

# Unusual primitives in programming languages

Raphael 'kena' Poss

January 2024

**modified:** 2024-01-18  
**slug:** unusual-primitives-in-programming-languages  
**category:** Programming  
**tags:** functional programming, language comparison, analysis, programming languages

This post explores *unusual primitive constructs* in programming languages, from data-like primitives to somewhat esoteric features. Unusual primitives are interesting because they indicate an attempt by the language designer to help the programmer think in a new way.

Acknowledgement: This writeup is a response and followup to Hillel Wayne's excellent [Unusual basis types in programming languages](#) (2024). If you have not read it yet, consider reading it as a prelude; a review is included below. I also recommend subscribing to Hillel's newsletter if you are interested in programming languages and PL design.

## Contents

<a href="#">Introduction</a>	2
<a href="#">Unusual primitives in the "data types" category</a>	3
<a href="#">Hillel's inventory</a>	3
<a href="#">Sets</a>	3
<a href="#">N-dimensional arrays</a>	3
<a href="#">Symbols</a>	4
<a href="#">Pairs</a>	4
<a href="#">Dataframes a.k.a. tables</a>	4
<a href="#">One more unusual primitive in its historical context: structs</a>	5
<a href="#">Other unusual data primitives</a>	5
<a href="#">Numbers with units</a>	5
<a href="#">Decimal floating-point</a>	6
<a href="#">Almost-data primitives</a>	6
<a href="#">Extra-functional state metadata</a>	7
<a href="#">Data safety under concurrent execution</a>	7

Control flow	7
Program structuring	7
Non-data, non-control primitives	8
Copyright and licensing	8

---

#### Note

The latest version of this document can be found online at <https://dr-knz.net/unusual-primitives-in-programming-languages.html>. Alternate formats: [Source](#), [PDF](#).

## Introduction

*Primitives* in programming languages are the basic building blocks from which other language and library constructs are derived. For example, `int` is a primitive type in C; the cons cell<sup>1</sup> is a primitive run-time data representation in Lisp and `GOTO` is a primitive control flow modifier in Basic.

Computing theory tells us it is possible to build arbitrarily complex languages using just one or two primitives. For example, it is possible to build every arithmetic operator using just boolean NAND gates<sup>23</sup>. So why do language designers commonly include more primitives?

One common answer is performance: primitives get special cased during compilation/execution and can be optimized better. Another common answer is programmer productivity: making common idioms simpler to write.

Yet, there are primitives in languages that are also *unusual*: very few languages include them, and they remain so for a while, sometimes years. Community interest in performance and productivity is not enough to see them propagate, or at least not quickly.

This is interesting! It is interesting because an unusual primitive is an attempt from the language designer to teach programmers to think in a new way.

In most cases, a primitive is only unusual because it was just invented; sooner or later it propagates to many languages and is not unusual any more. In the text below, we will designate those as “seeds”. For example, Simula 67 was the first language with objects, classes and inheritance<sup>4</sup>. At the time, these primitives were unusual, but in hindsight we can call them seeds. I find it a useful exercise to consider which unique primitives in contemporary languages are likely to become seeds over time.

In other cases, a unusual primitive does not catch on, by propagating to other languages. Or perhaps it remains specific to very few languages. Below, we will designate those as “isolates”. Failure to propagate can mean a few things: either a primitive is not primitive enough, i.e. it can be too easily expressed in terms of other primitives; or it is too complex to understand, teach or use. Or perhaps it is just too goofy. We will explore examples of each below.

Finally, some primitives used to be common and well-understood, and over time became rarer, i.e. less often included in new programming languages. From the perspective of a newly educated programmer, these primitives now appear unusual, even if they were neither seeds nor isolates in a historical context. We will designate those as “faded”.

To summarize:

Initially	Later	Now	Our designation	What makes it interesting
Unusual	Common	Common	<i>Seed</i>	Foundational knowledge that helps programming across multiple languages
Unusual	Unusual	Unusual	<i>Isolate</i>	Useful for designing “magic solutions” in niche domains
Unusual	Common	Unusual	<i>Faded</i>	Understand things that have been tried already and may not be useful any more

## Unusual primitives in the “data types” category

The following section explores primitives that define “data” in programming languages, i.e. things that can be printed out on paper while the computer is switched off.

### Hillel’s inventory

This section recaps and expands Hillel Wayne’s inventory from his [engaging newsletter](#). Hillel focuses on primitive constructs that have *dedicated syntax* in their respective languages.

#### Sets

Sets are non-ordered collections of unique values.

```
# Python
>>> {"a", "b"} | {"a", "c"}
{'c', 'b', 'a'}
>>> {"a", "b"} - {"b"}
{'a'}
```

Sets, when introduced, were seed primitives: introduced in Pascal (1971), unusual for a few years, then caught on in numerous other languages.

*How sets changed the way programmers think*: many algorithms become orders of magnitude simpler to write when one has access to set types.

#### N-dimensional arrays

As in, arbitrary array dimensions as a language primitive, i.e. not built by nesting.

```
julia> [1; 2;; 3; 4;; 5; 6;;;
        7; 8;; 9; 10;; 11; 12]
2×3×2 Array{Int64, 3}:
[:, :, 1] =
 1  3  5
 2  4  6

[:, :, 2] =
 7  9  11
 8 10  12
```

<sup>1</sup><https://en.wikipedia.org/wiki/Cons>

<sup>2</sup>[https://en.wikipedia.org/wiki/Functional\\_completeness](https://en.wikipedia.org/wiki/Functional_completeness)

<sup>3</sup>[https://en.wikipedia.org/wiki/One-instruction\\_set\\_computer](https://en.wikipedia.org/wiki/One-instruction_set_computer)

<sup>4</sup><https://en.wikipedia.org/wiki/Simula>

N-dimensional arrays are an isolate primitive: introduced in FORTRAN (1957), they were well-respected and often cited in the context of APL but never really propagated to many other languages. Currently mostly seen in APL derivatives like J, K and Julia.

Notably, C and C++ include N-dimensional array types as a primitive but they are so cumbersome to use in those languages that most programmers avoid them, preferring to use *liffe* vectors<sup>5</sup> instead.

*How N-dimensional arrays changed the way programmers think:* like sets, certain algorithms are orders of magnitude simpler to express when N-dimensional arrays are primitive types.

## Symbols

Symbols are immutable strings, commonly memoized, often internally represented as a numeric constant.

```
# Ruby
> puts :symbol1 == :symbol2
true
> puts :symbol1 + :symbol2
undefined method `+' for :symbol1:Symbol (NoMethodError)
```

Symbols are a faded primitive: introduced in Lisp (1960), then well-understood, studied and used until the late 1980s, nowadays unusual again.

*How symbols changed the way programmers think:* symbols made it many times easier to understand the data used by programs after the program was written (i.e. they made it easier to read program code) and during execution (i.e. they also made debugging and program reflection easier by giving names to special numeric values).

The relative importance of symbol primitive types faded with the introduction of more expressive and descriptive type systems, and compiler-generated run-time debugging metadata.

## Pairs

Pairs are unary forward relations. They are also an isolate primitive, never very common. Currently seen (as per Hillel) in Raku and maybe Alloy.

```
> (1 => 2){1}
2
> {"a" => 1, "b" => 2}.kv
(b 2 a 1)
```

*How pairs changed the way programmers think:* as per Hillel,

You can pull the element out of a list, so why not pull the kv pair out of a mapping? [...] [Pairs] make it easy to add optional flags to a function. I think keyword arguments are implemented as pairs, too.

## Dataframes a.k.a. tables

Dataframes are collection of records, usually non-ordered.

---

<sup>5</sup>[https://en.wikipedia.org/wiki/liffe\\_vector](https://en.wikipedia.org/wiki/liffe_vector)

```

# postgres
=> WITH t1(txt,num) AS (VALUES ('a',1),('b',2)),
      t2(num,txt2) AS (VALUES (1, 'c'))
  SELECT * FROM t1 NATURAL JOIN t2;
 num | txt | txt2
-----+-----+-----
   1 | a   | c
(1 row)

```

Seed or isolate primitive, depending on viewpoint. Tables were rather unusual when originally defined by E.F. Codd in relational algebra in 1970 and implemented short after in a few data-oriented languages. Even in 1974, when SQL was introduced, only few languages included tables as a primitive type.

Since then, dataframes/tables have become required primitives for all languages used for complex data analysis. In that way, they were seeds for an entire industry. However, none of the top ten mainstream programming languages besides SQL include dataframes as primitives. This was also true throughout the history of programming languages. From that viewpoint, dataframes are isolates.

*How dataframes changed the way programmers think:* by introducing algebraic joins, projections and selections as primitive operators, they made it possible to reason about and slice/dice entire datasets at a higher level of abstraction.

### One more unusual primitive in its historical context: structs

Hillel posited that Booleans, numbers, strings, lists and struct types are not so unusual. Let's inspect them more closely; we can already recognize that one of them is not like the others!

Booleans, numbers, strings and lists (ordered collections, like 1D arrays or linked lists) are definitely not unusual. They were manipulated and well-understood by people building algorithms even before computers and programming languages were first implemented.

Struct types, a.k.a records, on the other hand, were seed primitives! Before COBOL introduced structs as a primitive language feature (in 1960), programmers had to assign each field in a dataset to a separate variable, and each field needed to be copied manually to duplicate a record or pass it as argument. It took a few years after COBOL demonstrated the usefulness of primitive struct types for them to catch on, mainly via PL/I (1964) and Pascal (1970). Upon COBOL's introduction, struct types were thus quite unusual.

What struct types taught programmers, at the time, was *type/use orthogonality*: they made it possible to add/remove fields to a record type in one region of a program without changing every other region of the program that was accessing the records of that type.

(This way of thinkin was further developed with object orientation, but we had to wait 7 more years for that, until Simula 67 was introduced.)

### Other unusual data primitives

#### Numbers with units

Seen e.g. in FORTRAN. Primitive types with units couple a unit with the type of a number variable.

For example:

```

real :: distance = 5.0d0 * meter
real :: time = 2.0d0 * second
real :: speed = distance / time

```

This enables two things:

- it prevents the programmer from mistakenly mixing variables of incompatible units together.
- it enables native language support for unit conversion.

Sadly, numbers with units are an isolate primitive. This is sad because many bugs in robots and other computers that interact with humans can be traced to mistaken mixing of units in computations.

*How numbers with units change the way programmers think:* when a programmer learns of languages that provide units as primitives, they become more aware of the high probability of mistake around mixed-unit computations and may tread more carefully when programming with languages without them.

## Decimal floating-point

While binary floating-point types have become ubiquitous, *decimal* floating point<sup>6</sup> remains a rather unusual primitive.

As far as I know, the only mainstream language with a primitive decimal FP type is C#. As such, it is an isolate primitive.

Example use:

```
decimal x = 0.1m;  
decimal y = 0.2m;  
decimal z = x + y;  
Console.WriteLine(z);
```

(With binary floating point,  $0.1+0.2$  typically yields a value slightly different from  $0.3$ .)

The main purpose of decimal FP is to work effectively with financial numbers with a variable amplitude but a fixed number of significant digits.

*How primitive decimal FP changes the way programmers think:* by teaching how FP arithmetic works over any base. By being included in the “basic types” section of a language’s documentation, it forces programmers to become aware of the fundamental components of a FP number, and thus helps them “open the black box”, even when they have not studied a FP specification before. This transition does not typically happen when support for decimal FP is restricted to an optional language module or library.

## Almost-data primitives

- regular expressions
- function closures
- grammars
- inspectable scope dictionary
- txn id in pg sql

---

<sup>6</sup>[https://en.wikipedia.org/wiki/Decimal\\_floating\\_point](https://en.wikipedia.org/wiki/Decimal_floating_point)

## Extra-functional state metadata

- Rust lifetimes
- Koka effects
- tainted values
- Ocaml subtypes (?)

## Data safety under concurrent execution

- Atomic values / CAS
- Locks / semaphores
- Futures / I-structures
- Channels

## Control flow

- Intercal PLEASE
- Intercal COME FROM
- POSIX pipes
- Raku phasers
- Go's `go` / POSIX shell `&`
- Go's `select`
- TCL `expect`
- `unless`

## Program structuring

- Preprocessing (now isolate)
- Ocaml modules

## Non-data, non-control primitives

- Threads
- Processes (erlang)
- Processes (unix)
- interrupts / signals
- VMs
- Netgraphs
- capabilities

What's next?

- blockchains / IPFS
  - LLM related primitives
- 

## Copyright and licensing

Copyright © 2014-2026, [Raphael Poss](#). Permission is granted to distribute, reuse and modify this document according to the terms of the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/>.