Turing-Completeness

• Lambda calculus is Turing-complete
• C and OCaml can express all the same computations
• And so can a lot of other things:
Turing-Completeness: Representation

• Functional, imperative, etc. languages can all simulate each other

• We can define any language in OCaml (syntax, type system, semantics, interpreter)
  — Or in C, or in Java, or…
Choosing a Language

• We don’t have to choose a language based on what’s possible to compute

• Reasons for choosing a language:
  — Familiarity
  — Existing/legacy code
  — Library support
  — Tool support
  — Match between syntax/paradigm and thing we’re trying to describe
  — Level of abstraction
Turing-Incomplete Languages

• We don’t have to choose language based on what’s possible to compute
• And our languages don’t have to be able to compute everything!
• There are some useful Turing-incomplete domain-specific languages (DSLs):
  — Regular expressions
  — HTML
  — Some versions of SQL
  — Interactive theorem provers
Combining Language Features

• Lambda-expressions in Java, Python, ...

\[ f = \text{lambda } x : x + 1 \]  
(* f evaluates to a closure *)

print \( f(5) \)

• References in OCaml

Already exist, but require major changes to the semantics!
Combining Language Features

• Object orientation + universal polymorphism, as in F#

• There are now two “top” types: Object and ‘a
  — F# generic functions are defined with universal types
  — Other .NET methods (ToString, etc.) are defined on Object
  — Type inference may infer Object or ‘a as the type of an argument depending on how it’s used, unpredictably
  — Reflection makes this even worse: “generic” functions can case on the type of the input, turning polymorphism *ad-hoc*
Combining Language Features

• Pattern matching in imperative languages
  ― Usually not done, because they don’t have inductive datatypes

• Pattern matching in Rust:

```rust
class Option<T> {
    enum Option {
        Some(T),
        None,
    }

    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}
Making a New Language

• Why make a new language?
  — To change the syntax
  — To make a design decision in the other direction
  — To make a common pattern easier to write
  — To combine features that can’t easily be added on to an existing language
  — To make a language that can’t express certain bad programs
Making a New Language: Swift

• Goal: a cleaner replacement for Apple’s Objective-C
• Object-oriented, with Java-like syntax
• Can add an interface implementation to an existing class
• Functions (closures) as values
• Option type for possibly-null values
• Type inference
• Has libraries for iOS programming
Making a New Language: Kotlin

• Goal: a faster, more concise replacement for Java
• Object-oriented, with Java-like syntax
• Both member methods and top-level functions
• Can extend classes with new methods
• Functions (closures) as values, including higher-order functions
• Type inference
• Can deconstruct an object into the tuple of its fields
• Has libraries for Android programming
Making a New Language: Rust

• Goal: a safer replacement for C
• C-like syntax, but actually more like a functional language
• Inductive data types and pattern matching
• Type inference
• Type classes for ad-hoc polymorphism
• “Borrowing” type system for safe concurrency
• Used by Mozilla to implement Firefox
Making a New Language

• Swift: meant to replace Objective-C, cleaner and easier to use, combines OO and functional features, type system prevents null pointer references, pushed by Apple

• Kotlin: meant to replace Java, cleaner and easier to use, combines OO and functional features, type system prevents null pointer references, pushed by Google

• Rust: meant to replace C, safer, combines OO and functional features, type system prevents data races, pushed by Mozilla