeNodes(tree.left):
      yield element
    yield tree.value
    for element in treeNodes(tree.right):
      yield element
```

- Python 3 gives us a nice shorthand:

```
def treeNodes(tree):
  if tree != null:
    yield from treeNodes(tree.left)
    yield tree.value
    yield from treeNodes(tree.right)
```

This construct can improve efficiency. JavaScript also has it: `yield*`. It **delegates** the yielding to another iterator.

- Another example: all combinations $C(n, k)$:

```python
def comb(n,k,start):
  if k == 0:
    yield ""
  elif k+start <= n:
    for rest in comb(n,k-1,start+1):
      yield str(start+1) + "," + rest
    yield from comb(n,k,start+1)

for result in comb(6,3,0):
  print(result)
```

- Same thing, in JavaScript:

```javascript
function* comb(n, k, start) {
   if (k === 0) {
      yield '';
   } else if (k+start <= n) {
      for (const first of comb(n, k-1, start+1)) {
         yield `${start+1},${first}`;
      }
      yield* comb(n, k, start+1);
   };
} // comb

for (const result of comb(6,3,0)) {
    console.log(result);
}
```

9 Macro package to embed iterators in C

- Class 7, 9/10/2021
- Macros are **IterSTART**, **IterFOR**, **IterDONE**, **IterSUB**, **IterYIELD**.
- Usage: book 48:14
- Implementation
  - `setjump` and `longjmp` for linkage between **for** and the controlling iterator, between **yield** and its controlled loop.

  - Padding between stack frames to let `longjmp()` be called without harming frames higher on the stack. Three integers is enough in Linux on an i686.
  - A `Helper` routine to actually call the iterator and act as padding.
  - The top frame must be willing to assist in creating new frames.

10 Power loops

- How can you get a dynamic amount of **`for`**-loop nesting?
- Application: n queens book 57:29
- Usual solution: single **`for`** loop with a recursive call.
- Cannot use that solution in Fortran, which does not allow recursion.
- Solution: Power loops. book 57:28
- Implementation: Only needs branches, no recursion. book 59:31
- How general is this facility?
- Do power loops violate principle 20?

11 General coroutines

- Class 8, 9/13/2021
- Problem: binary-tree node-equality test in symmetric order
- Solution
  - Independently advance in each tree book Figure 2.8
  - Each coroutine has its own stack; main has its own stack.
  - All the stacks are joined via static-chain pointers into a cactus stack.
  - A scope must not exit until all its children exit, or we must use a reference count. This is an example of the dangling NLRE problem.
- Syntax (Simula 67)
  - Records have initialization code that may execute **`detach`**.
  - Another coroutine may resume it via explicit **`call`**.

  - Ole-Johann Dahl, designer of Simula 67, got the Turing award in 2001.

- Class 9, 9/15/2021
- Solution in Python using ”generators”. See binEqual.txt from class web page.

12 IO

- Class 10, 9/17/2021
- Attempt to strip a programming language of all non-essential elements.
- What’s left: `goto` with parameters, hence formal-actual bindings.
- Parameters can be integer, anonymous function, or continuation.
- A function is passed by **closure**: pointer to code, pointer to NLRE.
- A **continuation** is a function whose parameters have already been bound. It is passed by a closure along with the parameters.
- Examples book 50:15 and following
- re-work the Io examples io.txt

Class 11, 9/20/2021 Chapter 2 exercises.

13 ML-Introduction

- Class 12, 9/24/2021
- This material is out of order, presented now so students can begin to work on the next programming assignment.
- ML is **functional**, but we are particularly interested in its type system, with these important aspects.
  - The language is **strongly typed**.
  - The compiler **infers** the types of identifiers if possible.
  - **Higher-order types** are easily represented.
  - Types can be **polymorphic**, that is, expressed with respect to type identifiers.

- Examples, from online file examples.ml
- Class 13, 9/27/2021: Continue ML examples.
- Class 14, 10/1/2021: Last ML examples: recursive datatypes

14 Parameters

- Nomenclature
  - **Formal parameter**: the identifier that the procedure uses to refer to the parameter; it is elaborated when the procedure is invoked.
  - **Actual parameter**: the expression that computes the value of the parameter; it is evaluated in the environment of the caller.
  - **Linkage**: The machine-oriented mechanism by which the caller A causes control to jump to the called procedure B, including initializing B's stack frame, passing parameters, passing results back to A, and reclaiming B's stack frame when it has completed.
- Parameter-passing modes
  - **value mode**: The formal has its own L-value, initialized to the R-value of the actual. The language design may restrict the formal to read-only use. Value mode is the only mode available in C.
  - **result mode**: The formal has its own L-value, not initialized. When B returns, the formal's L-value is copied back to the actual (which must have an L-value, so the actual cannot be an arbitrary expression). Result mode was introduced in Algol-W.
  - **value-result mode**: The formal has its own L-value, initialized to the R-value of the actual. As B returns, its value is copied back to the actual (which must have an L-value). Value-result mode was introduced in Algol-W.
  - **reference mode**: The formal has the same L-value as the actual (which must have an L-value, which might be a temporary location). The language may allow the programmer to specify read-only use of the formal parameter. Reference mode is the only mode available in Fortran.

- **name mode**: All accesses to the formal parameter re-evaluate the actual (either for L-value or R-value, depending on the access to the formal). This evaluation is in the RE of the caller, typically by means of a compiled procedure called a **thunk**. Name mode was invented for Algol-60 and never used again.
- **macro mode**: The formal parameter is expanded as needed to the text of the actual parameter, which by itself need not be syntactically complete. No modern language uses this mode.

15 Types

- A **type** is a property of an R-value or of an identifier that can hold R-values.
- The property consists of a set of values.
- **Strong typing** means the compiler
  - knows the type of every R-value and identifier
  - enforces type compatibility on assignment and formal-actual binding
    - **compatible** means type equivalent, a subtype, or convertible
- A **subtype** consists of a subset of the values
  - Assignment and formal-actual binding require a dynamic check.
  - Subtype examples: range of integers, subclass of class
  - `subTypeVar := baseTypeVar`
- Class 15, 10/4/2021
- Type equivalence
  - **structural equivalence**: expand type to a canonical string representation; equivalence is string equality.
    - lax: ignore field names, ignore array bounds, ignore index type, flatten records.
    - pointers require that we handle recursive types (and still build finite strings)

- **name equivalence**: reduce a type to an instance of a type constructor
    - **type constructors**: **`array`**, **`pointerTo`**, **`enum`**, **`struct`**, **`derived`**.
    - lax (**declaration equivalence**): multiple uses of one type constructor are equivalent
- Implementing structural type equivalence: exercise 3.8 at end of chapter.

16 Dimensions

- Class 16, 10/6/2021
- book Figure 3.7
- `www.cs.uky.edu/~raphael/convert.cgi`
- Applies to reals.
- Can be embedded into a strong type system.

Class 17, 10/8/2021 Chapter 3 exercises

17 Unusual first-class values

- A **first-class value** can be returned from a function. In languages with variables, a first-class value can be stored in a variable.
- A **second-class value** can be an actual parameter.
- A **third-class value** can be used "in its ordinary way".
- Class 18, 10/11/2021
- Labels and procedures
    - Usually third class values; the "ordinary way" is in a **`goto`** statement or a procedure-call statement.
    - They could be second class. They must be passed as a closure. For a label, the closure includes the RE of its target, so the stack can be unwound to the right place. For a procedure, the closure includes its NLRE to resolve non-local references.

    - To make them first class, we still need to build a closure, which we can then store in a variable. But the lifetime of that variable might exceed the life of the RE stored in the closure. This is the **dangling-NLRE problem**. book 76:11

- To resolve the dangling-NLRE problem
    - Let it happen and call the program "erroneous".
    - Prevent it from happening by restricting the language.
        - Don't let labels or procedures be first-class: Pascal
        - Don't let scopes nest, so there is no need for closures (for procedures; labels are still problematic): C
        - Only allow top-level procedures (or labels) be first-class: Modula-2
    - Let it happen and make it work: allocate all frames from the heap and use explicit release (reference counts suffice).

18 Lisp introduction

- Class 19, 10/13/2021
- Lisp is **homoiconic**: programs and data structures have the same form, so one can execute data and one can manipulate program.
- Lisp is **functional** (functions have no side effects, and there are no variables)
- Lisp has been very influential and was widely used in AI.
- Lisp by examples
- Lisp deep binding (in examples)
- Class 20, 10/15/2021
- The Lisp metacircular interpreter book 135:31 ff, also on web page.
- Pure Lisp (a restriction for Assignment 3): no use of **`set`**, **`rplaca`**, **`rplacd`**, and a few other non-functional features. Although **`defun`** does change the context (introducing new functions always has a side effect), we allow it.

19 Haskell

- Class 21, 10/18/2021
- Haskell is a functional programming language much like ML, with additional features.
- Haskell uses indentation for grouping, much like Python. This design makes programs compact, but harder to read (if routines are long) and very difficult for blind programmers to construct.
- All functions are fully curried.
- Haskell evaluates all expressions **lazily**.
- Examples online

20 What does polymorphic mean?

People use the term **polymorphic** to mean various features.

- Class 22, 10/20/2021
- Static procedure **overloading** with compile-time resolution (Ada, Java). The type of the procedure (its **signature**) determines whether it is a viable candidate for resolving the overloading.
- Dynamic method binding (**overriding**, with **dynamic dispatch**. (Java, Smalltalk, C++ deferred binding). The dynamic type (class) of the value (object) determines which procedure (method) to invoke.
- Types described by type identifiers (perhaps with the compiler inferring the types of values) (ML, Haskell). Here, the dynamic type constraints on the parameter and return value determine the effective type of the function.
- Passing an ADT as a parameter (Russell).
- Generic packages (Ada), templates (C++), generics (Java).

21 Can types be second or first class?

- Letting a type be second-class is a step toward polymorphism.
  - Second-class types allow generic packages (Ada), templates (C++).

  - However, we only allow types to be passed at compile time during generic-package instantiation.
  - One can restrict the range of the actual type (the "type of the type") by a Java-like interface (or Haskell-like class).

- An attempt at first-class types: **`dynamic`**. book 112:68 Actually, this idea attempts to extend strong typing to dynamic types; it is not the same as making types first-class.
- Another attempt at first-class types: Russell.
  - But what Russell calls a "type" we would call an ADT (abstract data type), which is something like a Java class.
  - So a Russell "type" actually introduces dynamic class definitions.
  - We will see dynamic class definitions in Smalltalk later.
- Java reflection: class `Class`. Example from Cycle Shop at Home.

22 The Lambda Calculus

- Mathematical foundation of ML and of Lisp.
- Three kinds of **term**
  - **identifier**, such as `x`
  - **abstraction**, such as `(λ x . (* x 2))`
  - **application**, such as `(f x)`
- Syntax of terms
  - parentheses are optional; application and abstraction are left-associative, and application has a higher precedence.
  - Curried functions may be written without currying:
    `(λ x . (λ y . (λ z . T))) = (λ x y z . T)`
- Class 23, 10/22/2021
- Identifiers can be **free** or **bound** (or simply missing) in terms.
- `x` is bound in `λ x . T`.
- `x` is free in `T` if:

  - `T` is just `x`
  - `T` is `(f p)`, and `x` is free in either `f` or `p`.
  - `T` is `(λ y . T)`, `x` is not `y`, and `x` is free in `T`.
  - Examples: book 153:49

- Simplification by β reduction: book 153:50.
  - applicative order: evaluate inner λ first.
  - normal order: evaluate outer λ first.
  - Church-Rosser theorem: if `T` $\rightarrow$ `S` and `T` $\rightarrow$ `R`, then there is some `U` such that `S` $\rightarrow$ `U` and `R` $\rightarrow$ `U`
- Renaming by α conversion: book 153:52
- Normal form
  - β reduction until no more reduction is possible.
  - It is not always possible to reduce to a normal form: book 155:55
- Class 24, 10/27/2021 Exercises, Chapter 4
- Class 25, 10/29/2021
- **Combinator**: term with no free identifiers. Important combinator Y: `Y = (λ f . (λ x . f (x x)) (λ x . f (x x)))` 153:57
- Simplification by η reduction: `(λ x . F x)` $\stackrel{\eta}{\rightarrow}$ `F`
- Connection between Lambda Calculus and ML. book 156:60

23 Smalltalk

- Examples (online)
- Class 26, 11/01/2021
- More examples (online)
- Class 27, 11/03/2021
- More examples (online)
- The two hierarchies of classes in Smalltalk
  - **`class`** gives the is-a hierarchy.
  - **`superclass`** gives the subclass-of hierarchy.

24 Intellectual history of object oriented programming

- Class 28, 11/05/2021
- Records in Cobol, Pascal. Fields are variables, always visible.
- Records (called classes) in Simula67. Fields can also be procedures. Their NLRE is the record in which they sit. Already some object-orientation: Subclasses inherit fields with possibility of override, and there is some control over visibility.
- Abstract data types (ADTs) in languages like CLU ("clusters"), Modula ("modules"), Ada ("packages"). Separation of specification from implementation. Typically ADTs export a type and some operations. Clients then create instances of that type, either on the stack as local variables or from the heap.
- Monitors, which are ADTs with concurrency control. Unfortunately, concurrency control protects program, not data. We'll say more about monitors later.
- Classes (Smalltalk, C++, Java). Do not export types, only variables and procedures. The entire class becomes a type.

25 Object-oriented programming: What is it?

- Nomenclature: **Objects** (values) are instances of **classes** (types). They communicate by sending **messages** (procedure calls) to each other to invoke **methods** (procedures). The state of an object is defined by the values of its **instance variables**. The set of methods that instances of a class accepts constitute its **protocol**. Together, the methods and the instance variables are called the **members** of a class.
- Class 29, 11/08/2021
- **Data encapsulation**: One may only affect the state of an object by invoking its methods. (Doesn't hold if instance variables are exposed.)
- **Inheritance**: Subclasses (a form of subtype) inherit the members of their superclass. Programmers introduce subclasses either to **specialize** or to **reuse code**.

- **Overriding — Deferred binding**: ″An instance method **overrides** all accessible instance methods with the same signature in superclasses, enabling dynamic dispatch.″ [Java Puzzlers, p. 180] Resolution must be dynamic, at invocation time.
- **Overloading — Static binding**: ″Methods in a class **overload** on another if they have the same name and different signatures. [Java Puzzlers, p. 181] Resolution is at compile time.

26 Rules for method resolution in Java

- `coderanch.com/t/417622/certification/Golden-Rules-widening-boxing-varargs` by Anand Shrivastava
- Rules
  1. Primitive Widening > Boxing > Varargs.
  2. Widening and Boxing (WB) not allowed.
  3. Boxing and Widening (BW) allowed.
  4. While overloading, Widening + vararg and Boxing + vararg can only be used in a mutually exclusive manner i.e. not together.
  5. Widening between wrapper classes not allowed
- Examples

Overloaded methods	**Invocation**	**Called**	**Rule**(s)
f(Integer i), f(long l)	f(5)	long	1
f(int...i), f(Integer i)	f(5)	Integer	1
f(Long l), f(int...i)	f(5)	int...i	2, 1
f(Long l), f(Integer...i)	f(5)	Integer...i	2, 1
f(Object o), f(Long l)	f(5)	Object o	2, 3
f(Object o), f(int...i)	f(5)	Object o	3, 1
f(Object o), f(long l)	f(5)	long l	3, 1
f(long...l), f(Integer...i)	f(5)	ambiguous	4
f(long...l), f(Integer i)	f(5)	Integer	1
f(Long l)	Integer i; f(i)	error	5
f(Long l), f(long...l)	Integer i; f(i)	long...	5, 1

27 Control over information hiding

Class 30, 11/10/2021

language	mode	other instance of same class	related[1]	inherited by subclass	other
Smalltalk	variable	n	-	y	n
Smalltalk	method	y	-	y	y
C++/Java	public	y	y	y	y
C++/Java	protected	y	y	y	n
C++/Java	private	y	y/n	n	n
Java	package-private (default)	y	y	y[2]	n
Eiffel	(default)	y	y	y	y
Eiffel	specified	y[3]	y	y	n
Eiffel	none	n	n	y	n

28 Unusual abilities of Java

- Interfaces
  - The interface provides signatures of methods and declarations of variables.
  - A class can choose to **implement** a list of interfaces.
  - The class is then obligated to provide those methods and variables.
  - Others can assume the class has these methods and variables.
- Effectively homoiconic, but not straightforward
  - serialization: instance $\leftrightarrow$ string
  - introspection: class $\leftrightarrow$ instance of Class

[1] ”related” means friend class (C++), class in the same package (Java), or class to which a member is explicitly exported (Eiffel).

[2] y in same package; n in a different package. It is possible to subclass a class from a different package.

[3] y only if explicitly related.

29 Epilogue about object-oriented programming

- A new view of types: The type of a value is the protocol it accepts.
- Types as first-class values: Smalltalk treats types as values.
  - All values are created from the heap.
  - Types can outlive the environment in which they are created, but deep binding of nonlocal variables does not produce dangling pointers, because of heap allocation.
  - Methods may be added dynamically to types; these changes affect all existing members of the type. This deferral is cognate to shallow binding: The protocol of a value is determined at the time of invocation, not at time of elaboration.
  - This deferral seems to be a consequence of the way methods are introduced; they are not introduced at type-elaboration time.

30 Other object-oriented ideas

- Class 31, 11/12/2021
- **Duck typing**: if an object satisfies the interface of A, it can be used wherever an instance of A is expected, including assignment into A variables.
  - Like structural equivalence, but determined dynamically.
  - The language **go** uses a static structural type system.
  - Generic templates (such as ”the type must satisfy a given interface”) are applied statically, and the type must fulfil the entire interface, even if (dynamically) only a portion is needed.
  - Objections to duck typing: method names may be ambiguous in English, so implementing a method by some name is not the same as providing the expected semantics of that method.
  - C#: modifier *dynamic* on a formal parameter means to defer type checking to runtime and only ensure that the called methods exist.
  - Java: reflection allows the program to determine at runtime if it provides the necessary methods.

  - JavaScript and Python use duck typing; any method call is valid if the object provides it.

- **Go** disallows overloading of methods and operators, which are "occasionally useful but ... confusing and fragile in practice."

31 Concurrent programming

- We will not cover this chapter in depth.
- Basic idea: multiple **threads** of execution.
  - Need to be able to start (and maybe stop) threads.
    - **fork**
    - **cobegin** and **coend**
    - process call, like procedure call (Modula, Go)
  - Threads might need a nonlocal referencing environment, leading to cactus stacks (as in Simula).
    - Can avoid cactus stacks by requiring that threads start at top-level procedures.
  - Threads that share referencing environments must coordinate activities.
    - Mutual exclusion: Preventing simultaneous execution of **critical regions**.
      - **semaphores**: `up` and `down` operations.
      - Class 32, 11/15/2021
      - **mutexes**: based on semaphores, define critical regions.
        - In Java, every object has an implicit mutex; the **`synchronize`**`(object)` syntax lets one acquire/release.
        - Java also has a `Lock` interface.
      - **conditional critical regions**: language forces guarded access to shared variables and organizes conditional waiting. book p. 222
      - **monitors**: mutually exclusive guarded procedures protecting shared variables that are packaged into the same module. book p. 225
    - Synchronization: Blocking until a situation is right.

        - semaphores, but initialized to 0 instead of 1.
        - eventcounts and sequencers.
        - barriers
    - Programming errors and remedies
        - Not recognizing critical regions.
            - Some access to shared variables are atomic; it depends on the language. In Java, access to **`int`** is atomic, but access to **`double`** is not; all access to **`volatile`** variables is atomic.
        - Deadlock: cycle in the waits-for graph. Solution: prevent or break the cycle.
        - Livelock: lack of progress, although no threads are blocked; they are all responding to each other.
        - Starvation: a thread makes no progress, even though others do. Standard example: dining philosophers. Solution: entry control.
- Standard examples for synchronization
    - Bounded buffer. Need to apply both mutex and synchronization.
    - Readers-writers problem. Monitors are too conservative. Solutions are prone to starve readers or writers. Platoon methods work.

Class 33, 11/17/2021 Exercises, Chapter 5

32 Non-shared memory

- We omit this section this semester.
- Threads that do not share referencing environments must coordinate activities, typically by passing messages.
    - Rendezvous (Ada) book 241:16,18
    - Remote procedure call
    - Explicit messages (send and receive)
        - Hoare's CSP. book 251:21.

- Messages in Go on static-typed channels. Channel parameters can be indicated as write-only or read-only (or neither). There is a **`select`** statement, which can read from the first available channel or make read non-blocking. See `https://gobyexample.com/goroutines`.

33 Prolog

- Class 34, 11/19/2021
- Examples online.
- Ideas from other languages
  - Lisp: lists are an essential data type.
  - Snobol: backtracking is built in.
  - Logic, including propositional calculus: implications
- Nomenclature and syntax
  - A **term** is a lower-case functor followed by an optional comma-separated list of fields. Examples:

    ```
    a
    foo(a, X)
    ```

  - A **field** can either be a term or an upper-case variable.
  - A **rule** is a left-hand side term with an optional right-hand side. Examples:

    ```
    good(Food) :- edible(Food), nutritious(Food) .
    edible(water) .
    ```

  - A **right-hand side** is the symbol **:-** followed by a comma-separated list of terms.
  - If a rule has no right-hand side, we call it a **fact**.
  - A **program** is a set of rules.
  - A **query** is the symbol `?-` followed by a comma-separated list of terms. Examples:

    ```
    good(water) .
    good(Food) .
    good(Food) , liquid(Food).
    ```

- The query-solution algorithm
  - Backtrack, controlled by failure and success.
  - For each rule, try to match the head with the query (functor and arity). Fail if no more rules. Failure on next step leads to retry.
  - for each match, try to unify the query with the head (might be 0, 1, or more solutions). Fail if no more unifications. Failure on next step leads to retry.
  - for each unification, try to satisfy each term of the right-hand side recursively. Failure on any term leads to retry. Succeed if the last term succeeds.
  - We did not cover the following items.
  - Difference lists: book 282:28
- We did not cover the following items.
- Prolog-like features of other languages
  - CLPR example
  - Puzzle Lingo examples, smodels
  - Other *lparse* examples, like unattackedQueen or tile.

34 Formalizing programming languages

- Class 35, 11/22/2021
- Formalizing syntax: What is the appearance of a correctly "spelled" program?
  - BNF (introduced for Algol 60)
  - Extensions to BNF
  - Attributed grammars to cover type correctness
- Formalizing semantics: What is the meaning of a program?
- Axiomatic semantics (Hoare 1967) book 363:11 Based on placing **assertions** in the program and providing **axioms** that allow one to prove statements of the form `{P} S {Q}` meaning if predicate `P` is true before statement `S` starts, then after statement `S` completes, if it does, then `Q` must hold."

- Axiom of assignment: $\{Q_{x \to E}\}$ `x := E` $\{Q\}$
- Example: {`y = 12`} `x := y + 2` {`x = 14`}
- Weak and strong predicates
  1. if $P \Rightarrow Q$, we say that P is **stronger than** Q.
  2. Strengthening a precondition P in {P} S {Q} **weakens** the entire statement; weakening the precondition **strengthens** the statement.
  3. Axioms try to show the strongest statements, that is, the weakest preconditions for which the statement always holds.
- Axiom of selection (**`if`** statements)
  $\{B \land P\}\ S_1\ \{Q\},\ \{\neg B \land P\}\ S_2\ \{Q\} \vdash$
  $\{P\}$ **if** B **then** S_1 **else** S_2 $\{Q\}$
- Axiom of iteration (**`while`** statements)
  $\{B \land I\}\ S\ \{I\} \vdash$
  $\{I\}$ **while** B **do** S $\{\neg B \land I\}$
  but no guarantee of completion.

- Extended example: factorial

```
1  {true}
2  {1 = 1!}
3  count := 1;
4  {1 = count!}
5  answer := 1;
6  {answer = count!}
7  while count != n do
8      {answer = count!}
9      count := count + 1;
10     {answer = (count-1)!}
11     answer := answer * count;
12     {answer = count!}
13 end;
14 {answer = count! ∧ count = n}
15 {answer = n!}
```

- But the loop might not terminate: if `n` < 1.
- Evaluation
  - It is possible to prove small programs correct.
  - Complex control structures (like **`break`** and concurrency) are very hard to model.
  - Designing the proper overall preconditions and postconditions of a piece of code is at least as hard as designing the code.
  - Does not prove termination.
  - Only as good as the preconditions and postconditions
  - Led to a fad of proving programs correct
  - Led to a fad of teaching programming by precondition / postcondition / loop invariant.
- Extension: weakest preconditions (Dijkstra 1975). Can prove termination, but hard to discover loop invariants.

35 Formalizing semantics: Denotational semantics

- Class 36, 11/29/2021

- Components
  - Abstract syntax: Already-parsed source program
  - Semantic domains: Mathematical sets representing values that arise in describing program semantics.
  - Semantic functions: Functions that take syntax and yield values in semantic domains
- Simplest language: Binary literals. book 370:23 The semantic function `E` gives the denotation of programs.
- Class 37, 12/1/2021
- Expressions. book 379:34
- Range checks and divide-by-zero possibility, including the $\perp$ symbol for error. book 381:38
- Class 38, 12/3/2021
- Environments to store initialized constants. Declarations update the environment. The semantic function `E` now takes an environment parameter. book 384:43
- Class 39, 12/6/2021
- Variables. Programs now specify what variable is their final meaning. book 388:46
- Assignment statement.
- Conditional and do-n-times statements. book 390:50
- While loop. Define $p_i u$ as the environment one gets after checking the Boolean condition i times, starting in environment u.

correct number of iterations	0	1	2	3	4	∞
$p_0\ u$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$p_1\ u$	u	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
$p_2\ u$	u	u'	$\perp$	$\perp$	$\perp$	$\perp$
$p_3\ u$	u	u'	u''	$\perp$	$\perp$	$\perp$
...						
limit	u	u'	u''	u'''	u''''	$\perp$

36 APL

- Class 40, 12/8/2021
- APL: Primes example from book, 354:9.66