CS 314 Principles of Programming Languages

Guang Wang
Department of Computer Science
Rutgers University

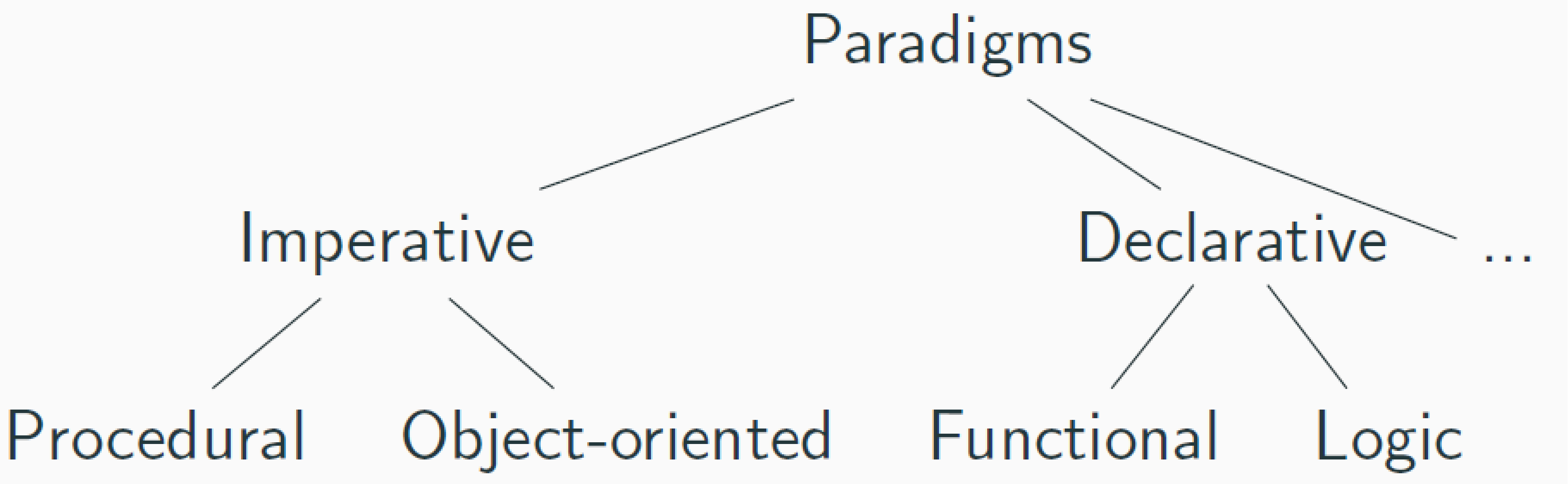
Programming languages

	Static	Dynamic
Imperative	C, Java	Python
Functional	Haskell	Scheme

Programming paradigms

- Imperative
  - A program is a sequence of actions that modify state.
  - Matches the von Neumann architecture / Turing machines.
- Functional
  - Composition of functions operating on a set of data.
  - Based on lambda calculus.
- Logic
  - Logical specification of a problem.
  - Programs declare the form of the solution, not how to find it.
- Object oriented
  - Objects hold state and have methods that can mutate state.
  - Objects communicate by passing messages or calling methods.
  - Somewhat orthogonal to other paradigms.

Programming paradigms



- Imperative, procedural: C, Pascal
- Imperative, object-oriented: C++, Java, C#, Python
- Functional: Haskell, OCaml, F#, Scheme
- Logic: Prolog

Imperative programming

- What is imperative programming?
  - Program = series of statements that change state
  - Assignment used to change values stored in memory

Closely matches execution of underlying hardware.

- Common features in imperative languages:
  - Procedures
  - Loops
  - Blocks
  - Conditional branches
  - Unconditional branches

Dynamic typing

- Types prevent some operations:
- But variables can be reassigned to refer to different types:

Lambda calculus

- A mathematical model of computation, based on mathematical functions.
- Consider the following:
  - variables
  - functions (abstraction)
  - functions applied to some argument (application)

We usually write

$$f(x) = x + 5$$

Lambda calculus functions aren't named, so it's more like

$$x \mapsto x + 5$$

- variables
  - $x, y, z, \ldots$
- abstraction
  - $\lambda x.x$
- application
  - $(\lambda x.x)y$

Lambda Calculus

$$(\lambda x.x)y$$

- A function with parameter x and body x.
- Replace every occurance of the parameter in the body with the actual argument (y).
- Replace every x in x with y.
- y

Lambda Calculus

- Note that function arguments can be other functions!

$$(\lambda x.x)(\lambda x.y)$$

What does $(\lambda x.x)(\lambda x.y)$ mean?

- $\lambda x.x$ is a function with parameter x and body x.
- Replace every occurance of the parameter in the body with the actual argument $(\lambda x.y)$.
- Replace every x in x with $\lambda x.y$.
- $\lambda x.y$

Lambda Calculus

Note that function bodies can be other functions!

$$(\lambda x.(\lambda y.x))wv$$

What does $(\lambda x.(\lambda y.x))wv$ mean?

- $\lambda x.(\lambda y.x)$ is a function with parameter x and body $\lambda y.x$.
- Replace every occurance of the parameter in the body with the actual argument (w, not wv!).
- Replace every x in $\lambda y.x$ with w.
- $\lambda y.w$
- But we still have v, so we can apply this: $(\lambda y.w)v$
- w

Lambda Calculus

And both function bodies and arguments can be other functions!

$$(\lambda x.(\lambda y.x))(\lambda z.w)$$

What does $(\lambda x.(\lambda y.x))(\lambda z.w)$ mean?

- $\lambda x.(\lambda y.x)$ is a function with parameter x and body $\lambda y.x$.
- Replace every occurance of the parameter in the body with the actual argument $(\lambda z.w)$.
- Replace every x in $\lambda y.x$ with $\lambda z.w$.
- $\lambda y.(\lambda z.w)$

Lambda calculus

Our substitution rule for function application is formally called β-reduction.

α-equivalence

These are the same:

- $\lambda x.x$
- $\lambda y.y$
- $\lambda z.z$

α-conversion

We can rename xs in $\lambda x.M$ with y, as long as y is not already a free variable in the body:

$\lambda x.M \equiv \lambda y.M[x := y]$, where $y \notin FV\ M$

- $\lambda x.xx = \lambda y.yy$
- $\lambda x.xy \neq \lambda y.yy$

Free and bound variables

Free and bound variables

In the expression $\lambda x.xy$, we say the x in the body is *bound* (by the enclosing λ), but y is free.

Variable capture

This is called variable capture: a variable that was free becomes bound.

$$(\lambda x.\lambda y.xy)yz \Rightarrow (\lambda y.yy)z$$

- The x in $\lambda y.xy$ is free (although bound in $\lambda x.\lambda y.xy$)
- But both ys in $\lambda y.yy$ are bound.

Normal form

- We say a lambda term is in normal form when it can't be reduced any further.

Functional programming

Fundamental concept: application of (mathematical) functions to values

- ❑ Referential transparency: The value of a function application is independent of the context in which it occurs
- value of f (a, b, c) depends only on the values of f , a, b and c
- It does not depend on the global state of computation
- all vars in function must be local (or parameters)

Pure Functional Languages

- no explicit assignment statements
- no iteration
- recursion is widely used
- all storage management is implicit
  - needs garbage collection
- Functions are First Class Values
  - Can be returned as the value of an expression
  - Can be passed as an argument
  - Can be put in a data structure as a value
  - (Unnamed) functions exist as value
- A program includes:
  - A set of function definitions
  - An expression to be evaluated

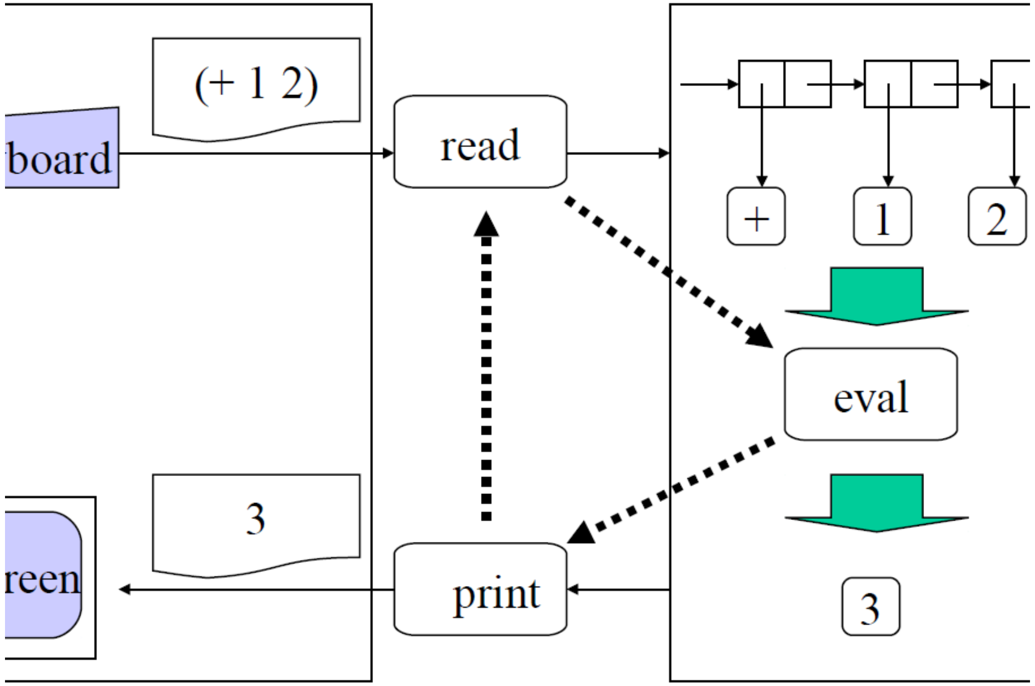
Scheme

Scheme is an *interactive* language

- **"Read-eval-print loop"**

Scheme is a *functional* language

- **A program is an expression to be evaluated**
- **Functions are data like any other data**



Expressions

A program is an expression to be evaluated

An expression is:

- **– A literal constant: 3, 3.1416, “hello”**
- **– A variable that has been bound to some value: x, ?a, +**
- **– A function application: (+ x 1)**
- **– A special form: (lambda (x) (+ x 1))**

A function application is written as a list: (+ 3 5)

- **– Evaluate first element of this list → function to apply: addition**
- **– Evaluate rest of elements of the list → arguments to apply the function to: 3 and 5**

Equality Checking

- The eq? predicate doesn't work for lists.

- For lists, need a comparison function to check for the same structure in two lists

```
(define equal?
  (lambda (x y)
    (or (and (atom? x) (atom? y) (eq? x y))
        (and (not (atom? x)) (not (atom? y))
             (equal? (car x) (car y))
             (equal? (cdr x) (cdr y))))))
```

Higher-order Functions: map

```
(define map
  (lambda (f l)
    (if (null? l)
    '()
    (cons (f (car l)) (map f (cdr l))))))
```

- map takes two arguments: a function and a list
- map builds a new list by applying the function to every element of the (old) list

(map abs '(-1 2 -3 4))   (1 2 3 4)
(map (lambda (x) (+ 1 x)) '(-1 2 -3))   (0 3 -2)

Higher Order Functions: reduce

- reduce: a higher order function that takes a binary, associative operation and uses it to "roll-up" a list

- Example:
- (reduce + '(10 20 30) 0)
- (+ 10 (reduce + '(20 30) 0))
- (+ 10 (+ 20 (reduce + '(30) 0)))
- (+ 10 (+ 20 (+ 30 (reduce + '() 0))))
- (+ 10 (+ 20 (+ 30 0)))
- 60

Higher Order Functions: sum

- Now we can compose higher order functions to form compact powerful functions.

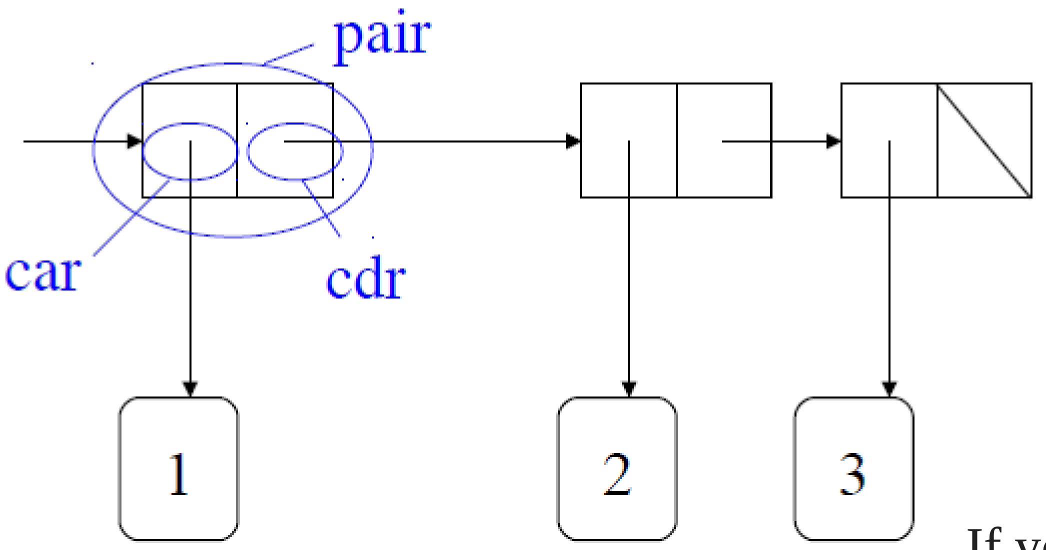
- (sum (lambda (x) (* 2 x)) '(1 2 3))
- (reduce (lambda (x y) (+ 1 y)) '(a b c) 0)

lambda

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- It has various uses in particular fields of programming besides other types of expressions in functions.

Scheme: Lists

Elements, separated by whitespace, surrounded by ( )



(car '(1 2 3)) => 1

(cdr '(1 2 3)) => (2 3)

(cons '1 '(2 3)) => (1 2 3)

(car ‘((a) b (c d))) => (a)
(car (car ‘((a) b (c d)))) => a
(car (car (car ‘((a) b (c d))))) => *error*

car will return the first *element* of a list
cdr will return the *list* without the first element
cons means "construct." Cons takes two arguments and returns a *list* constructed from those two arguments (combination).

If you take the cdr of a list, and then the cdr of that, and so on, and eventually reach the empty list ( ), the list you started with is called a “proper list”. If stop at an item (other than ( )) which does not have a cdr, e.g., a symbol or a number, it is called an improper list. Improper lists are rarely used, and we do not cover them in 314.

Haskell

- **Functional**
  - Functions are values
  - Focus on evaluating expressions rather than executing instructions
- Pure
  - Expressions are referentially transparent:

    (1) No mutation

    (2) No side effects

    (3) Same function + same arguments = same value
- Lazy
- Statically typed

Haskell

- **Pure**
  - Expressions are referentially transparent:

  (1) No mutation

  (2) No side effects

  (3) Same function + same arguments = same value

  - This allows for:

  (1) Equational reasoning replacing equals by equals

  (2) Parallelism expressions don’t affect each other

  (3) Easier debugging?

Haskell

- **Laziness**
- Expressions aren't evaluated until their results are needed
  - Easy to dene new syntax
  - Infinite data structures
  - Easy to compose functions together
- But it complicates understand the time/space usage of your code.

Haskell

Statically typed

- Every expression has a type, checked at compile-time.
  - Type inference
  - Helps with design
  - Helps with debugging
  - Makes code easier to read and understand

Don't repeat yourself

- Haskell is very good at abstraction.
  - Algebraic data types
  - Polymorphism
  - Type classes
  - Monoids, functors, monads, ...

Wholemeal programming

Same idea in Haskell:

```haskell
sum (map (3*) lst)
```

In Scheme:

```scheme
(reduce + (map (lambda (x) (* 3 x)) lst) 0)
```

```java
int acc = 0;
for (int i = 0; i < lst.length; i++) {
    acc = acc + 3 * lst[i];
}
```

Variables

```
-- this is a comment

{- this is also
   a comment -}

x :: Int -- x has type Int
x = 3
```

= is like mathematical equality,
not assignment!

Variables are immutable. This is illegal:

```
x = 3
x = 4
```

Types

- Int (42)
- Integer (12345678909872184652998347212983498723 4)
- Float (3.14)
- Double (3.14)
- Bool (True, False)
- Char ('a', 'b') – Unicode
- String – a list of Chars

Arithmetic

```
ex01 = 3 + 2
ex02 = 19 - 27
ex03 = 2.35 * 8.6
ex04 = 8.7 / 3.1
ex05 = mod 19 3
ex06 = 19 `mod` 3 -- backticks make mod infix
ex07 = 7 ^ 222
ex08 = (-3) * (-7) -- negative numbers should be written with parentheses
```

Arithmetic

Haskell doesn't do implicit type conversion. This doesn't work:

```
x :: Int
x = 3

y :: Integer
y = 4

z = x + y
```

Arithmetic

Use `fromIntegral` to convert from Int or Integer to another numeric type:

```
x :: Int
x = 3

y :: Integer
y = 4

z = (fromIntegral x) + y
```

Arithmetic

- To convert floating-point to an integer type:
  - round
  - floor
  - ceiling

Arithmetic

Division does floating-point division and the operands must be floating-point values.

For integer division, use `div`:

```
ex09 = i1 `div` i2
ex10 = 12 `div` 5
```

Boolean logic

```
ex11 = True && False
ex12 = not (False || True)
```

Equality

Compare for equality with == and /=, or the usual ordering relations <, <= >, >=.

```
ex13 = ('a' == 'a')
ex14 = (16 /= 3)
ex15 = (5 > 3) && ('p' <= 'q')
ex16 = "Haskell" > "C++"
```

Thanks!!!