Core Design Concepts Discussed:

Conceptual Models | Syntax | Analysis | Evaluation

Names, Functions, and Types

Harley Eades III

Programming Language Varieties

- Imperative Programming: State-based instructions
- Declarative Programming: Describe what a computation should perform.
- Functional Programming: Function-based programs.
- Object-Oriented Programming: Programs are organized into classes and objects.

The Core Design Concepts

- Conceptual Model allowing humans to reason and construct programs.
- Syntax for expressing computation.
- Analysis for discovering bugs in syntactically-valid programs.
- Evaluator for running syntactically-valid programs.

A Spectrum of Change

While we have this notion of core-design concepts, programming languages change over time.

Just consider languages supporting functional programming.

Functional Programming Adoption Timeline

- 1930s: The lambda-calculus discovered by Alonzo Church.
- 1930s: Proved Turing Complete by Alan Turing in this paper introducing Turing Machines.
- 1950s: First high-level programming language called LISP developed by John McCarthy.
- 1960s: First abstract machines was developed.
- 1970s: ML was created by Robin Milner.
- 1980s: Miranda the first lazy language as developed by David Turner.
- 1980s: Haskell and its open standard for functional languages began.
- 1990s: Haskell implementations take off starting in 1992.
- 1990s: Standard ML is defined and implemented.
- 1990s: OCaml begins at Inria.
- 2000s: Functional programming enters the mainstream:
  - 2005-2007: C# 2.0 and 3.0
  - 2009: PHP 5.3 and 5.4
  - 2009: Python
  - 2011: C++11
  - 2014: Java 8
  - 2014: Apple's Swift
  - 2015: Javascript (ES6)

Building on top of the Core

Programming languages begin with a:

- core conceptual model and a
- core set of features.

Building on top of the Core

Then more features are added on top of this core design striving to make the language:

- More usable.
- More powerful.
- More correct.

Building on top of the Core

As an example, consider Javascript:

- Core design: object-oriented first.
- Conceptual model: Everything is an object.
    - Even functions are objects with fields and methods.
    - 🤯

Core Design Concepts:

Syntax

Names

Let Bindings: Variables in OCaml

REPL variable declaration:

```
# let x = e;;
```

Names

Let Bindings: Variables in OCaml

$$\overset{\text{name}}{\downarrow}$$

```
# let x = e;;
```

Names

Let Bindings: Variables in OCaml

```
         name  expression
           ↓       ↓
# let x = e;;
```

Core Design Concepts:

Analysis

Names

```
# let x = e;;
```

- `x` is an immutable variable.
- `x` is an alias for `e`.
- everywhere `x` is used it will be literally replaced with `e` at run time.
- immutability makes it easier to reason about the correctness of our programs, and results in more correct programs, because we do not need to manage any type of reference or state.

Core Design Concepts:

Evaluation

Names

Let Bindings: Variables in OCaml

```
# let x = 42;;
```

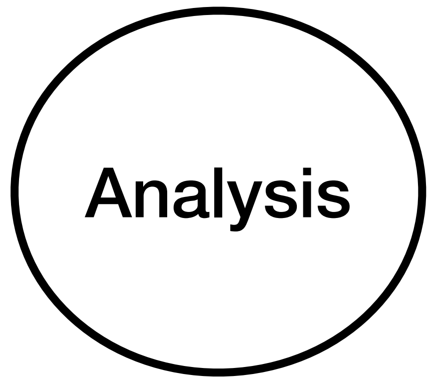
Names

Let Bindings: Variables in OCaml

```
# let x = 42;;
val x : int = 42
#
```

$\uparrow$ type of x

Core Design Concepts:



Names

Let Bindings: Variables in OCaml

```
# let x = 42;;

val x : int = 42
```

<u>Analysis:</u> At compile time, before a program is evaluated, it is <u>type checked</u>.

This process attempts to examine the structure of the program and make sure that the program can be given a type that makes sense.

Core Design Concepts:

Analysis

Type Systems

OCaml is an example of a static type system:

"use the structure of the program to verify its type at compile time"

much like C#, Java, Swift, and Typescript.

This is different from dynamic typing:

"verify the type of a program at run time"

which is used by languages like Python.

Technically, C# is a gradual type system which combines both static and dynamic typing.

More on type systems in a later lecture.

Core Design Concepts:

Evaluation

Names

Let Bindings: Variables in OCaml

```
# let x = 42;;
val x : int = 42
# (x + 2) * 3;;
```

At runtime, `x` will be replaced with `e`.

This process is known as substitution.

More on substitution in a later lecture.

Names

Let Bindings: Variables in OCaml

```
# let x = 42;;
val x : int = 42
# (x + 2) * 3;;
- : int = 132
```

Core Design Concepts:

Syntax

Functions

Let Bindings: Basic Functions in OCaml

```
# let f x = e;;
```

Functions

Let Bindings: Basic Functions in OCaml

$$\overset{\text{function name}}{\downarrow}$$

```
# let f x = e;;
```

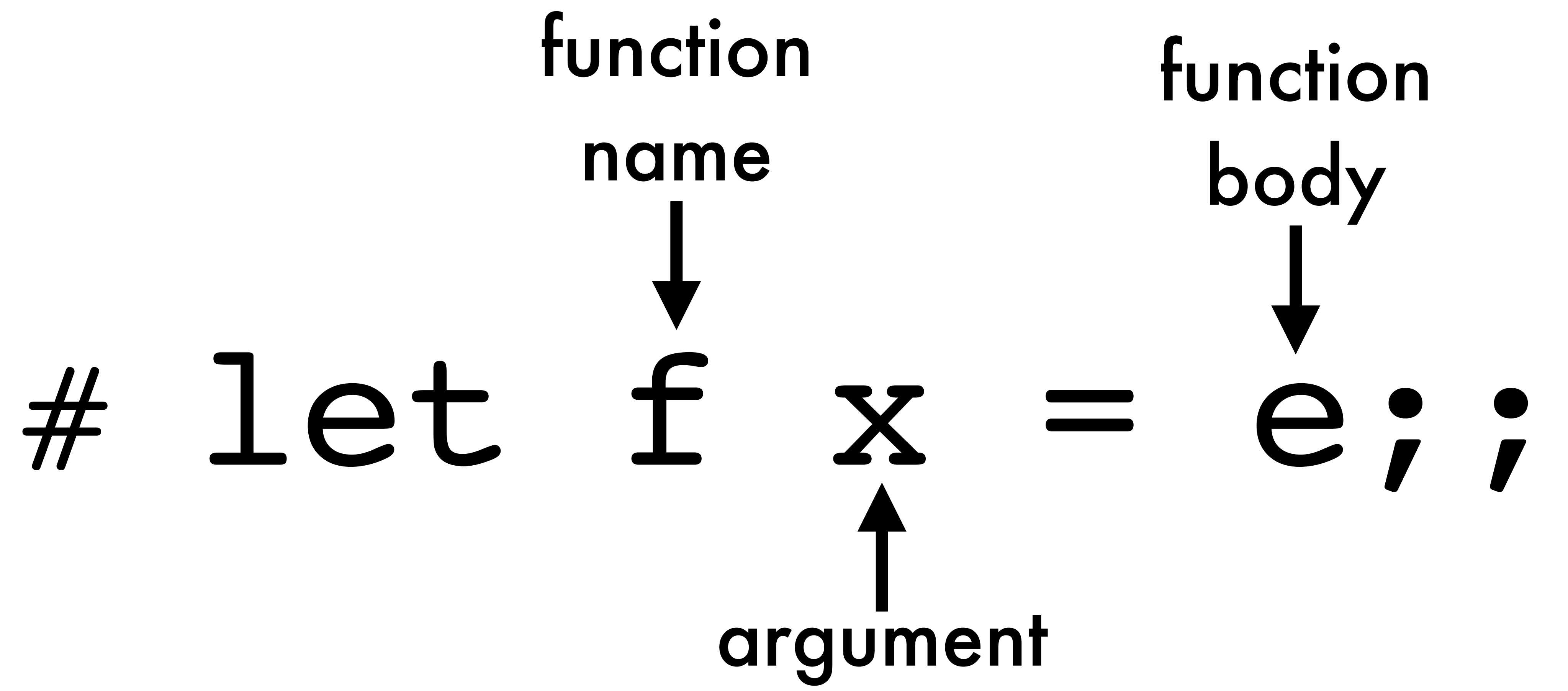
Functions

Let Bindings: Basic Functions in OCaml

$$\texttt{\# let }\overset{\substack{\text{function}\\\text{name}\\\downarrow}}{\texttt{f}}\;\underset{\substack{\uparrow\\\text{argument}}}{\texttt{x}}\;\texttt{= e;;}$$

Functions

Let Bindings: Basic Functions in OCaml



```
# let f x = e;;
```

Core Design Concepts:

Evaluation

Functions

Let Bindings: Basic Functions in OCaml

```
# let dt x = (x + 2) * 3;;
```

Core Design Concepts:

Evaluation

Functions

Let Bindings: Basic Functions in OCaml

```
# let dt x = (x + 2) * 3;;
val dt : int -> int = <fun>
```

function type

Core Design Concepts:

Syntax

Function Types

$$\underset{\substack{\uparrow \\ \text{argument} \\ \text{type}}}{\texttt{a}} \quad \texttt{->} \quad \underset{\substack{\uparrow \\ \text{body} \\ \text{type}}}{\texttt{b}}$$

- Arguments of function types are separated by an arrow.
- The last type, on the right, is the type of the body of the function.

Core Design Concepts:

Syntax

Function Types

$$\underset{\substack{\uparrow \\ \text{1st} \\ \text{argument} \\ \text{type}}}{\texttt{a}} \;\; \texttt{->} \;\; \underset{\substack{\uparrow \\ \text{2nd} \\ \text{argument} \\ \text{type}}}{\texttt{b}} \;\; \texttt{->} \;\; \underset{\substack{\uparrow \\ \text{body} \\ \text{type}}}{\texttt{c}}$$

- Arguments of function types are separated by an arrow.
- The last type, on the right, is the type of the body of the function.

Core Design Concepts:

Conceptual Model

Function Types

```
  a -> (b -> c)
= a -> b -> c
≠ (a -> b) -> c
```

The arrow type is right associative.

More on the last type in a future lecture.

Core Design Concepts:

Syntax

Function Types

```
a -> b -> c
```

- C#:
  ```
  Func<a,Func<b,c>>
  ```
- Swift:
  ```
  (name1: a) -> (name2: b) -> c
  ```
- Typescript (Javascript):
  ```
  (name1: a) => (name2: b) => c
  ```

Core Design Concepts:

Evaluation

Functions

Let Bindings: Basic Functions in OCaml

```
# let dt x = (x + 2) * 3;;
val dt : int -> int = <fun>
# dt 42;;
```

Core Design Concepts:

Evaluation

Functions

Let Bindings: Basic Functions in OCaml

```
# let dt x = (x + 2) * 3;;
val dt : int -> int = <fun>
# dt 42;;
```

function application

Core Design Concepts:

Syntax

Functions Application

Arguments to a function are separated by spaces.

$$\underset{\text{function}}{\underset{\uparrow}{\texttt{f}}}\ \texttt{a}\ \texttt{b}\ \underset{\text{arguments}}{\underset{\uparrow}{\texttt{...}}}\ \texttt{c}$$

Core Design Concepts:

Syntax

Functions Application

Arguments to a function are separated by spaces.

```
f a b ... c
```

Other styles of syntax:

- C-style:

```
f(a, b, ..., c)
```

- Named Parameters (Swift):

```
f(name1: a, name2: b, ..., namei: c)
```

Core Design Concepts:

Conceptual Model

Functions Application

Application is left associative:

```
((f a) b) ...) c

= f a b ... c
```

Function application will be discussed more in a later lecture.

Core Design Concepts:

Evaluation

Functions

Let Bindings: Basic Functions in OCaml

```
# let dt x = (x + 2) * 3;;
val dt : int -> int = <fun>
# dt 42;;
- : int = 132
```