Fri., 30 Oct. 2015

Mark your calendar: Pizza lunch, 1:30-2:30 Nov. 3 with Hyland Software

Exam Weds., Nov. 4 (review Mon. Nov. 2)

Questions about lab?

Today: more on functions--closures
Fri., 6 Nov. 2015

Exams back Monday
Questions on lab?
Today: more on functions
Closures
Exceptions
Closures (Chapter 3, p. 153)

In languages that support “first-class functions,” a function may be a return value from another function; a function may be assigned to a variable. This raises some issues regarding scope!
Closures

JavaScript example:  http://goo.gl/kzUCes

function f(name) {
    var x = "hi there";
    function g() {
        return x+" " + name;
    }
    return g;
}

var k = f("bob");

Function f returns the function g. Therefore, variable k is assigned a function.
Once f is done, how will k (i.e., g) know the values of x and name?
Closures

One solution (NOT the one used by JavaScript!): use the most recently-declared values of variables “name” and “x”. This is called “shallow binding.” Common in dynamically-scoped languages.

Another solution (used by JavaScript and most other statically-scoped languages): bind the variables that are in the environment where the function is defined. This is an illustration of “deep binding” and the combination of the function and its defining environment is called a closure.
Closures

Another example: C#

```csharp
static void Main(string[] args) {
    Func<int,int> g = returnFunc(7);
    Console.WriteLine(g(9));
}

static Func<int,int> returnFunc(int x) {
    Func<int,int> g =
        delegate(int z) { int y = 10; return x+y+z; };
    return g;
}
```

See program Closure.cs in nov6 folder of repo
Exceptions

See program Exc.java in nov6 folder of repo
Mon., 9 Nov. 2015

Exam returns

Questions about labs 7 or 8?

Last words on functions and subroutines

Chapter 9: Object-oriented programming
Last Words on Functions

We’ve considered the following topics:

- Parameter passing (e.g., pass by value, pass by reference)
- Special syntax (default values, named parameters)
- Mechanisms for function calls (activation record stack, static and dynamic pointers, calling sequences)
- Parameter evaluation (applicative, normal, lazy)
- Closures
- Functions as first-class objects (lab 7)
- Exceptions
Last Words on Functions

We didn’t cover:

- Generics (you have seen a lot of this in CMPSC 112)
- Events (we’ll revisit this topic in chapter 13)

Let’s quickly visit one other topic: coroutines. A coroutine is a function that can be suspended and resumed; several coroutines can be active at once, transferring control back and forth between them.
Coroutines

Coroutines are supported in languages such as Simula and Modula-2; useful for writing discrete event simulation code.
Generators in Python

Many other languages have features that allow the implementation of coroutines (even if they are not “built in” to the language). Python has generator functions:

```python
>>> def gen99():
...   for i in range(100):
...      yield i # NOTE: not “return i"

>>> a = gen99() # call the function just once
>>> next(a)
0
>>> next(a)
1
```
Generators in Python

```python
>>> next(a)
2

>>> for i in range(10):
...   print next(a),
...
3 4 5 6 7 8 9 10 11 12

>>> for i in range(10):
...   print next(a),
...
13 14 15 16 17 18 19 20 21 22
```
Generators in Python

Several generators can be active at the same time; see sample program “gen.py” in the shared repository. This isn’t precisely a coroutine example (we don’t have “call” and “response” directly transferring back and forth). See https://docs.python.org/3/library/asyncio-task.html (“Tasks and coroutines”, Python 3.5 documentation)
Object Oriented Programming

Important words:
- Encapsulation
- Inheritance
- Dynamic method binding (polymorphism)
Encapsulation

Data and functions bound together into a single object. “Data hiding” -- hide implementation details from user. (More accurately, control access to data using public and private variables and methods.)
Inheritance

Hierarchy of classes and objects. Shared behaviors and data--code re-use. Static polymorphism: one interface for many kinds of object.
Dynamic Method Binding

An object’s methods are determined at runtime rather than compilation time, since subclasses can override methods and can be used wherever the superclass is allowed. (Example next time.)
Weds., 11 Nov. 2015

Final project discussion

(Description on the website)

Topics in object-oriented languages:
- dynamic binding
- overloading
Last Time:
The three properties of Object Oriented Languages:
Encapsulation (data hiding)
Inheritance
Dynamic binding (polymorphism)
The first two are reasonably clear (“private” vs “public”, subclasses). What about the third?
Dynamic Binding

Consider the following Java program:

class Super {
    public String talk() {
        return "hello";
    }
}

class Sub extends Super {
    public String talk() {
        return "goodbye";
    }
}

class Sub extends Super {
    public static void f(Super a) {
        System.out.println(a.talk());
    }
}

Super x = new Super();
Sub y = new Sub();
f(x); f(y);

What gets printed when we call f(y)?
Dynamic Binding

Answer:
$ javac DynamicDemo.java
$ java DynamicDemo
hello
goodbye

At runtime, Java determined the correct class of the parameter and invoked y’s “talk” method. This is dynamic method binding

See folder nov11 in the shared repo
Overloading

When two methods in a class have the same name but different parameters, we say that the method name is “overloaded.” This is familiar from Java (where, for instance, we have two different “substring” methods for the String class or multiple constructor methods). In C++ we can even overload symbolic operators like “+” and “*” (really, any operator).
Overloading an Operator in C++

```cpp
class Pirate {
    public:
        Pirate(string name) { this->name = name;}
        Pirate operator +(Pirate p) {
            return Pirate(p.getName() + name);
        }
    ...
    ...
    Pirate x("Fred");
    Pirate y("Mary");
    Pirate a = x + y; // Creates a Pirate named "MaryFred"
}
```
See “betterexample.cpp” in the nov11 folder of the shared repo
Overloading an Operator in C++

There are many aspects of operators that we must worry about: precedence, associativity, etc.

C++ avoids these by forcing overloaded operators to have the same precedence and associativity that the original operators had.

(See program “better2.cpp” in the nov11 folder.)
Fri., 13 Nov. 2015

ziuQ (opposite of a quiz!)

Last remarks on object-oriented languages:
One more time: static vs. dynamic binding
Multiple inheritance
Interfaces
Static vs. Dynamic Binding

Let’s revisit this topic (see notes from Nov. 11). What about C++? See sample program “staticbind.cpp” in the nov13 folder of the class repository. There, we see a parent class Super and a child class Sub, each with a get() method.

```cpp
Sub a(10,20);
Super b = a;
cout << a.get() << endl; // Which "get"?
cout << b.get() << endl; // Which "get"?
```

Both will use Super’s “get()” method!
Static vs. Dynamic Binding

By default, C++ uses static binding.

However, you can still obtain the same behavior as dynamic binding by using virtual methods and pointers as shown in program `dynamicbind.cpp` (next slide).
Static vs. Dynamic Binding

class Super {
    public: virtual int get() {return x; }
    ... 
}

class Sub: public Super {
    public: int get() {return y;}
    ...
}

int main() {
    Sub  a(10,20);
    Super *b = &a; // NOTE: pointer variable
    cout << b->get() << endl; // Which "get"?
Multiple Inheritance

In Java, classes and subclasses form a tree—a class may have many subclasses, but each subclass extends exactly one parent class. This is called the “class hierarchy.” Java does not permit “multiple inheritance.”
Multiple Inheritance

On the other hand, the C++ language allows classes to inherit from several different parent classes: *multiple inheritance*. For example, consider the following set of classes:
Multiple Inheritance in C++

class Person { ... };  
class Student : public Person { ... };  
class Employee : public Person { ... };  
class StudentEmployee : public Student, public Employee { ... };  

In C++, constructors for subclasses can invoke the constructors of their parent classes, e.g.,

\[ \text{Student(string name, int year, double gpa)}: \text{Person(name)} \ldots \]

This invokes the constructor of the Parent class and “passes up” the name parameter.
Multiple Inheritance in C++

In our example, `StudentEmployee` can invoke the constructors of both parents:

```cpp
StudentEmployee(string name,... etc...) :
  Employee(name,...),Student(name,...) {

This invokes the constructors of both parent classes and “passes up” the name parameter.
```
Multiple Inheritance in C++

In our example, assume `name` is an instance variable in the class `Person`, with accessor method “`getName()`”. Then both `Student` and `Employee` will inherit this variable as well as the “`getName`” method. Now consider the class `StudentEmployee`. It inherits `name` and `getName` from both `Student` and `Employee`. What happens here?

```cpp
StudentEmployee joe("Joe Jones", ... etc. ...);
cout << joe.getName() << " graduates in " << ... etc. ...
```

```
error: request for member ‘getName’ is ambiguous
error: candidates are: std::string Person::getName()
error: std::string Person::getName()
```
Multiple Inheritance

This is called the “diamond problem” (or, more colorfully, the “Diamond of Death”) after the diamond shape of the class diagram: “Person” above “Employee” and “Student”, both above “StudentEmployee”. In C++, one way to avoid the error on the previous page is to simply choose one of the “getName()” methods and ignore the other one (next slide):
Multiple Inheritance

class StudentEmployee: public Student, public Employee {
...
    string getName() { return Student::getName(); } 
    ...
};

We explicitly name one of the two conflicting “getName” methods in the StudentEmployee class.

See programs “diamond1.cpp” and “diamond2.cpp” in the nov13 directory of the class repository.
Multiple Inheritance

Can we gain the benefits of multiple inheritance in Java?

Sort of ... in Java we can create “interfaces”. They are similar to classes, but an interface has no instance variables and contains only abstract methods. A class can implement more than one interface. It’s not quite the same as multiple inheritance, but yields many of the same benefits.
Mon., 16 Nov. 2015

Questions about lab?
Questions about projects?
Starting chapter 10: Functional programming

Today:
• a bit of history--LISP
• modern functional languages
Some History

LISP -- “LIST Processing”. Invented in 1958 by John McCarthy (the person who coined the term “Artificial Intelligence”; the person who invented “garbage collection” for managing memory)

LISP (nowadays written as “Lisp”) is the second-oldest programming language still in use today (FORTRAN is first).
Lisp Basics

Fundamental structures: atoms and lists.
Atom: indivisible unit (e.g., a number, a character)
List: a structure (like the linked lists you studied in CMPSC 112) with a “head” and a “tail”. Lists are written as parenthesized expressions, e.g., (+ 1 2 3 4 5)
List Notation

$ clisp

[1]> (+ 1 2 3 4 5)
15

[2]> (* 3 5 4)
60

[3]> (sqrt 2)
1.4142135

[4]> (- (+ 2 3) (* 4 8))
-27

The first element of a list is usually considered to be a function; the remaining elements are the arguments.
Quoted Lists

If we want a list to be just an unevaluated list of data, we can “quote it”:

[5]> '(a b c d)
(A B C D)

[6]> '(+ 1 2 3 4 5)
(+ 1 2 3 4 5)

[7]> (quote (1 2 3 4 5))
(1 2 3 4 5)
“first” and “rest” (CAR and CDR)

[11]> (first '(a b c))
A
[12]> (rest '(a b c))
(B C)
[13]> (car '(a b c))
A
[14]> (cdr '(a b c))
(B C)

“CAR” was the assembly language abbreviation for “Contents of the Address Register”; “CDR” was “Contents of the Decrement Register” (from the IBM 704 computer)
The Empty List -- () or NIL

[15]> (first ' '(10))
10

[16]> (rest ' '(10))
NIL

[17]> ()
NIL
Constructing Lists

[18]> (cons '+ '(1 2 3))
(+ 1 2 3)

[19]> (cons '10 NIL)
(10)

[20]> (cons '10 (cons '20 (cons '30 NIL))))
(10 20 30)
Creating Functions

[21]> (defun f (x y) (+ x y))
F
[22]> (f 20 30)
50
[23]> (defun g (list value) (cons value list))
G
[24]> (g '(a b c) '100)
(100 A B C)
Conditionals

\[
\text{(cond}
\begin{align*}
& \text{(condition value)} \\
& \text{(condition value)} \\
& \ldots \\
& \text{(condition value)} \\
& \text{(T value)} \\
\text{)}
\]

(defun mn (a b) (cond
  ((< a b) a)
  (t b)))

MN

(mn 30 40)
30

(mn 40 30)
30
Some More Built-In Functions

**list**: put things into a list

[18]> (list 'a)
(A)

[19]> (list '(a))
((A))

[20]> (list (list (list (list '(10 20 30)))))
((((((10 20 30)))))

[21]> (list 'a 'b 'c)
(A B C)

[22]> (list '(a b) '(10 20 30) '40 'x)
((A B) (10 20 30) 40 X)
Some More Built-In Functions

**length**: return length of a list

[23]>> (length '(a b c d e f g))
7

[24]> (length (list '(a b c d e f g)))
1

**append**: join two lists together

[25]>> (append '(a b c) '(10 20 30))
(A B C 10 20 30)

[26]> (append 'a 'b) ; NOT LISTS--ERROR
*** - APPEND: A is not a list
More User-Defined Functions

rot: rotate a list one place to the left

[33]> (defun rot (lst) (append (rest lst) (list (first lst))))

ROT

[34]> (rot ' (1 2 3 4))

(2 3 4 1)

[35]> (rot ' (a b c d e f g h))

(B C D E F G H A)
Recursion: Good Old Factorial!

[36]> (defun fac (n)
  (cond
    ((<= n 0) 1) ; fac(n) = 1 if n <= 0
    (t (* n (fac (- n 1)))) ; n * (n-1)! otherwise
  )
)
FAC

[37]> (fac 3)
6

[39]> (fac 10)
3628800

[40]> (fac 30)
2652528598121910586363084800000000
Problem (Next Time)

Write a Lisp function named “half” that takes an argument list “lst” and returns a new list consisting of the second half of lst concatenated with the first half, e.g.,

(half `(1 2 3 4 5 6 7 8))

will return the value (5 6 7 8 1 2 3 4).
Weds., 18 Nov. 2015

Questions about projects, labs?

Today: more functional programming
Write a Lisp function named "half" that takes an argument list "lst" and returns a new list consisting of the second half of lst concatenated with the first half, e.g.,

(half '(1 2 3 4 5 6 7 8))

will return the value (5 6 7 8 1 2 3 4).
Puzzle From Last Time

Recall the “rot” function from last time:

(defun rot (lst)
  (append (rest lst) (list (first lst))))

Generalize using recursion:

(defun rotn (lst n) ; rotate left n places
  (cond
    ((<= n 0) lst) ; do nothing if n <= 0
    (t (rot (rotn lst (- n 1))))
  ))
Puzzle From Last Time

(defun half (lst)
  (rotn lst (floor (length lst) 2)))

[49]> (half '(1 2 3 4 5 6 7))
(4 5 6 7 1 2 3)

[50]> (half '(1 2 3 4 5 6 7 8 9 10))
(6 7 8 9 10 1 2 3 4 5)

NOTE: There are more efficient ways; this is just to illustrate function definition
Other Languages: Haskell

We’ve already looked (briefly) at Haskell. Let’s look more closely!

Haskell naming conventions: variables MUST begin with lowercase. Thus, Char, Int, Integer, Bool are types; f, x, y are variables.

NOTE: “variable” is not used in quite the same way as in languages such as C, Java. (Later)
Haskell

Every value in Haskell has a type; we can inspect the type using :type (or just :t). Examples:

```
Prelude> :type False
False :: Bool
Prelude> :t "hello"
"hello" :: [Char]
Prelude> :t (True,"hello")
(True,"hello") :: (Bool, [Char])
```

The symbol “::” means “is of type” or “has type”
Haskell

In Haskell, functions are “first class objects”, so they, too, have types:

```haskell
Prelude> let double x = 2*x
Prelude> double 10
20
Prelude> :t double
  double :: Num a => a -> a
```

The “a -> a” is the type: “function from a to a”. The “Num a =>” is a constraint: a is numeric. NOTE: a is a “type variable.”
Haskell

More examples:

```
Prelude> let g x y = x ++ y
Prelude> :t g
  g :: [a] -> [a] -> [a]
Prelude> let h x y z = [x,y,z]
Prelude> :t h
  h :: t -> t -> t -> [t]
Prelude> :t addlist
  addlist :: Num t => [t] -> [t] -> [t]
```
Haskell

Why all the arrows? Because Haskell allows currying!

Prelude> let addemup p q r = p+q+r
Prelude> :t addemup -- function of 3 arguments
  addemup :: Num a => a -> a -> a -> a
Prelude> :t addemup 10 -- function of 2 arguments
  addemup 10 :: Num a => a -> a -> a
Prelude> :t addemup 10 20 -- function of 1 argument
  addemup 10 20 :: Num a => a -> a
Prelude> :t addemup 10 20 30 -- "constant function"
  addemup 10 20 30 :: Num a => a
Haskell

Defining functions with pattern matching:

Prelude> let {f 10 = "ten"; f 20 = "twenty"; f x = "?"}

Prelude> f 10
"ten"

Prelude> f 20
"twenty"

Prelude> f 30
"?"

Prelude> :t f
f :: (Num a, Eq a) => a -> [Char]

Function f can be applied to any class of numeric type ("Num"), including Int, Integer, Floating, etc.

Class "Eq" is the set of all things that can be compared for equality. According to one source, "Num" is a subclass of "Eq"; according to another, it should be, but isn’t implemented this way. This is an apparent redundancy; I would have expected:

Num a => a -> [Char]
What About Multiline Definitions?

By default, Haskell doesn’t like you to enter multi-line function definitions in ghci mode:

```
Prelude> let {f 10 = "ten";
<interactive>:14:19:    parse error (...
```

But we can enter “multiline mode” with:

```
Prelude> :set +m
Prelude> let {f 10 = "ten";
Prelude> f x = "?”}
Prelude> f 15
"?”
```
Pattern Matching: Example

Prelude> let {g [] = "empty"; g (x:xs) = "nonempty"}
Prelude> g [1,2,3]
"not empty"
Prelude> g "abc"
"not empty"
Prelude> g [x | x <- [2,4,6], x `mod` 2 == 1]
"empty"
Pattern Matching: Example

Prelude> let {  
Prelude|   fib 1 = 1;  
Prelude|   fib 2 = 1;  
Prelude|   fib n = fib (n-1) + fib (n-2)}
Prelude> fib 1  
1
Prelude> fib 2  
1
Prelude> fib 10  
55
Fri., 20 Nov. 2015

Projects

- Remember, progress report due Nov. 30
- Remember, your project requires a formal report in addition to programs

Today: some last words about Haskell (particularly aspects of the language that occur in other functional languages)
More About Haskell -- “let”

What is “let”? It’s a binding/scoping mechanism; its main purpose is to allow us to give names to values and expressions to make it easier to write programs. For instance, here’s a long, messy expression (it was all typed on one line, but appears “wrapped” on this slide):

```haskell
Prelude> [head "abcde","abcde"!!((length "abcde") `div` 2),last "abcde"]
"ace"
```
More About Haskell -- “let”

Prelude> :set +m -- turn on multiline
Prelude> let
Prelude|    s="abcde"
Prelude|    l=length s
Prelude| in [head s,s!!(l `div` 2),last s]
"ace"

The construction “let bindings in expression” means “let the following bindings hold inside this expression”. NOTE: scope is limited!

Prelude> s
<interactive>:9:1: Not in scope: ‘s’
“let” in Other Languages

Most functional programming languages have something similar to the “let” function. In Common Lisp we can write:

\[
\text{F} \quad \begin{array}{l}
(\text{defun} \ f \ (x) \\
(\text{let} \ ((x^2) \ (x^3)) \\
(+ \ x \ x^2 \ x^3))
\end{array}
\]

\[
\text{F} \quad \begin{array}{l}
(\text{f} \ 2) \\
(\text{f} \ 3)
\end{array}
\]

Let \( x^2 \) stand for \( x^2 \), let \( x^3 \) stand for \( x^3 \) in the expression “\( x + x^2 + x^3 \)”. 

14

39
“let” in Other Languages

And of course, we have seen the “let” instruction in JavaScript:

```
"use strict";
function f(x) {
    let max = x * x;
    let sum = 0;
    for (let i = 0; i < max; i++) {
        sum += i;
    }
    return sum;
}
console.log("f(5) = " + f(5));
```

“max” and “sum” are visible only within function `f`; “i” is visible only within the “for” loop.
More About Haskell

Function definition through pattern-matching is used in several functional programming languages (for example, ML, Erlang) and similar syntax is available in a number of other languages as well (F#, Mathematica, and more).
The “map” Function

From the very first functional language, LISP, up to the present, one operation has been nearly universal: the “map” function. Here are some examples in Haskell:

Prelude> let x = [1.0..10.0]
Prelude> map sqrt x
[1.0,1.414,1.732,2.0,2.236,2.449,2.645,2.828,3.0,3.162]
Prelude> map succ x
[2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0]
Prelude> map pred x
[0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0]

Use “map” to apply a function to every element of a list.
The “map” Function

We can map any function to a list as long as the list elements are of the appropriate type for the function:

```
Prelude> let y = ["this","is","a","list","of","strings"]
Prelude> map length y
[4,2,1,4,2,7]
Prelude> map head y
"tialos"
Prelude> let z = [1..10]
Prelude> let f x y = x+y
Prelude> map (f 5) z  -- NOTE: f 5 is a curried function
[6,7,8,9,10,11,12,13,14,15]
```
Here is the “map” function in Python:

```python
>>> from math import *
>>> x = range(1,11)
>>> map(sqrt,x)
[1.0,1.414,1.732,2.0,2.236,2.449,2.645,2.828,3.0,3.162]
>>> y = ["this","is","a","list","of","strings"]
>>> map(len,y)
[4, 2, 1, 4, 2, 7]
>>> map(lambda x:x.find("i"),y)
[2, 0, -1, 1, -1, 3]
```

In the last example we define an “anonymous” function that maps a string “x” into the result of a “find” operation on x. Python has lambda expressions!
The “map” Function

Here is the “map” function in Common Lisp (where it is called “mapcar”):

[16]> (mapcar 'sqrt '(1 2 3 4 5))
(1 1.4142135 1.7320508 2 2.236068)

[17]> (defun f(x) (* 10 (+ x 3)))
F

[18]> (mapcar 'f '(1 2 3 4 5 6))
(40 50 60 70 80 90)
Back to Haskell: “fold” Operations

Another very common operation in functional programming languages is the “fold” or “reduce” operator. It’s like inserting an operator between successive elements in the list. In Haskell, we can write things like:

Prelude> let x = [3,4,7,2,1,8,9,10,6]
Prelude> foldl (*) 1 x
725760
Prelude> foldl (+) 0 x
50

More precisely, “foldl” evaluates left to right: (((3*4)*7)*2)*...
When we use the “foldl” operation we have to specify what to do in the case of an empty list:

- \( \text{foldl} (*) 1 [1,2,3] = 1 \times 1 \times 2 \times 3 \)
- \( \text{foldl} (+) 0 [1,2,3] = 0 + 1 + 2 + 3 \)
Back to Haskell: “fold” Operations

One more example of “foldl”:

```haskell
Prelude> foldl (-) 0 [2,5,3,1]
-11
= (((0 - 2) - 5) - 3) -1
```

For right-associative evaluation, use “foldr”:

```haskell
Prelude> foldr (^) 1 [3,2,2]
81
= 3 ^ (2 ^ (2 ^ 1))
```
Fold Operations in Other Languages

```python
>>> x = [3,5,7,2,1,6,7,2,3]
>>> def f(x,y):
...    return x*y
...    return x*y
...>>> reduce(f,x)
52920
>>> def f(x,y):
...   return x-y
...   return x-y
...>>> reduce(f,x)
-30
```

In Python and some other languages, the “fold” operator is named “reduce.”
Mon., 23 Nov. 2015

Progress reports on projects due next Monday!

Prolog and Logic Programming

First half of class:

http://www.cs.utexas.edu/~cannata/cs345/Class%20Notes/12%20prolog_intro.pdf

(Just do a Google search for “gprolog tutorial”)

Prolog Background

Japan’s “Fifth Generation” project--1982

“These Fifth Generation computers will be built around the concepts of logic programming.”

- The use of logic to express information in a computer.
- The use of logic to present problems to a computer.
- The use of logical inference to solve these problems.
The Fifth Generation Project

“The projectimagined a parallel processing computer running on top of massive databases (as opposed to a traditional filesystem) using a logic programming language to define and access the data.”

Depending on who you ask, the Fifth Generation project was either “Ahead of its time” or a failure.
A “program” consists of a database of facts and a set of rules. Facts are expressed as “predicates”--the programmer supplies the meaning. Examples:

parent(hank,ben). % “hank is a parent of ben”
isa(swan,bird). % “a swan is a bird”
required(cs111). % “cs111 is required”
prereq(cs111,cs112).
eats(unicorn,rose).
stooges(moe,larry,curly).
Prolog

Constants ("atoms" and names of predicates) begin with lowercase letters; variables are capitalized.

Rules specify conditions that must hold for a predicate to be true:

```
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

This means "X is a grandparent of Y if there exists a Z such that X is a parent of Z and Z is a parent of Y." The symbol " :- " should be read as "if" and a comma should be read as "and".
A “program” is more like a database of facts and rules; we solve problems by querying this database. Example:

beats(scissors, paper).
beats(paper, rock).
beats(rock, lizard).
beats(lizard, spock).
beats(spock, scissors).
beats(scissors, lizard).
beats(lizard, paper).
beats(paper, spock).
beats(spock, rock).
beats(rock, scissors).

throws(sheldon, spock).
throws(leonard, lizard).
throws(bernie, paper).
throws(amy, rock).
throws(howard, scissors).

wins(X, Y):-
  throws(X, R), throws(Y, S), beats(R, S).
The last item is a rule:

\[ \text{wins}(X,Y):= \text{throws}(X,R), \text{throws}(Y,S), \text{beats}(R,S). \]

It should be read as:

“\(X\) wins over \(Y\) if, for some values of \(R\) and \(S\), \(X\) throws \(R\), \(Y\) throws \(S\), and \(R\) beats \(S\).”

(For those of you with some mathematics background, we would say \(R\) and \(S\) are “existentially quantified”: “\(X\) wins over \(Y\) if \(there \ exist\) values \(R\) and \(S\) such that…”)

Prolog
Prolog

$ gprolog
| ?- [facts].
(1 ms) yes
| ?- wins(X,Y).
X = sheldon
Y = amy ? ;
X = sheldon
Y = howard ? ;
X = leonard
Y = sheldon ? ;
...

Consult a database named “facts.pl” (ordinary text file in local directory)

Pose a query: “For what values of X and Y does X win over Y?”

System responds with candidate values for variables X and Y

Each time “;” is entered, a new search is made; when no more solutions are found, system says “no”
Prolog

How does it work?
Prolog tries to match the pattern of the query with one of the facts or with the left-hand side of one of the rules. Example: “\texttt{wins(X,Y)}” matches the pattern of the left-hand side of rule:
\begin{verbatim}
\texttt{wins(X,Y) :- throws(X,R),throws(Y,S),beats(R,S).}
\end{verbatim}
If a fact is found, we’re done, otherwise we recursively query each of the terms in the right-hand side of the rule:
“\texttt{throws(X,R)}” and “\texttt{throws(Y,S)}” \textit{BOTH} match the fact “\texttt{throws (sheldon,spock)}”, but there is no match for “\texttt{beats(spock, spock)}”, so we backtrack to find more matches....
... and eventually we find a match-up:

\[
\text{throws}(X,R), \quad \text{throws}(Y,S), \quad \text{beats}(R,S)
\]

- throws(sheldon, spock)
- throws(amy, rock)
- beats(spock, rock)

When a match is made that involves a variable, a BINDING occurs between the variable and the matched item. So, X = sheldon, R = spock, Y = amy, S = rock. Bindings must be consistent.
Prolog

This process of matching patterns in queries to patterns of rules and facts is called “unification.” We say that “throws(X,R)” unifies with “throws(sheldon,spock)”.
Project progress reports due today!

Schedule for rest of semester:

Today--missed topics
Weds., 2 Dec: hand out exam review; begin review
Fri. 4 Dec: first batch of presentations
Mon. 7 Dec: **COOKIES!** 2nd batch of presentations
Thu. 10 Dec, 9 a.m. -- FINAL EXAM
Missed Opportunities

Chapter 12 is about concurrency and programming languages. Issues include:

- how to denote parallelism in a program?
- how to synchronize parallel processes?
- how to share resources (e.g., memory)?

(Some of these are implementation issues rather than language issues.)
Missed Opportunities

Concurrency in Java: threads

```java
class Threads implements Runnable {
    private static int n;
    public void run() {
        System.out.println("Thread "+ n);
    }

    public static void main(String[] args) {
        for (n = 0; n < 5; n++) {
            (new Thread(new Threads())).start();
        }
    }
```
Concurrency in Java: threads

rroos@aldenv111:~$ java Threads
Thread 3
Thread 4
Thread 3
Thread 3
Thread 5

rroos@aldenv111:~$ java Threads
Thread 4
Thread 5
Thread 5
Thread 4
Thread 4
Chapter 13: Scripting Languages

Examples include:

- shell languages (e.g., “bash”, “csh”, “zsh”, “tcsh”, and many others)
- text-processing languages (e.g., “awk”, “perl”, and others)
- “glue” and general-purpose languages (e.g., Python, Perl, Ruby, etc.)
- “extension” languages (e.g., JavaScript, Visual Basic, VimScript, etc.)

Some languages fall under several categories
Scripting Languages

Bash script for renaming a group of files:

```bash
for file in *.html
do
    mv "$file" "${file%.html}.txt"
done

$ ls *.jpeg
pic1.jpeg  pic2.jpeg  pic3.jpeg
$ ./rename
$ ls *.jpeg
ls: cannot access *.jpeg: No such file or directory
rroos@aldenv111:~$ ls pic*jpg
pic1.jpg  pic2.jpg  pic3.jpg
```
Scripting Languages

We have looked at JavaScript in a little more detail than other languages, but mostly we have focused on features of the language itself rather than its use in “extending” the features of HTML, CSS, etc. in web pages. For instance, in Chrome and just about any other browser, search for a menu item called “Developer” or “Tools” or “View Source” and look at the underlying code:
Scripting Languages

Here’s what it looks like on my laptop:
Scripting Languages

```html
<doctype html>
<html>
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js">
  </head>
  <script>
    // Function to generate a random binary-to-decimal conversion problem
    function generate() {
      var i = Math.floor(64*Math.random());
      var result = "";
      var j = i; // we want to preserve i
      while (j != 0) {
        if (j%2==0) result = '0'+result;
        else result = '1'+result;
        j=Math.floor(j/2);
      }
      if (i==0) result = '0'; // special case!
      var ans = {dec:i,bin:result};
      return ans;
    }
  </script>
</html>
```

From: http://cs.allegheny.edu/sites/rroos/cs210f2015/binary.html
Scripting Languages

We can augment the behavior of HTML elements “callbacks”, i.e., functions that get passed into event handlers such as the one that handles a “button click”:

```javascript
// Problem generation:
$("button").click(function(){
    // When user clicks the "click" button, create a random problem:
    ans = generate();

    // Display the problem:
    $("#problem").html("<tt>"+ans.bin+"</tt>");

    // Clear the answer box and evaluation:
    $("#userresponse").val("");
    $("#evaluation").html("");
});
```