CS 320: Principles of Programming Languages

Katie Casamento, Portland State University
Based on slides by Mark P. Jones, Portland State University, Winter 2017

Fall 2018
Week 10: Programming Paradigms

Language = syntax + **semantics**

static semantics: aspects of a program's behavior/meaning that can/must be checked at compile time

dynamic semantics: the behavior of a program at runtime

The rest of the course: studying semantics

- Programming paradigms
  - Imperative, functional, object-oriented, logic, concurrent, …
- Formalizing/analyzing static semantics
  - Fundamentals of static analysis
  - Building blocks for type systems
  - Static vs. dynamic typing
- Formalizing/analyzing dynamic semantics
  - Denotational, operational, and axiomatic semantics
- Proving program correctness
  - Preconditions, invariants, and postconditions

Programming language paradigms

Notions of computation

Programs are descriptions of computations.

- What is a "computation"?
  - Calculating the value of an expression
  - Executing a sequence of commands/instructions
  - Applying logical deduction to reach a conclusion
  - Simulating interacting agents to obtain a result
  - Biological processes or chemical reactions
- Different notions of computing lead to different paradigms
  - Different ways of thinking about computation
  - Different ways of capturing ideas in programs
  - Different tools for solving problems

Multi-paradigm programming

Most major languages these days support programming in multiple different paradigms within the same program/project.

- Each problem can be solved with the most appropriate tools for the job
- Code can be ported more easily between different languages/ecosystems
- Skills are transferable between different languages/ecosystems

Examples:

- C++: procedural, object-oriented (class-based), functional
- Python: imperative, procedural, object-oriented (class-based), functional
- JS: imperative, procedural, object-oriented (prototype-based), functional, reactive

Example: calculating factorial of `n`

imperative

```
int r = 1;
int i = 1;
while (i <= n) {
    r = r*i;
    i = i+1;
}
print r;
```

procedural

```
int fact(int i) {
    if (i<2) {
        return i;
    } else {
        return
            i*fact(i-1);
    }
}
print fact(n);
```

functional

```
print
    (product [1..n])
```

Functional programming

Computing by calculating

An old dictionary definition for "computer":

> "a person who makes calculations, especially with a calculating machine"

This is the basis for functional programming:

- Using a wide range of types of values
  - numbers, strings, tuples, lists, trees, functions, …
- Scaling to large problems

History of functional languages

- Lisp (John McCarthy, 1958)
- Scheme (Guy Steele + Gerald Jay Sussman, 1970)
- ML (Robin Milner + others, 1973)
- Haskell (Haskell Committee, 1990)
- OCaml (INRIA, 1996)
- Scala (Martin Odersky, 2004)
- F# (Microsoft, 2005)
- Clojure (Rich Hickey, 2007)
- and many more (Erlang, Racket, Elm, Idris, …)

Characteristics of functional languages

- Computation is performed by *evaluating* expressions
- Functions are *first class values*:
  - Can be stored in data structures
  - Can be passed into functions as arguments
  - Can be returned from functions as results
- Pure functional languages (e.g. Haskell):
  - No global variables, no assignment, no side effects
  - Functions in the mathematical sense (same arguments → same results)
  - How do we work without side effects?
    - Higher-level coding: `fact n = product [1..n]`
    - Explicit state via recursion: `length (x:xs) = 1 + length xs`
    - Modeling side-effecting computations as ASTs to be executed later

Imperative and procedural programming

Computing by executing

A more recent dictionary definition for "computer":

> "an electronic device for storing and processing data […] according to instructions given to it in a variable program"

A basis for modern computing equipment:

- Programs are described by sequences of primitive machine instructions represented by binary data stored in electronic memory
- A processor executes a program by decoding and acting on the instructions in sequence, one after another

Imperative programming

- Programs specify:
  - The data/state that will be manipulated
  - The instructions that make up the code
- The program starts at the *entry point*
- Programs run by starting at the entry point and running commands, one after another
  - "imperative": "giving an authoritative command"
- Program state changes as the program runs

Program initialization

- The state must be set to some appropriate initial value before the program is executed
- This can be handled by:
    - Calculating initial values for the state and arranging for those values to be loaded in memory with the rest of the program
    - Inserting extra code at the entry point to calculate and save appropriate initial values

Procedural programming

- A derivative form of imperative programming where the code is structured as a collection of procedure/subroutine definitions
- Allows abstraction, reuse, and modular construction of software
- One procedure (usually called `"main"`) is chosen as the entry point
- Programs run by executing the entry point procedure
- Procedures *call* other procedures

History of procedural languages

- Fortran (John Backus, 1957)
- COBOL (Howard Bromberg + Howard Discount + Vernon Reeves + Jean E. Sammet + William Selden + Gertrude Tierney, 1959)
- BASIC (John Kemeny + Thomas Kurtz, 1964)
- Pascal (Niklaus Wirth, 1970)
- C (Dennis Ritchie, 1972)
- Ada (Jean Ichbiah, 1983)
- …

Procedure libraries

- A *library* in a procedural language is a collection of procedures, designed to provide services to some other program (a *client*)
- Supports abstraction, reuse, modular construction
- Library code is executed only when invoked by the client
- The procedures that are provided to clients define an *interface* to the library
    - often called an *application programming interface (API)*
- Some procedures in library code may be for internal use only, excluded from the interface

Public and private state

It's common to partition the state of procedural code:

- *public* state is directly accessible to clients
- *private* state is only accessible to code within the library that manages it

Limiting access to private state helps to isolate bugs and promote modularity:

- If the private state is invalid/inconsistent, this must be result of a bug in the library itself (not the client)

Encapsulated state

The goal is for state to be *encapsulated*: clients shouldn't have access to private state or internal procedures.

Multiple options:

- Rely on programmer discipline (Python)
- Enforce encapsulation with static analysis (Java, C++, …)

Object-oriented programming

Object-oriented programming

- Computations are performed by collections of *objects* that *invoke methods* or *send messages* to one another
- Objects contain data (*fields*) and procedures (*methods*)
- Objects can share methods through *inheritance*
  - Class-based: C++, Java, C#, Python, PHP, Objective-C, …
  - Prototype-based: JS, Lua, Self, …
- Objects are initialized with *constructor* procedures
- Objects can *override* inherited methods (*dynamic dispatch*)

Prototype-based OOP

Prototype-based OOP

- Originated in Self (David Ungar and Randall Smith, 1987)
- Most common in dynamically typed scripting languages
- Relevant today because of Javascript/ECMAScript and Lua

Objects

In a prototype-based OOP language, objects are essentially collections of fields.

```
var student = {
    name: "Stu Dent",
    gpa: 1.7,
    scholarship: true,
    sayHello: function() { print("hello"); }
}

student.gpa         // returns 1.7
student.sayHello() // prints "hello"
```

Prototypes

An object has a special field called the *prototype*, which can point to another object.

```
var Person = {
    sayGoodbye: function() { print("goodbye"); }
}
student.prototype = Person;
```

Field access on an object tries to resolve a field first on the object itself, then on the object's prototype, then on the object's prototype's prototype, etc.

- `student.sayHello()   // field of student`
- `student.sayGoodbye() // field of student.prototype`

Constructors

The `new` operator uses a function as a *constructor*, to create an object with a pre-defined prototype.

```
function Person(name) {
    this.name = name;
    this.sayName = function() { print(this.name); }
}

person = new Person("Per Son");

student.name       // returns "Per Son"
student.sayName() // prints "Per Son"
```

Constructors

Specifically, `new Foo` (for some function `Foo`) does three things:

1. Create an object whose prototype is `Foo.prototype`
2. Call `Foo` with the given arguments, binding `this` to point to the object created in step 1
3. Return the object

Constructors

```
    function Person(name) {
        this.name = name;
        this.sayName = function() { print(this.name); }
    }
```

`var person = new Person("Per Son")` is equivalent to:

```
    var obj = { prototype: Person.prototype };
    obj.name = "Per Son";
    obj.sayName = function() { print(obj.name); }
    var person = obj;
```

Simulating classes

Some prototype-based languages (e.g. ES6) have syntactic sugar for simulating a *class* using prototypes.

```
class Person {
    constructor(name) { this.name = name; }
    sayName() { print(this.name); }
}
```

This code is exactly equivalent to the code on the previous slide!

Static methods

Methods declared as *static* are fields on the constructor object itself, not the objects created with it.

```
class Person {
    constructor(name) { this.name = name; }
    sayName() { print(this.name); }
    static sayHello() { print("hello"); }
}

person = new Person("Per Son");
person.sayName();  // valid
person.sayHello(); // not valid
Person.sayHello(); // valid
```

Class-based OOP

Class-based OOP

- Originated in Simula (Ole-Johan Dahl and Kristen Nygaard, 1965)
- Popularized by Smalltalk (Alan Kay, Dan Ingalls, Adele Goldberg, 1972)
- Comes in statically-typed and dynamically-typed variants
- Most common form of OOP (Java, C#, Python, Ruby, …)

Classes

A *class* specifies a collection of fields and methods that will be present in every object that is an *instance* of the class.

```
class Person {
    String name;
    Date dob;
    long ssn;
}
```

- Different fields can have different types
- Fields are accessed by name

```
    Person user = new Person(...); print("hello" + user.name);
```

Object initialization

To create a new object, we need to

- allocate storage for an instance of some class
- initialize the fields of that new object

Some OOP languages use special syntax for constructors to make allocations explicit:

```
    Person user = new Person(...);
```

The "this" pointer

Code within the definition of a class can access the fields of the class by using a special value, usually called "this" or "self":

```
class Person {
    String name;
    Date dob;
    long ssn;

    Person(String name, Date dob, long ssn) {
        this.name = name;
        this.dob = dob;
        this.ssn = ssn;
    }
}
```

Access modifiers

Fields and methods can be declared with *access modifiers*, which specify where they're accessible from; different languages support different sets of access modifiers.

- *public* fields are accessible in any code in the project that uses the class
- *private* fields are only accessible in the definition of the class
- *protected* (C++, Java, C#) fields are only accessible in the definition of the class and its subclasses
- *internal* (C#) or *package private* (Java) fields are only accessible in the build target (assembly/package) that contains the class
- Modifiers can sometimes be combined (e.g. *protected internal* in C#)

Static fields

A field declared with the *static* modifier has a single instance that "belongs" to the class.

```
class Counter {
    static int count = 0;
    int num;

    Counter() {
        // every "new Counter()" call creates an object
        // with a different value in this.num
        this.num = count++;
    }
}
```

Method calls

A method call has three syntactic components:

```
    receiver.method(arguments)
```

The *receiver* object becomes the value of `this` during execution of the method body.

Static methods

Methods declared static don't have access to a receiver object; they can only access fields marked static.

```
class Counter {
    static int count = 0;
    int num;

    Counter() {
        this.num = count++;
    }

    static int numAllocated() { return count; }
}
```

Inheritance and dynamic dispatch

Class-based inheritance

- Every class can *extend* another *superclass*, making it a *subclass*
- The subclass *inherits* all of the fields and methods of its superclass, and can add new fields and methods

```
class Super {
    int x;
    int getX() { return x; }
}

class Sub extends Sup {
    int y;
    int getY() { return y; }
}
```

Subclass constructors

The constructors of a subclass have access to the constructors of its superclass.

```
class Super {
    private int x; // inaccessible in subclass method definitions
    Sup(int x) { this.x = x; }
}

class Sub extends Sup {
    int y;
    Sub(int x, int y) {
        super(x); // use constructor for Super to initialize x
        this.y = y;
    }
}
```

Subtyping

If we have `class Sub extends class Super`, we can use a value of type `Sub` anywhere that a value of type `Super` is expected.

- An object of type `Sub` has all the fields and methods of `Super`
- Sometimes called the *Liskov substitution principle*
- We sometimes use the syntax `C <: D` to mean "C is a subclass of D"

```
Super p = new Super(1);  // okay: Super = Super
Super q = new Sub(2, 3); // okay: Sub <: Super
Sub r   = new Super(4);  // not okay: Super ≮: Sub
Sub s   = new Sub(5, 6); // okay: Sub = Sub
```

Method overriding

A subclass can *override* the implementation of any method of its superclass (except those declared private).

```
class Super {
    int x;
    int getX() { return x; } // new Super(2).getX() == 2
}

class Sub extends Sup {
    Sub(int x) { super(x); }
    @Override int getX() { return x*x; } // new Sub(2).getX() == 4
}
```

Abstract classes

An *abstract* class can be used as the superclass of another class, but can't be instantiated itself.

```
abstract class Super { // new Super(1) is a static error
    private int x;
    Super(int x) { this.x = x; }
}

class Sub extends Super { // new Sub(1, 2) is valid
    int y;
    Super(int x, int y) { super(x); this.y = y; }
}
```

Abstract methods

The definition of an abstract class may include *abstract methods*, which specify the *signature* of a method without a method body.

```
abstract class Prop {
    abstract bool eval(Env e);
    String hello() { print("hello"); }
}
```

- Every *concrete* (non-abstract) class must provide implementations of all abstract methods of its superclass
- An *interface* is an abstract class with only abstract methods

Abstract class/method example

```
abstract class List {
    abstract int length();
}

class NilList extends List {
    NilList() { }
    int length() { return 0; }
}

class ConsList extends List {
    private int head; private List tail;
    ConsList(int head, List tail) { … }
    int length() { return 1 + tail.length(); }
}
```

Double dispatch

Suppose we want to add a method for testing whether two Lists are equal.

```
abstract class List {
    abstract bool equals(List that);
}

class NilList extends List { …
    bool equals(List that) { return that.length() == 0; }
}

class ConsList extends List { …
    bool equals(List that) { ??? }
}
```

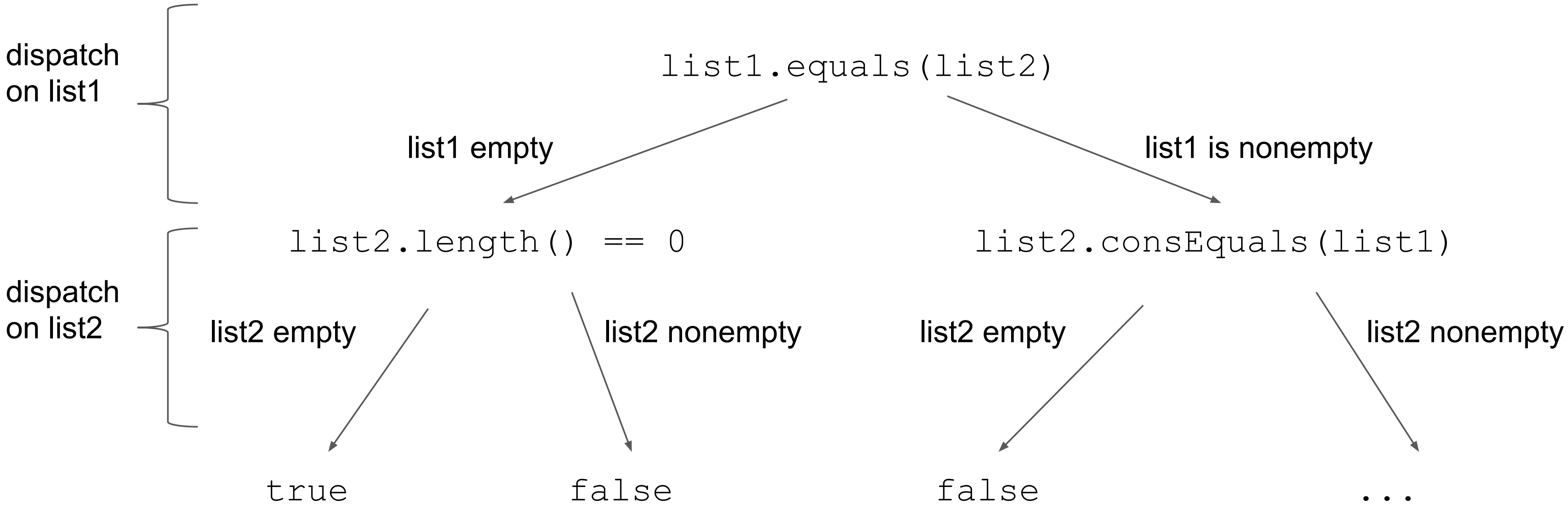
Double dispatch

```
abstract class List { …
    abstract bool equalsCons(ConsList that);
}

class NilList extends List { …
    bool equalsCons(ConsList that) { return false; }
}

class ConsList extends List { …
    bool equalsCons(ConsList that) {
        return this.x == that.x && this.tail.equals(that.tail);
    }
}
```

Double dispatch

```
class ConsList extends List { …
    bool equals(List that) {
        return that.equalsCons(this);
    }
}
```

Why "double" dispatch?



Comparison with Haskell

```
data List = Nil | Cons Int List

length Nil = 0
length (Cons head tail) = 1 + length tail

equals Nil that = length that == 0
equals (Cons head tail) that = case that of
    Nil -> False
    Cons head' tail' -> head == head' && equals tail tail'
```

OOP vs. FP

- The Haskell List type corresponds to an abstract class
- The Haskell constructors correspond to subclasses
- The Haskell functions correspond to methods
- Pattern matching corresponds to double dispatch
- The OOP implementation is more verbose, but essentially the same code

Common code, OOP style

	Nil	Cons
`length`	`0`	`1 + tail.length()`
`equals`	`that.length() == 0`	`that.consEquals(this)`

Decomposition with classes

	`Nil`	`Cons`
`length`	`0`	`1 + tail.length()`
`equals`	`that.length() == 0`	`that.consEquals(this)`

Adding a new class

	`Nil`	`Cons`	`Append`
`length`	`0`	`1 + tail.length()`	`...`
`equals`	`that.length() == 0`	`that.consEquals(this)`	`...`

Adding a new class is (relatively) easy

```
class AppendList extends List {
    private List left, right;
    Append(List left, List right) { … }

    int length() {
        return left.length() + right.length();
    }

    int equals(List that) {
        ... // may require some extra methods
    }
}
```

Adding a new method

	`Nil`	`Cons`	`Append`
`length`	`0`	`1 + tail.length()`	`...`
`equals`	`that.length() == 0`	`that.consEquals(this)`	`...`
`sum`	`...`	`...`	`...`

Requires a new change in every class!

Common code, FP style

	`Nil`	`Cons`
`length`	`0`	`1 + length tail`
`equals`	`length that == 0`	`case that of ...`

Decomposition with functions

	Nil	Cons
`length`	`0`	`1 + length tail`
`equals`	`length that == 0`	`that.consEquals(this)`

Adding a new function

	Nil	Cons
length	0	1 + length tail
equals	length that == 0	case that of ...
sum	0	head + sum tail

Adding a new function is (relatively) easy!

```
sum Nil = 0
sum (Cons head tail) = head + sum tail
```

Adding a new constructor

	`Nil`	`Cons`	`Append`
`length`	`0`	`1 + length tail`	`...`
`equals`	`length that == 0`	`case that of ...`	`...`
`sum`	`0`	`head + sum tail`	`...`

Requires changes to every function!

The "expression problem"

Coined by Philip Wadler:

- "The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g. no casts)."

Various solutions have been proposed (e.g. multi-paradigm languages), but they typically require awkward boilerplate and/or large complex sets of language features.

Programming language design is not done yet!

Summary

- Languages can be classified according to paradigms
  - Functional, procedural, object-oriented, …
  - Paradigms are fuzzy and often somewhat subjective
  - Still, can be useful for evaluating a new language against existing languages
- Functional programming emphasizes the evaluation of expressions and the use of functions as first-class values
- Procedural programming views programs as collections of procedures with associated state
- Object-oriented programming views programs as collections of stateful objects that interact by sending messages
- Many modern languages are multi-paradigm