Functional Programming

bend your mind

Imperative

- Imperative programming is about
  - modifying mutable variables
  - using assignments
  - using control structures such as if-then-else, loops, break, continue, return.
- The most common informal way to understand imperative programs is as instruction sequences for a Von Neumann computer.

This maps well to computers

- Mutable variables ≈ memory cells
- Variable dereferences ≈ load instructions
- Variable assignments ≈ store instructions
- Control structures ≈ jumps
- Problem? Scaling up

Von Neuman bottleneck

- In the end, pure imperative programming is limited by the "Von Neumann" bottleneck:
  One tends to conceptualize data structures word-by-word.

- We need other techniques for defining high-level abstractions such as collections, polynomials, geometric shapes, strings, documents. Ideally: Develop theories of collections, shapes, strings, …

Theories?

- A theory consists of
  - one or more data types
  - operations on these types
  - laws that describe the relationships between values and operations
- Normally, a theory does not describe mutations!

functional

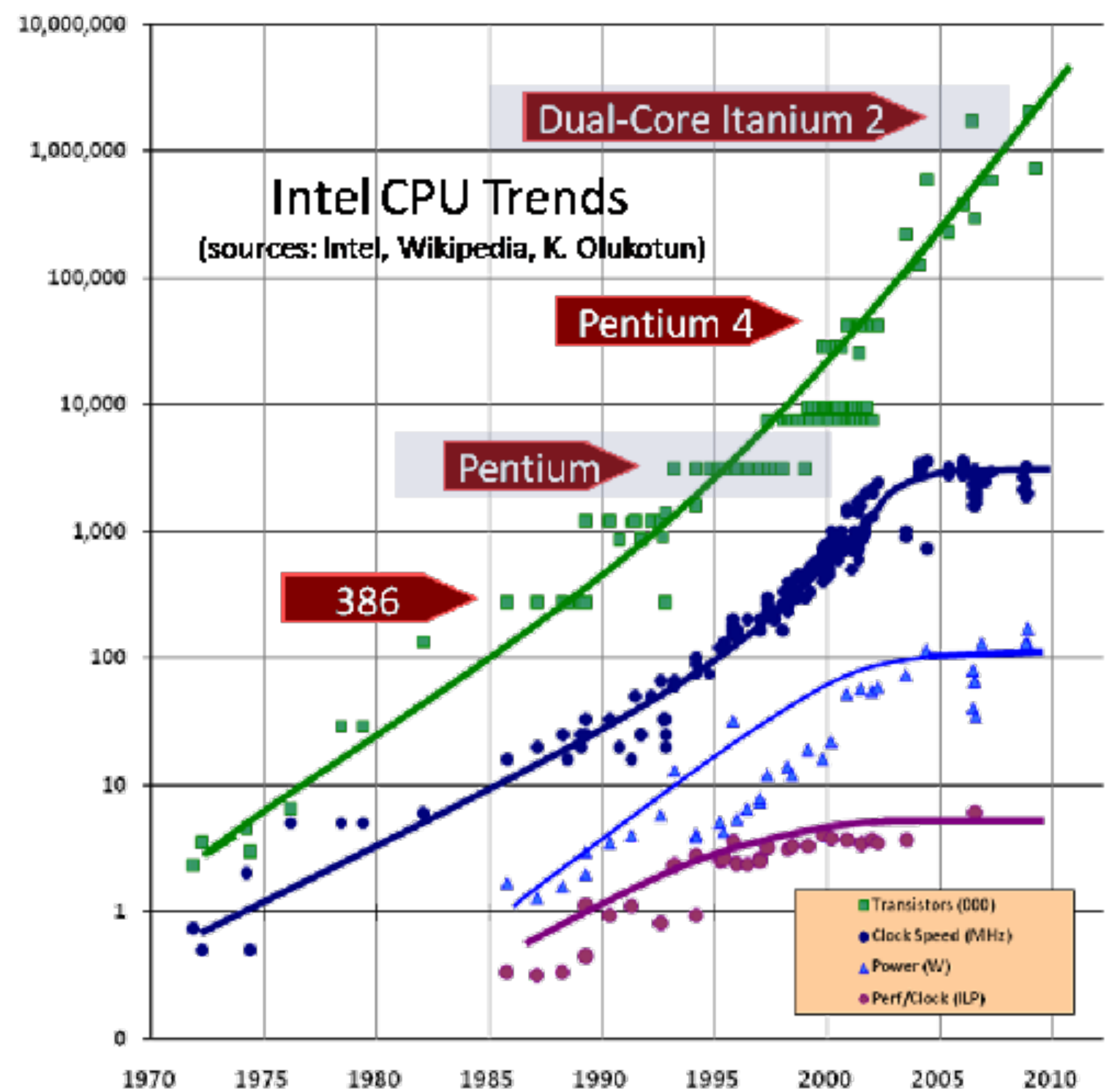
- In a restricted sense, a functional programming language is one which does not have mutable variables, assignments, or imperative control structures.
- In a wider sense, a functional programming language enables the construction of elegant programs that focus on functions.
- In particular, functions in a FP language are first-class citizens.
  - they can be defined anywhere, including inside other functions
  - like any other value, they can be passed as parameters to functions and returned as results
  - as for other values, there exists a set operators to compose functions

Functional paradigm

- The two **defining features** of the functional paradigm are that
  - all computations are treated as the evaluation of a function
  - you avoid changing state and mutable data.
- But there are some additional **common features** of functional programming:
  - First-class functions: functions can serve as arguments and results of functions.
  - Recursion as the primary tool for iteration.
  - Heavy use of pattern matching.
  - Lazy evaluation, which makes possible the creation of infinite sequences.

parallel

Functional Programming is becoming increasingly popular because it offers an attractive method for exploiting parallelism for multicore and cloud computing.



Why is FP good for parallel?

- FP’s recognized good fit for concurrency appeals to people writing multi-processor apps, high-availability apps, web servers for the social network, and more.
- FP’s higher-level abstractions appeal to those looking for faster development time or more understandable code.
- FP’s emphasis on immutability has a strong appeal for anyone concerned about reliability.

Language history

- 1959 Lisp 1975-77 ML, FP, Scheme
- 1978 Smalltalk 1986 Standard ML
- 1990 Haskell, Erlang 1999 XSLT
- 2000 OCaml
- 2003 Scala, XQuery
- 2005 F#
- 2007 Clojure

Pure vs Impure

At its core, functional programming is about programming with pure, side-effect-free functions.

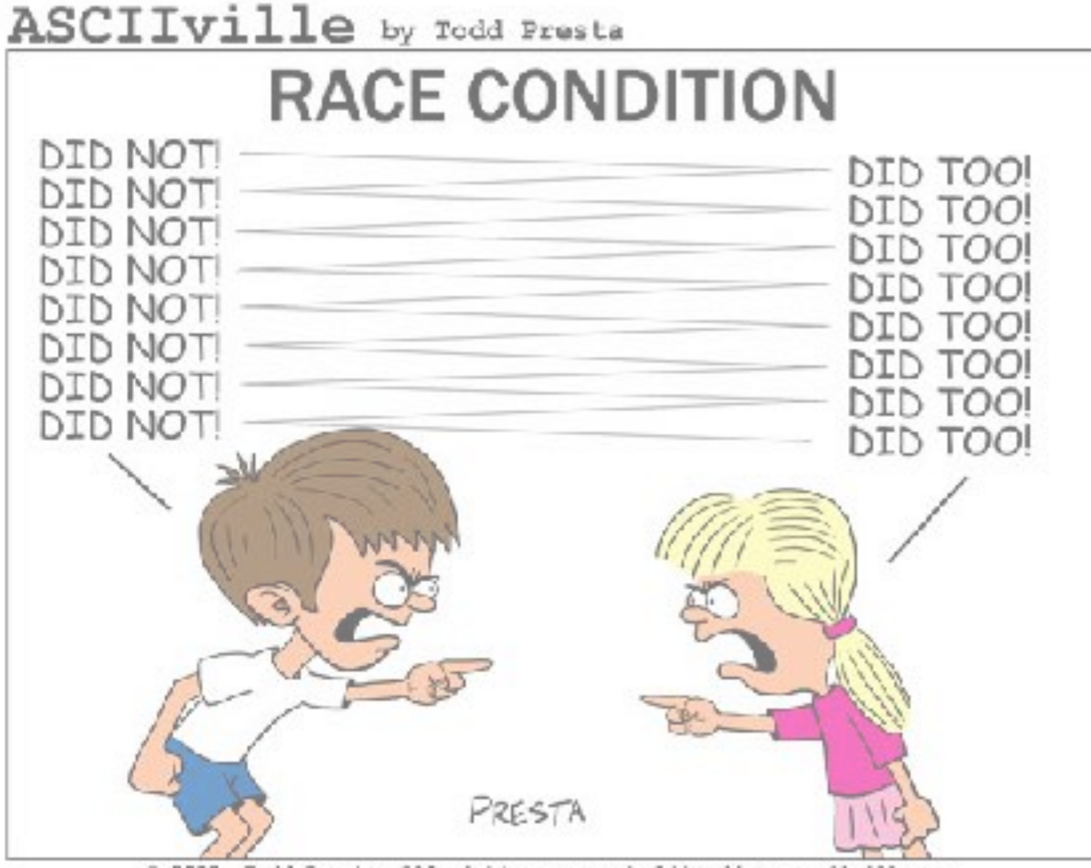
- a function that doesn’t rely on mutating state is called a **pure function**
- a function,that is mutating state, that may be shared across many pieces of a program is called and **impure function**

```
//Pure
public int incrementCounter(int counter) {
  return counter++;
}

//Impure
private int counter = 0;
public void incrementMutableCounter() {
  counter++;
}
```

Threads ?
Race Conditions?

think and dicuss



A simple program

```
public List<Integer> filterOdds(List<Integer> list) {
  List<Integer> filteredList = new ArrayList<Integer>();


  for(Integer current : list) {
    if(1 == current % 2) {
      filteredList.add(current);
    }
  }
  return filteredList;
}
```

A bit of refactoring

```
public List<Integer> filterOdds(List<Integer> list) {

  List<Integer> filteredList = new ArrayList<Integer>();

  for (Integer current : list) {

    if (isOdd(current)) {

      filteredList.add(current);
    }

  }

  return filteredList;

}

private boolean isOdd(Integer integer) {

  return 1 == integer % 2;

}
```

Requirements can change

But … what if we need evens?

```
  for (Integer current : list) {
    if (isEven(current)) {
      filteredList.add(current);
    }
  }
  return filteredList;
}

private boolean isEven(Integer integer) {

  return 0 == integer % 2;

}
```

DRY Principle

Don’t Repeat Yourself

Let’s isolate the piece, that can change

```
public interface Predicate {

  public boolean evaluate(Integer argument);

}
```

We use higher order functions

```
public List<Integer> filter(List<Integer> list, Predicate predicate)
{
  List<Integer> filteredList = new ArrayList<Integer>();
  for (Integer current : list) {
    if (predicate.evaluate(current)) {
      filteredList.add(current);
    }
  }
  return filteredList;
}

class isEven implements Predicate {
  public boolean evaluate(Integer argument) {
    return 0 == argument % 2;
  }
}

class isOdd implements Predicate {
  public boolean evaluate(Integer argument) {
    return 1 == argument % 2;
  }

}
```

Demo

Rewrite from imperative style

Lab

functional_prog_intro_03