

Language Design & Implementation

Extra Material

What are some of the design decisions?

- Domain-specific or general-purpose?
- Functional or imperative/procedural or logic-deductive or mixed-paradigm?
- Turing-complete or not?
- Declarative or imperative?
- Dataflow / stream processing?
- C-derived syntax, or something else?
 - Infix, prefix, postfix?
 - Text or graphical?
 - Simple grammar vs complex grammar.
- Type system model?
 - Dynamically-typed or statically-typed?
 - Manifest typing or type inference?
 - Weakly-typed or strongly-typed?
 - Algebraic types? E.g., sum of products?
- Modularisation? Namespaces?
- Some sort of object-orientation? Or not?
 - Inheritance? Encapsulation? Polymorphism? Templates? Interfaces? Substitutability?
- Functions/procedures/operators/predicates?
- Constraints?
- Lambdas, closures, continuations?
- First-class... Functions, threads, control structures, variables, whatever?
- Variables: Mutable or immutable?
- Built-in arrays vs built-in containers vs library containers?
- Built-in I/O vs I/O via library operators
- Operator overloading? Operator overriding?
- Automatic garbage collection or explicit memory management?
- Lazy evaluation (and memoization?) vs eager evaluation?
- Pointers vs References vs Value Semantics
 - Function pointers?
- Pass by reference/value/name?
- Comments that do special things?
 - See literate Haskell.
 - Annotations? See Java.
- Esoteric or conventional?
- Interpreter or compiler or transpiler?

Understand the Distinction

- Models of Computation
 - Aka Computational Paradigms
- Language Paradigms

Models of Computation

- Turing Machines
- Lambda Calculus
- Cellular Automata
- Combinatory logic
- Etc.

Language Paradigm(s)

- Some say there are three and only three:
 - <http://wiki.c2.com/?ThereAreExactlyThreeParadigms>
- Some say there are three known, and one yet to be discovered (or at least implemented):
 - <http://wiki.c2.com/?ThereAreExactlyThreeParadigms>
- Some say there are many:
 - https://en.wikipedia.org/wiki/Programming_paradigm
- Some say there are none:
 - <http://wiki.c2.com/?ThereAreNoParadigms>

Understand the Distinction

Procedural vs Functional

A language isn't "functional" just because it has a construct called 'function'.

C is not a functional programming language, though it might be used to build functional programming languages.


JavaScript is *closer* to being a functional programming language than C.

It's helpful to understand why. (Homework!)

Understand the Distinction

Syntax vs Semantics

- Language = syntax + semantics



What you can say.

`a = 3;`



What you say *does*. (Literally, what it *means*.)

“The value denoted by evaluating the literal expression ‘3’ is assigned to the variable identified by the name ‘a’.”

Understand the Distinction

Imperative vs Declarative

- Imperative
 1. Do this first.
 2. Then do that.
 3. Do the other thing last.
- Declarative
 - There are three things.

Understand the Distinction

Languages vs IDEs or Editors

- Visual Studio
- Emacs
- Eclipse
- Netbeans
- IntelliJ
- Notepad++

Editors/IDEs (generally) aren't languages.

They're usually not interesting in language terms, but interesting Editor/IDE features might be language dependent.

Test: Can the language exist and be used without these?

There are visual programming languages *entirely* dependent on a graphical environment.

Despite the name, Visual Studio isn't one of them.

Understand the Distinction

Languages vs Standard Libraries

Is a language standard library...

- E.g. the C standard library (libc), C++ STL, C++ Standard Library, .NET Framework, Java Platform, etc.

...part of the language?

Understand the Distinctions

Dynamically typed vs Statically typed

Manifest vs Inferred

Nominal vs Structural

Duck-typing vs non- Duck-typing

First... What are types?

Second... What is type safety?

Understand the Distinctions

Dynamically typed vs Statically typed

Manifest vs Inferred

Nominal vs Structural

Duck-typing vs non- Duck-typing

- For many Languages, this is true:

- **Type** = a (relatively) invariant set of values and zero or more associated operators.
- **Operator** = a function or procedure that given one or more values, returns one or more values.
- **Value** = an immutable instance of a given type, i.e., one of a set of values.
- **Constant** = an invariant reference to a value.
- **Variable/parameter** = a possibly time-varying reference to a value.

Understand the Distinction

Dynamically typed vs Statically typed

Handy rule of thumb:

- Statically-typed languages tend to associate types with variables and values (expressions). A value (expression) can only be assigned to a variable if their types are *type compatible*, and this can often be determined (at least to some degree) at compile-time.
- Dynamically-typed languages tend to associated types only with values (expressions). A value (expression) can be assigned to a variable unconditionally.

Understand the Distinction

Dynamically typed vs Statically typed

Language Category S

Every variable is directly associated with, or has, a type. In other words, every variable has a "type" property.

The type of a variable can be determined by explicit declaration of its type (e.g., in C, C++, C#, Java): `int x`; Or by type inference (e.g., in C#): `var x = 3`;

Every value is associated with, or has, a type. E.g, integer, float, string, or some user-defined type.

The type of a value may be inferred or explicitly specified.

The following, in most languages, will be inferred to be an integer:

`23498`

The following, in most C-like languages, explicitly specifies that the value is of type "double":

`(double)23498`

<http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages>

Understand the Distinction

Dynamically typed vs Statically typed

Language Category S

(continued...)

Actions

1. **Assignment to a variable:** The variable's type is used to ensure that it can only be assigned values whose type is compatible. A language implementation does this by throwing an exception or generating an error if an attempt is made to assign a value whose type is not compatible with the variable's type. In some languages, "compatible" means the value's type must strictly be the same as the variable's type. In other languages, "compatible" means the value's type must be the same as or a subtype of the variable's type. Depending on the language, other definitions of "compatible" are possible. Compatible assignments of values replace the variable's current value.
2. **Invocation of operators:** Value types are used by the language to select the compatible specific operator when operators are polymorphic, and to ensure that argument types are compatible with operators' parameter types. See the "Operator Invocation" subsection below for more detail.

<http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages>

Understand the Distinction

Dynamically typed vs Statically typed

Language Category D1

Variables are not directly associated with a type. In other words, they do not have a "type" property.

Every value has a type, such as integer, float, string, or some user-defined type.
The type of a value may be inferred or explicitly specified.

PHP is a language in this category.

Actions

1. **Assignment to a variable:** Variables may be assigned any value of any type at any time.
2. **Invocation of operators:** Values' types are used by the language to ensure that the compatible specific operator -- if from a set of polymorphic operators -- is selected when invoked, and to ensure that values are compatible with operator parameters. Notably, many Category D1 languages do not expose this functionality to the language user, but it is used internally to (for example) dispatch the correct built-in "+" operator: the one implementing numeric addition if the operands are numeric, or the one implementing string concatenation if the operands are not numeric. See the "Operator Invocation" subsection below for more detail.

<http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages>

Understand the Distinction

Dynamically typed vs Statically typed

Language Category D2

Variables are not directly associated with a type. In other words, they do not have a "type" property.

Every value is represented by a string of characters.

ColdFusion and Perl are languages in this category.

Actions

1. **Assignment to a variable:** Variables may be assigned any value at any time.
2. **Invocation of operators:** Operators perform parsing as needed to determine whether each argument value (which is a string of characters) represents an integer, number, date, etc. See the "Operator Invocation" subsection below for more detail.

<http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages>

Understand the Distinction

Dynamically typed vs Statically typed

Notes on the Preceding Slides

In all three language categories, the only observable run-time state change is via variable assignment.

There is **no** other explicit state change.

In all three language categories, operators may interpret their operands as they see fit, including recognising values of various types that may be encoded within their operand values. For example, in PHP, the "is_numeric()" operator may be used to test whether or not its operand is numeric, which can include both operands of numeric type and numeric strings. (E.g., 123 is of numeric type, "123" is a numeric string.)

<http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages>

Understand the Distinction

Dynamically typed vs Statically typed

Operator Invocation (Part 1)

In most languages, a given operator name can have multiple operator definitions -- one operator definition for each combination of operand types. Each operator definition has its own operator signature, which consists of its name, operand types, and possibly its return type. For example:

- + (integer, integer) returns integer
- + (string, integer) returns string
- + (integer, string) returns string
- + (string, string) returns string

The interpreter (or compiler) uses the operand types to determine which definition to invoke. For example, an expression like `34 + "34"` can be written as `+(34, "34")`. Replacing the operand values with their types, we get `+(integer, string)`. That matches the third signature above, and so we invoke its corresponding operator definition.

The definitions determine the language behaviour. For example, in the above, only the first definition performs addition. The other three perform string concatenation. Thus, the language would return 68 given `34 + 34` and "3434" given `34 + "34"` or `"34" + 34`.

<http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages>

Understand the Distinction

Dynamically typed vs Statically typed

Operator Invocation (Part 2)

Given the following:

- `+(integer, integer)` returns integer
- `+(string, integer)` returns integer
- `+(integer, string)` returns integer
- `+(string, string)` returns string

The first three definitions attempt addition, the second and third parsing strings to determine if they represent numeric values. The last definition performs string concatenation. Thus, the language would return 68 given `34 + 34`, 68 given `34 + "34"`, `"3434"` given `"34" + "34"`, and an error given `34 + "splat"` because `"splat"` doesn't represent an integer.

<http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages>

Understand the Distinction

Dynamically typed vs Statically typed

Operator Invocation (Part 3)

Given the following:

`+(integer, integer)` returns integer
`+(string, string)` returns string

The first definition performs addition, the second performs concatenation. Thus, the language would return 68 given `34 + 34` and `"3434"` given `"34" + "34"`; `34 + "34"` is an error because there's no definition for `+(integer, string)`.

Some languages do not distinguish operand types outside of operators and treat all values as strings, so the only signature (for `+`) is effectively:
`+(string, string)` returns string

In such languages, when `+` is invoked it internally attempts to convert its operands to numeric values. If successful, the operator performs addition and returns a string containing only digits. If the conversion to numeric values is unsuccessful, the operator performs string concatenation on the operands and returns the result.

<http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages>

Understand the Distinction

Dynamically typed vs Statically typed

Operator Invocation (Part 4)

Other combinations are possible.

For example, given the following:

`+(integer, integer)` returns integer

`+(string, integer)` returns integer

`+(string, string)` returns string

The first two definitions attempt addition; the second parsing the string to determine if it represents a numeric value. Thus, the language would return 68 given `34 + 34`, 68 given `"34" + 34`, "3434" given `"34" + "34"`, an error given `"splat" + 34` because parsing the string fails, and an error given `34 + "splat"` because there is no operator definition for `+(integer, string)`.

<http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages>

Understand the Distinction

Dynamically typed vs Statically typed

Operator Invocation and Static Type Inference.

The return type of operators can be used to infer expression types prior to execution. For example, given operator signatures:

$+(integer, integer)$ returns integer
 $+(string, integer)$ returns string
 $+(integer, string)$ returns string
 $+(string, string)$ returns string

And given an expression like:

$34 + 38 + \text{"splat"} + 55$

It can be represented as operator invocations:

$+(+(+(34, 38), \text{"splat"}), 55)$

Now replace the operand values with their types:

$+(+(+(integer, integer), string), integer)$

By replacing each operator invocation with its corresponding return value until we can't simplify further, i.e., $+(+(+(integer, integer), string), integer) = +(+(integer, string), integer) = +(string, integer) = string$, we determine that the expression is of type "string".

Note that dynamic dispatch of various sorts – single dispatch, multiple dispatch, etc. – may complicate this.

<http://wiki.c2.com/?TypeSystemCategoriesInImperativeLanguages>

Understand the Distinctions

Single-dispatch vs Multiple-dispatch

- Single dispatch = conventional OO
 - Dynamically dispatch on the run-time type (as opposed to the declared or static type) of the (often implicit or invisible) first argument.
- Multiple dispatch = more powerful than conventional OO
 - Dynamically dispatch on the run-time type (as opposed to the declared or static type) of all arguments.
- Other forms of dispatch are possible...

Understand the Distinction

First-class vs Second-class

- Usually...
 - *First-class* constructs can be defined by expressions and assigned to (or bound to) variables or parameters at run-time and returned by operators/functions.
 - *Second-class* constructs can't.

Understand the Distinctions

Mutable vs Immutable

Mutable things can change. Immutable things can't.

Functional programming derives considerable benefit from avoiding mutability.

For example, if variables can be assigned once and only once, programs can become much easier to reason about.