Computational Structures in Data Science

UC Berkeley EECS
Lecturer
Michael Ball

Lambda

(Computing) In the News

- Ruth Bader Ginsburg died on Friday, and it sucks. It's OK to acknowledge the pain, and frustration.
- Burglars beware: tech pioneers aim to make South Africa's townships safer
- Autonomous Robots Are Coming To The Operating Room
- Amazon wins FAA approval for Prime Air drone delivery fleet

Computational Structures in Data Science

UC Berkeley EECS
Lecturer
Michael Ball

Lambda

Learning Objectives

- Lambda are anonymous functions, which use expressions
  - We don’t use return, they always return the value.
  - They are typically short and concise
  - They don’t have an “intrinsic” name when using an environment diagram.

lambda

- Function expression
  - “anonymous” function creation
  - Expression, not a statement, no return or any other statement

lambda <arg or arg_tuple> : <expression using args>

```
inc = lambda v : v + 1
```

```
def inc(v):
    return v + 1
```

Lambdas

```
>>> def inc_maker(i):
...     return lambda x:x+i
...
>>> inc_maker(3)
<function inc_maker.<locals>.<lambda> at 0x10073c510>

>>> inc_maker(3)(4)
7
>>> map(lambda x:x*x, [1,2,3,4])
<map object at 0x1020950b8>

>>> list(map(lambda x:x*x, [1,2,3,4]))
[1, 4, 9, 16]
>>>
```

Lambda with HOFs

- **A function that returns (makes) a function**

```
def leq_maker(c):
    return lambda val: val <= c
```

```
>>> leq_maker(3)
<function leq_maker.<locals>.<lambda> at 0x1019d8c80>

>>> leq_maker(3)(4)
False

>>> filter(leq_maker(3), [0,1,2,3,4,5,6,7])
[0, 1, 2, 3]
```

UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Types and Universality

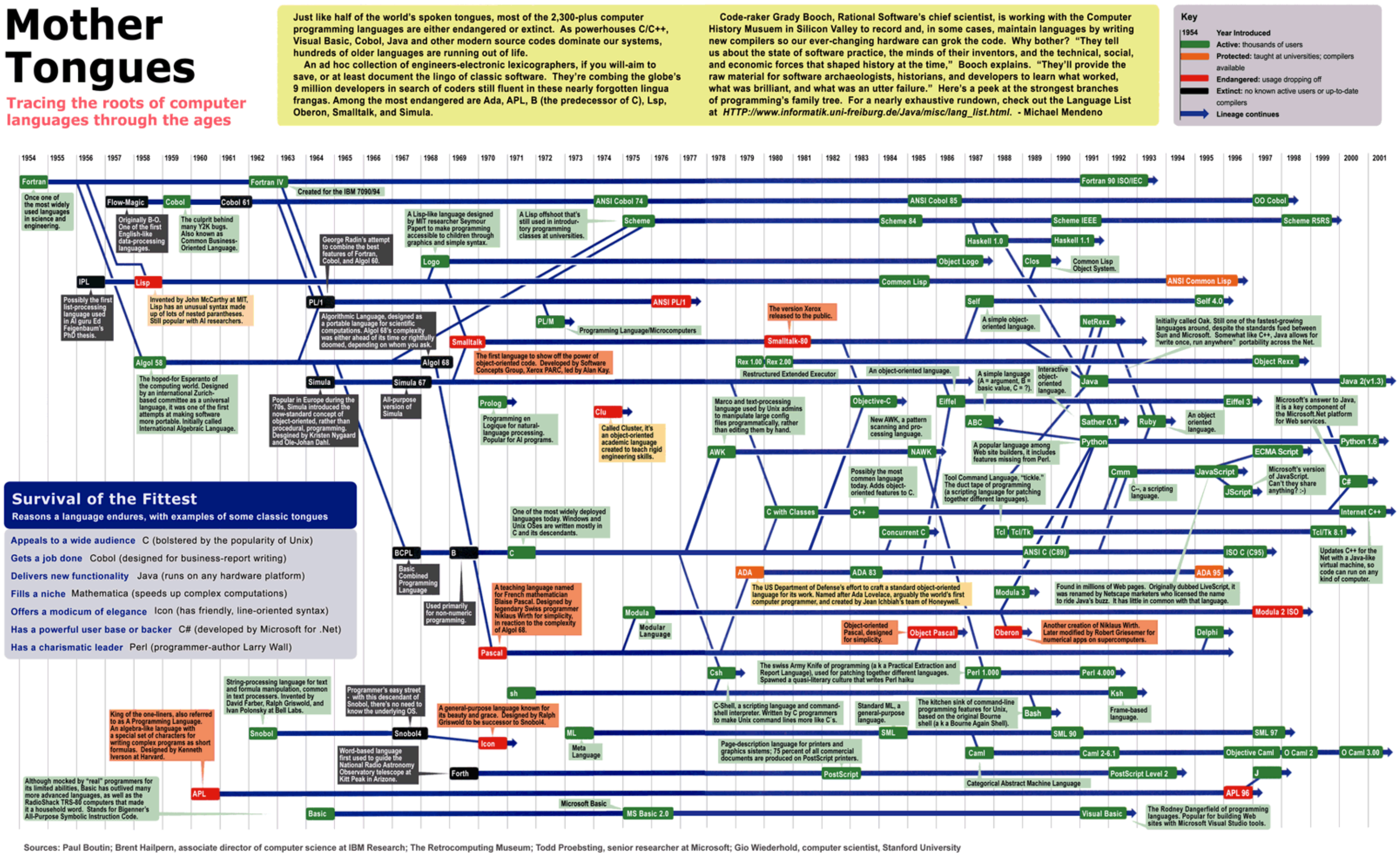
Learning Objectives

- Abstract Data Type:
  - Applying a name to a new concept.
  - Give us structure for organizing how we build and name functions
  - Write a simple ADT around a "point".

Universality

- Everything that can be computed, can be computed with what you know now.
- “Turning Completeness”
- Poorly or Well

Evolution of Programming Languages



Examples Data Types You have seen

- Lists
  - Constructors:
    - list(...)
    - [<exps>,...]
    - [<exp> for <var> in <list> [if <exp>]]
  - Selectors: <list> [<index or slice>]
  - Operations: in, not in, +, *, len, min, max
    - Mutable ones too (but not yet)
- Tuples. [We have only briefly used these.]
  - Constructors:
    - (<exps>,...)
  - Selectors: <tuple> [<index or slice>]
  - Operations: in, not in, +, *, len, min, max

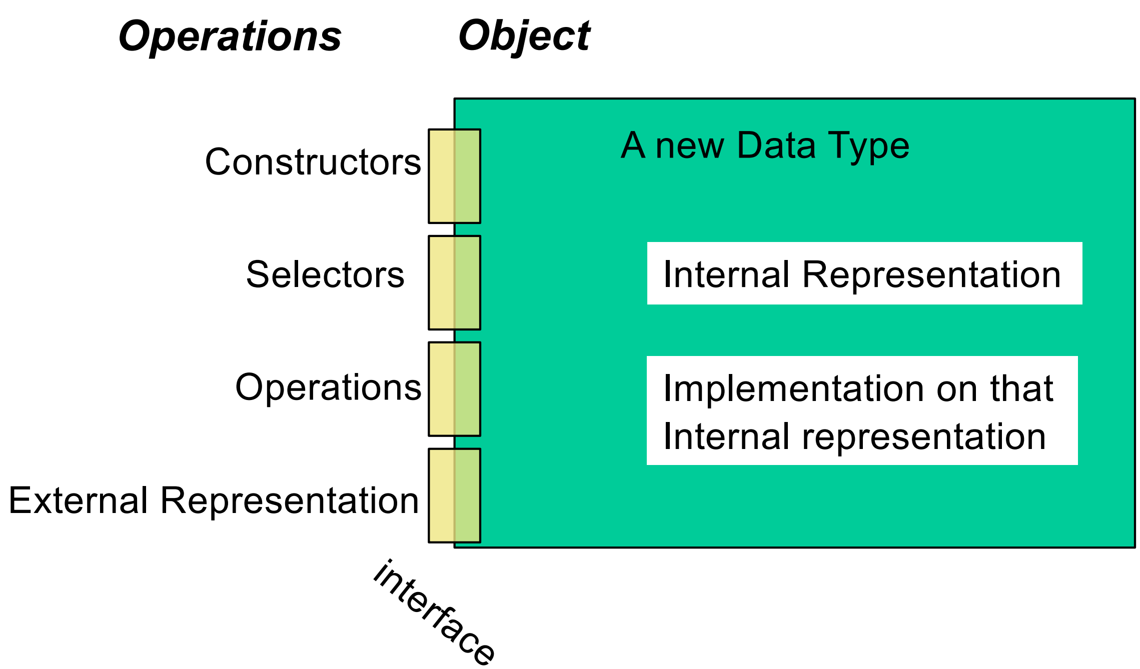
More “Built-in” Examples

- Strings
  - Constructors:
    - str(…)
    - “<chars>”, ‘<chars>’
  - Selectors: <str> [<index or slice>]
  - Operations: in, not in, +, *, len, min, max
- Range
  - Constructors:
    - range(<end>), range(<start>,<end>), range(<start>,<end>,<step>)
  - Selectors: <range> [<index or slice>]
  - Operations: in, not in, len, min, max

Abstract Data Type

- We have “built-in” types, but we can use functions to represent some similar kind of data.
- Today we’re going to look at a “point”
- Functions are lightweight. Later we will build more complex structures.

Abstract Data Type



Example: A "Point"

- A point (in Geometry) is typically an x-y pair. (Though, we could define another interface...)

- Constructor: point()
- Selectors: x(), y()
- Operations: distance()
- Implementation: Could be anything! This uses a tuple.

- Python Tutor

Bonus / Review

- Left over slides we didn't get to.

MAP

```
list(map(function_to_apply, list_of_inputs))
```

Transform each of items by a function.

e.g. square()

Inputs (Domain):

- Function
- Sequence

Output (Range):

- A sequence

```
def map(function, sequence):
    return [ function(item) for item in sequence ]

list(map(square, range(10)))
```

FILTER

```
list(filter(function, list_of_inputs))
```

Keeps each of item where the function is true.

Inputs (Domain):

- Function
- Sequence

Output (Range):

- A sequence

```
def filter(function, sequence):
    return [ item for item in sequence if function(item) ]
```

REDUCE

```
reduce(function, list_of_inputs)
```

Successively **combine** items of our sequence

- function: add(), takes 2 inputs gives us 1 value.

Inputs (Domain):

- Function, with 2 inputs
- Sequence

Output (Range):

- An item, the type is the output of our function.

```
def reduce(function, sequence):
    result = function(sequence[0], sequence[1])
    for index in range(2, len(sequence)):
        result = function(result, sequence[index])
    return result
```

Three super important HOFS

* For the builtin filter/map, you need to then call list on it to get a list.
If we define our own, we do not need to call list

```
list(map(function_to_apply, list_of_inputs))
```

Applies function to each element of the list

```
list(filter(condition, list_of_inputs))
```

Returns a list of elements for which the condition is true

```
reduce(function, list_of_inputs)
```

Applies the function, combining items of the list into a "single" value.