The memory models that underlie programming languages

By [Kragen Javier Sitaker](). Last updated 2016.

There are about six major conceptualizations of memory, which I'm calling "memory models"[2], that dominate today's programming. Three of them derive from the three most historically important programming languages of the 1950s — COBOL, LISP, and FORTRAN — and the other three derive from the three historically important data storage systems: magnetic tape, Unix-style hierarchical filesystems, and relational databases.

These models shape what our programming languages can or cannot do at a much deeper layer than mere syntax or even type systems. Mysteriously, I've never seen a good explanation of them — you pretty much just have to absorb them by osmosis instead of having them explained to you — and so I'm going to try now. Then I'm going to explain some possible alternatives to the mainstream options and why they might be interesting.

Page Contents [more]

Introduction

Every modern programming environment handles all six of these memory models to some extent, which is one reason our systems are so complicated and hard to understand.

Here I analyze how each of these memory models ① represents attributes of entities, ② interacts with data serialization, ③ performs, and ④ supports decoupling an aspect of your program by limiting access to it to a small part of the program (making it "local" or "private").

Prologue: programs with atomic variables only

Let's start with a simple programming language with no ability to structure data, because it has no closures and no data types other than finite-precision numbers and booleans. Here's a BNF definition that should more or less suffice, with more or less the usual semantics, and the usual precedence rather than that implied by the grammar:

```
program ::= def*
def ::= "def" name "(" args ")" block
args ::= "" | name "," args
block ::= "{" statement* "}"
```

```
statement ::= "return" exp ";" | name ":=" exp ";" | exp ";" | nest
nest ::= "if" exp block | "if" exp block "else" block | "while" exp block
exp ::= name | num | exp op exp | exp "(" exps ")" | "(" exp ")" | unop exp
exps ::= "" | exp "," exps
unop ::= "!" | "-" | "~"
op ::= logical | comparison | "+" | "*" | "-" | "/" | "%"
logical ::= "||" | "&&" | "&" | "|" | "^" | "<<" | ">>"
comparison ::= "==" | "<" | ">" | "<=" | ">=" | "!="
```

So, for example:

```
def f2c(f) { return (f - 32) * 5 / 9; }
def main() { say(f2c(-40)); say(f2c(32)); say(f2c(98.6)); say(f2c(212)); }
```

Let's declare recursion illegal and make the language eager and call-by-value, like most programming languages, and make all the variables implicitly local and zero-initialized when the subroutine is called, so that no subroutine can have any side effects. In this form, this programming language only suffices to program finite state machines. You can compile it into a circuit. (Not a theoretical-computer-science circuit, which is just a boolean expression DAG; an actual physical circuit, which can incorporate registers.) Each variable that occurs in the text of the program can be assigned a single register, each subroutine can be assigned a register for its return address, and you need one more register for the program counter. If you run programs in this language on a machine with gigabytes of RAM, it will do you no good; it will never be able to use any more variables than the ones it started with.

This doesn't make the language useless; there are a lot of useful computations that can be done in finite space. But it really limits its abstraction power, even for those computations.

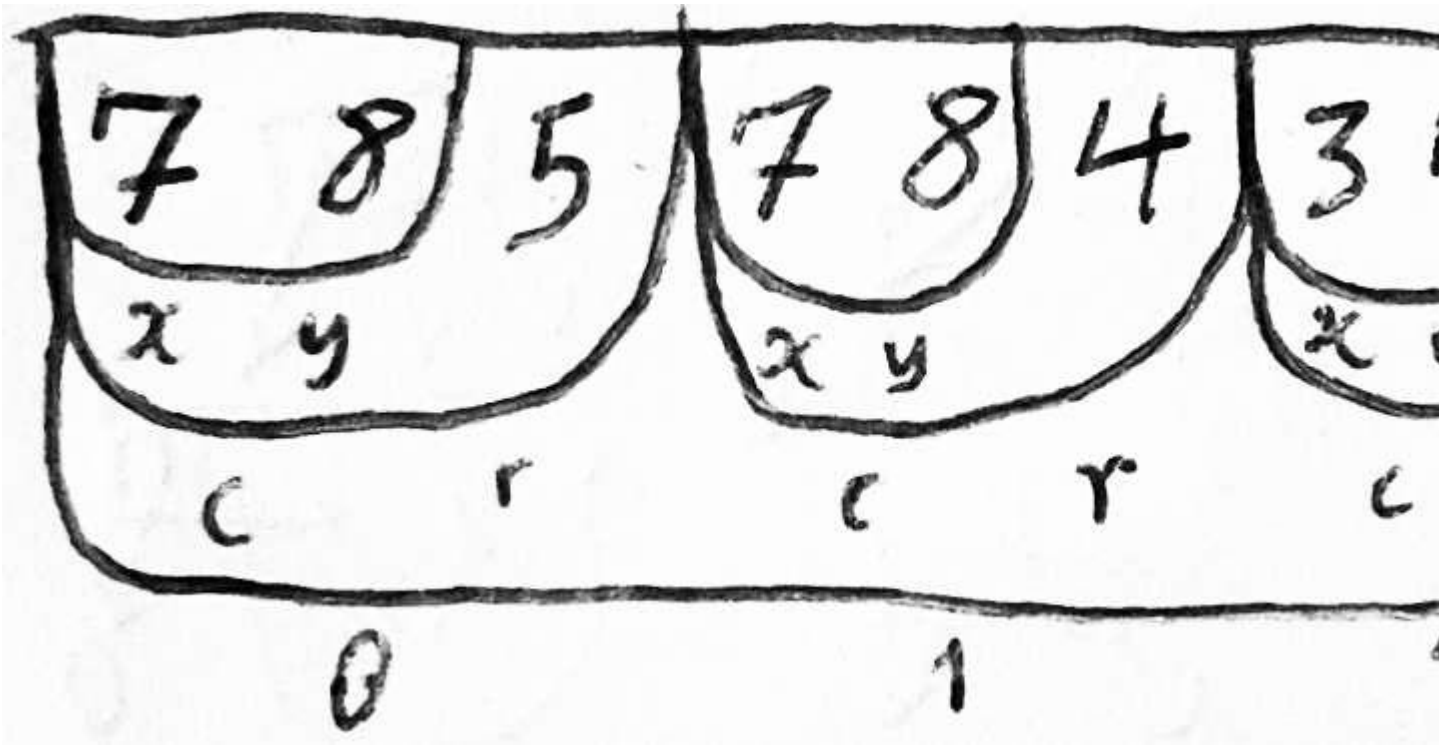
You could use `peek()` and `poke()` functions with the language to give you access to that memory — reading and writing a single byte at a given numerical address. And you could indeed use the memory effectively that way:

```
def strcpy(d, s, n) {
    while n > 0 { poke(d + n, peek(s + n)); n := n - 1; }
}
```

And those are more or less the facilities machine code and Forth give you. However, most programming languages don't stop there, and in fact many of them don't even provide `peek()` and `poke()`. Instead, they provide some kind of structure on top of the forbiddingly austere uniform array of bytes.

For example, even within the limitation of programming only finite state machines, nested records, arrays, and unions already provide enormous benefits.

NESTED RECORDS, THE COBOL MEMORY MODEL: MEMORY IS A TAX FORM



For COBOL, a data object is either indivisible — a fundamental object like a string or a number of some specific size — or it is an aggregate, either a record of data objects of different types stored one after the other, a union of alternative data objects that can be stored in the same location, or an array of a specific number of data objects of the same type stored one after the other.

(I am deviating significantly from COBOL terminology and taxonomy here to provide a simpler conceptualization of what it offers, with the benefit of 60 years of hindsight.)

This more or less represents the punched-card and paper-form origins of business data processing on computers, with its heritage of Hollerith's automation of the 1890 US census.

Any part of a COBOL program can read or change any part of this hierarchy of form fields at any time. (Again, more or less. I'm not sure you can actually change data that belongs to your input files but I don't care enough about that aspect of things.)

In this nested-record memory model, if you have several entities of the same type, each one will have an record of identical type (and size and subfields) dedicated to storing information about it; so all the information about a given entity will be contiguous in memory. You can load and store these chunks of contiguous data ("serialize" and "deserialize") on storage media such as disk, tape, or punched cards very simply. If you have several of them in memory at once, they might be in an array. An attribute of a type of entity — for example, the account type of bank accounts — is represented by a pair of byte offsets from the base address of records representing that entity that denote the beginning and end of that attribute. For example, an account object might have an account-holder field from bytes 10 to 35, which might contain a middle-name field from bytes 18 to 26.

There are a few interesting things to notice about this approach, some of which are actually benefits compared to how we usually do things today.

There are no pointers. That means there is no way to do dynamic memory allocation, no way to try to dereference a null pointer, no way to overwrite some area of memory with a wild pointer (although if two variables share storage with a REDEFINES clause, one can certainly overwrite the other; tagged unions avoid this problem), no way to run out of memory, no aliasing, and no memory spent on pointers.

On the other hand, it also means that every data structure in your program has an arbitrary limit on it, and the only way to use the same memory for different things at different times is to run the risk of using it for them both at the same time.

Nested records are quite frugal with memory. You only need to keep in memory the data pertaining to the entities you are currently concerned with. This means that you can usefully process megabytes of data on a machine with kilobytes of memory, which is in fact what people did with COBOL in the 1950s.

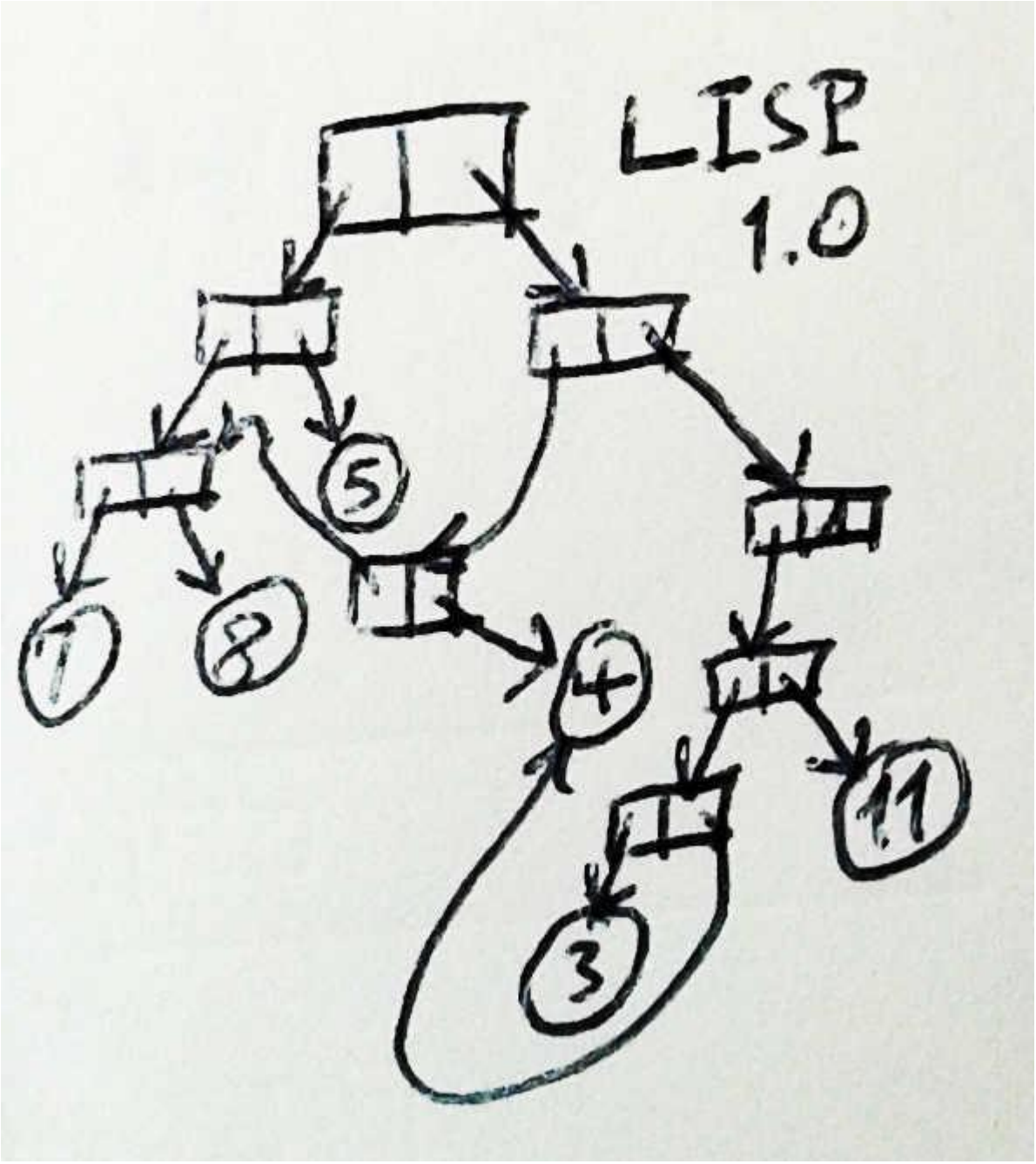
Each piece of data (field, subfield, item, whatever) has a single unique parent that immediately contains it, except for the top level that stands for memory as a whole.

In this memory model, if one part of the program has memory private to it (such as a stack frame or a private static variable), it can make some entities private to it by storing data about them there. This is useful for creating a local temporary variable and knowing it won't affect the execution of the rest of the program. However, the nested-record memory model does not provide a way for any part of the program to make an *attribute* private.

ALGOL (perhaps ALGOL-58, perhaps ALGOL-60) adopted the "record" construct from COBOL as its primary data structuring mechanism, other than arrays, and it is from ALGOL that nearly every other programming language has inherited this system, in one form or another.

C has almost precisely this set of data-structuring operators: primitive types like `char` and `int`, structs, unions, and arrays. However, C also has subroutines which not only take arguments but are recursive, necessitating something like stack allocation, and C also has pointers. Both of these extensions to the COBOL model more or less come from LISP.

Object graphs, the LISP memory model: memory is a labeled directed graph



LISP (even though it's Lisp now, it was LISP in 1959) could hardly be more different from COBOL. Not only does it have pointers, it almost has nothing *but* pointers. LISP's only data-structuring mechanism is something called a cons, which consists of two pointers, one of which is labeled "car" and the other of which is labeled "cdr". The value of every variable is a pointer. It may be a pointer to a cons, or a pointer to a symbol, or a pointer to a number (some Lisps use a pointer-tagging trick to avoid actually creating a number object in memory), or sometimes even a pointer to a subroutine, but it's a pointer.

Also, it has recursive subroutines with arguments, and in fact because of this and because of tail-recursion optimization, you can write programs that do anything at all without ever mutating the value of a variable.

(Most of this actually comes from Martin and Newell's IPL, but LISP was the incarnation of the IPL spirit that got adopted.)

Any object can be aliased by any number of pointers and mutated by way of any of them, so it has no unique parent.

This model is enormously flexible, in that it makes it pretty easy to write programs that perform operations like natural-language parsing, program interpretation and compilation, exhaustive search of possibilities, and symbolic mathematics. It also makes it easy to write a data structure (for example, a red-black tree) once and then apply it to many different kinds of objects. By contrast, COBOL-derived languages like C have significant difficulty with that kind of generalization, resulting in an enormous amount of duplicate code as programmers implement the same well-known data structures and algorithms over and over for new data types.

But it copes very poorly with memory limitations, it's bug-prone, and it requires a lot of ingenuity to implement efficiently. Because every object is identified just by a pointer, every object can be aliased. Every variable can be a null pointer. Because a pointer can point to anything, type errors (where a pointer to an object of one type is stored in a variable that is expected to be of some other type) are ubiquitous, and traditionally object-graph languages use run-time type checks in order to prevent debugging time from exploding, slowing the program down further.

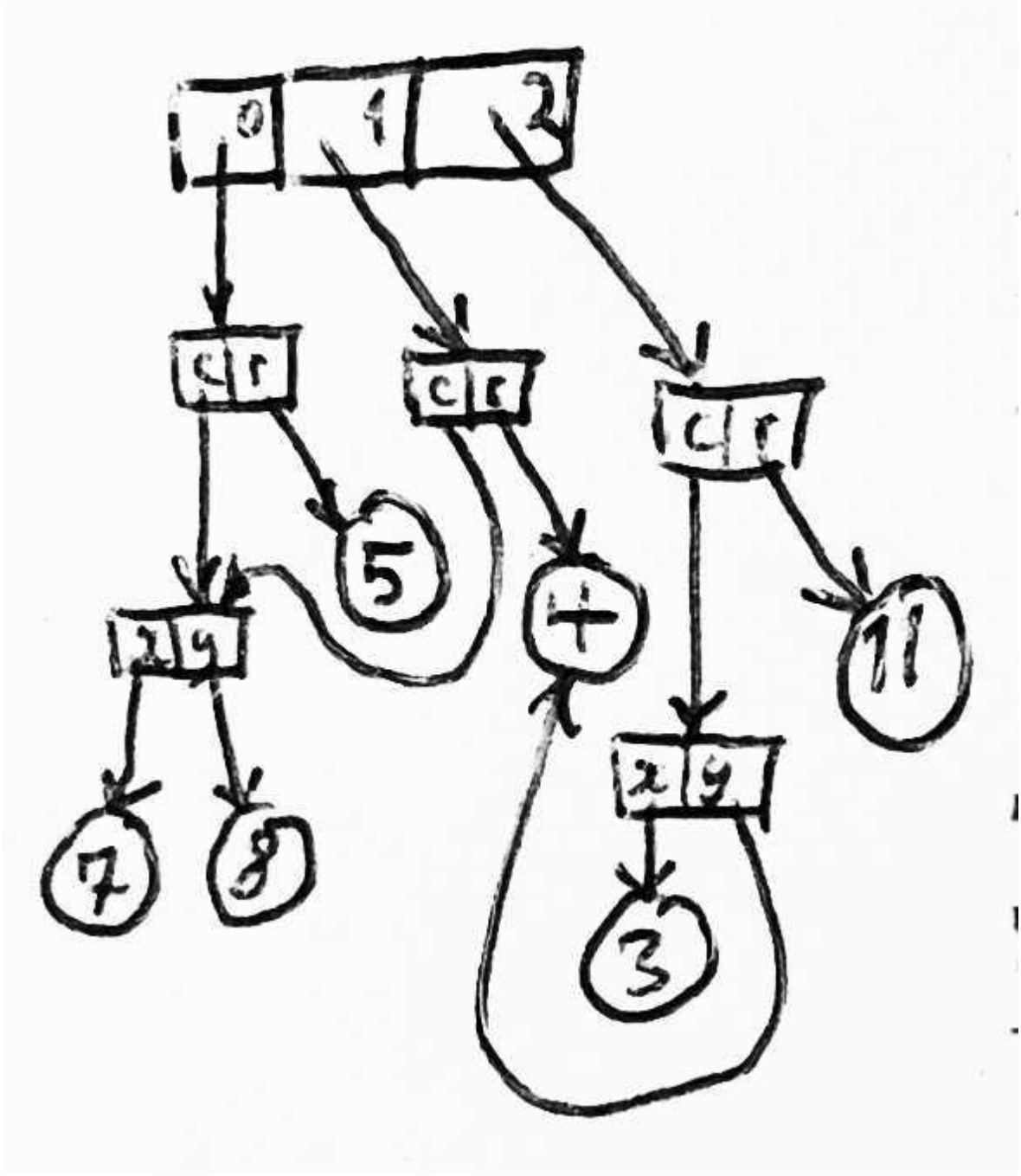
In this object-graph memory model, if you have several entities of the same type, each one will be identified by a pointer, and finding a particular attribute of an entity involves navigating the object graph starting from that pointer. For example, if you have an account object, you might represent it as an alist; then, to find its account-holder object (which might be shared with other account objects), you walk down the alist until you find a cons whose car is ACCOUNT-HOLDER, and take its cdr. Then, to find the middle-name of that account-holder, perhaps you index into a vector of account-holder attributes, getting a pointer to the relevant middle-name, which might be a string or, in ancient Lisps that didn't have strings, a symbol. Updating the middle-name might involve mutating that string, updating the vector to point to a new string, or constructing a new alist with a new account-holder object, depending in part on whether the account-holder might in fact be shared with other account objects, and whether it's desired for the other accounts for the same account-holder to have their name updated as well.

Garbage collection is nearly a necessity in these languages, and from McCarthy's discovery of Lisp in 1959 until Lieberman and Hewitt's discovery of generational garbage collection in 1980, programs using the labeled-directed-graph memory model usually spent between a third and half of their time running the garbage collector. Before that, some computers were built with multiple processors specifically in order to have another processor to run the garbage collector on.

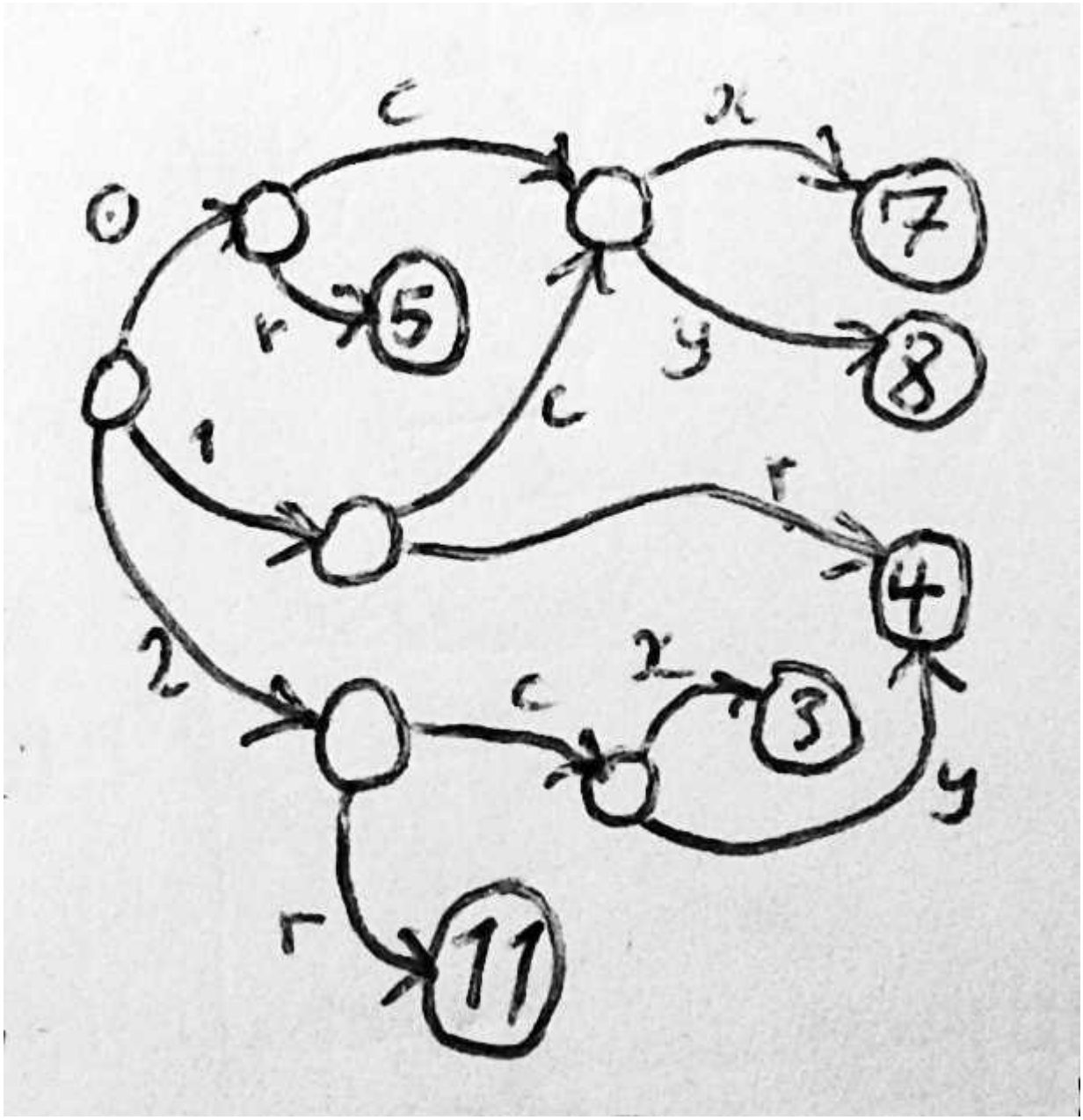
Object-graph languages put heavy demands on garbage collectors, not just because they prefer allocating new objects over mutating existing objects, but also because they tend to have a *lot* of pointers. COBOL-derived languages like C and Golang give the garbage collector an easier time, because they perform fewer, larger allocations; they tend to mutate the allocated objects instead of allocating modified versions; and programmers tend to nest records rather than link them with pointers where possible, so the pointers only occur where polymorphism, nullability (which can be thought of as a special case of polymorphism), or aliasing are desired.

Serializing an object graph is a bit tricky, both because it can contain circular references, but also because the part you want to serialize may contain references to a part you don't want to serialize, and you have to special-case both of these. For example, in some systems, a class instance contains a reference to its class, and the class contains references not only to the current versions of all the methods but also to its superclass, but maybe you don't want to serialize the entire bytecode for the class in every serialized object. Also, when you deserialize two objects that previously shared a reference (the two accounts for the same account holder mentioned earlier), you probably want to preserve that sharing. Finally, the particular policy you apply to these questions may vary depending on what purpose you're serializing for.

Like the nested-record model, the object-graph model allows you to make all the attributes of a particular entity local to a particular part of your program — you simply don't leak any references to its data structure to the rest of the program — but not to make a particular attribute of all entities private. However, unlike the nested-record model, the object-graph model reduces dependencies on the *memory size* of any given node, which opens the door to object-oriented inheritance, which *does* give you some private attributes, despite its other serious problems.



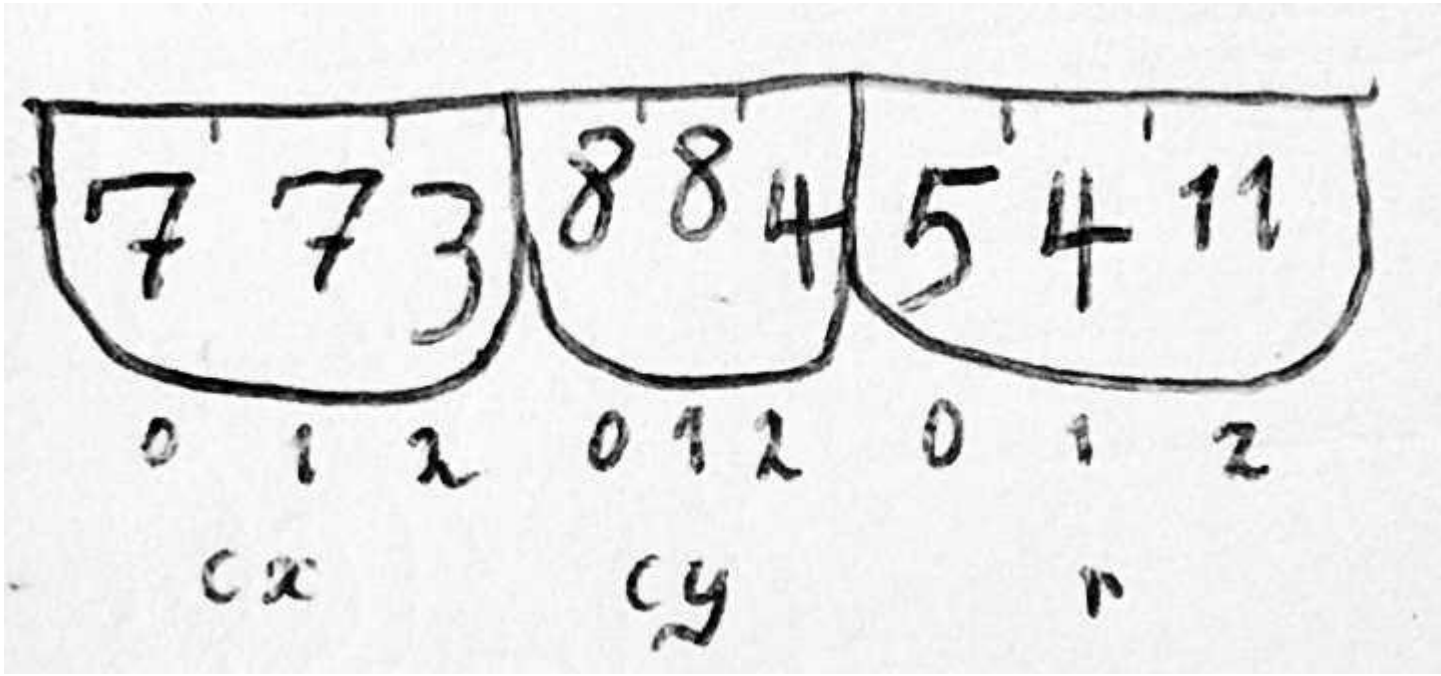
Most currently popular programming languages use this model. Not only current Lisps, but also Haskell, ML, Python, Ruby, PHP5, Lua, JavaScript, Erlang, and Smalltalk use it. All of them have expanded the set of object types that exist in memory beyond a simple pair; typically they include at least arrays of pointers and hash tables mapping either strings or pointers to other pointers. Some also include tagged unions and immutable records. Hash tables, in particular, offer a kind of way to add new properties to existing entities without affecting other code that uses those entities — in most cases!



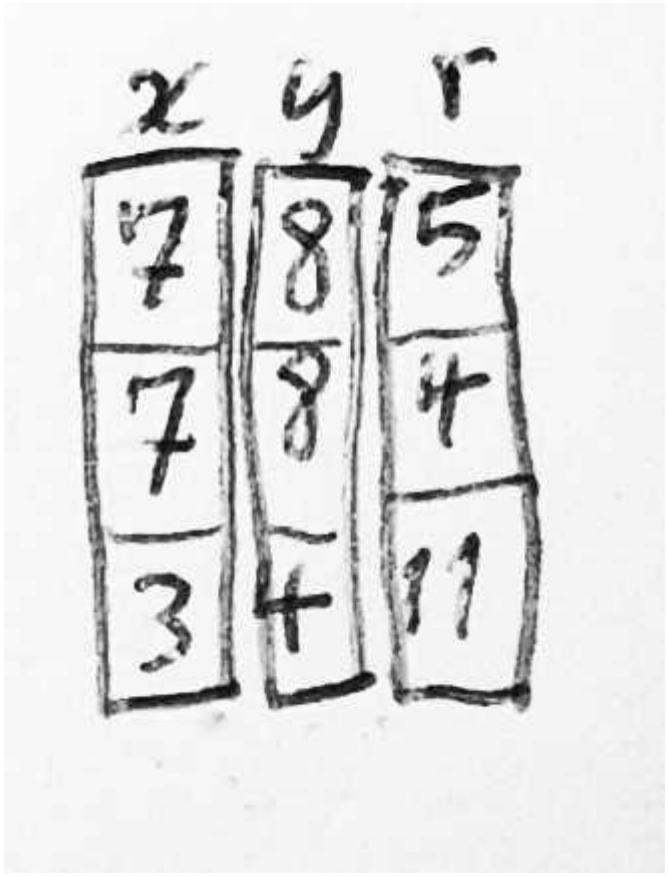
In general, in these languages, you can only follow graph edges in the direction they point, and the edge labels are constrained to be unique within the source node (a cons has only one car, not two or ten), but not its destination. Ted Nelson's ZigZag data structure is an exploration of what happens if you require them to be unique in both the source and destination. UnQL is, in some sense, an exploration of eliminating the uniqueness constraint entirely.

Java (and C#) uses a slightly modified version of this memory model: in Java, there are things in memory that are not pointers, called "primitive types". Among other things, this means you can't store them in normal container types, although Java has worked desperately to paper over this deficiency in recent years.

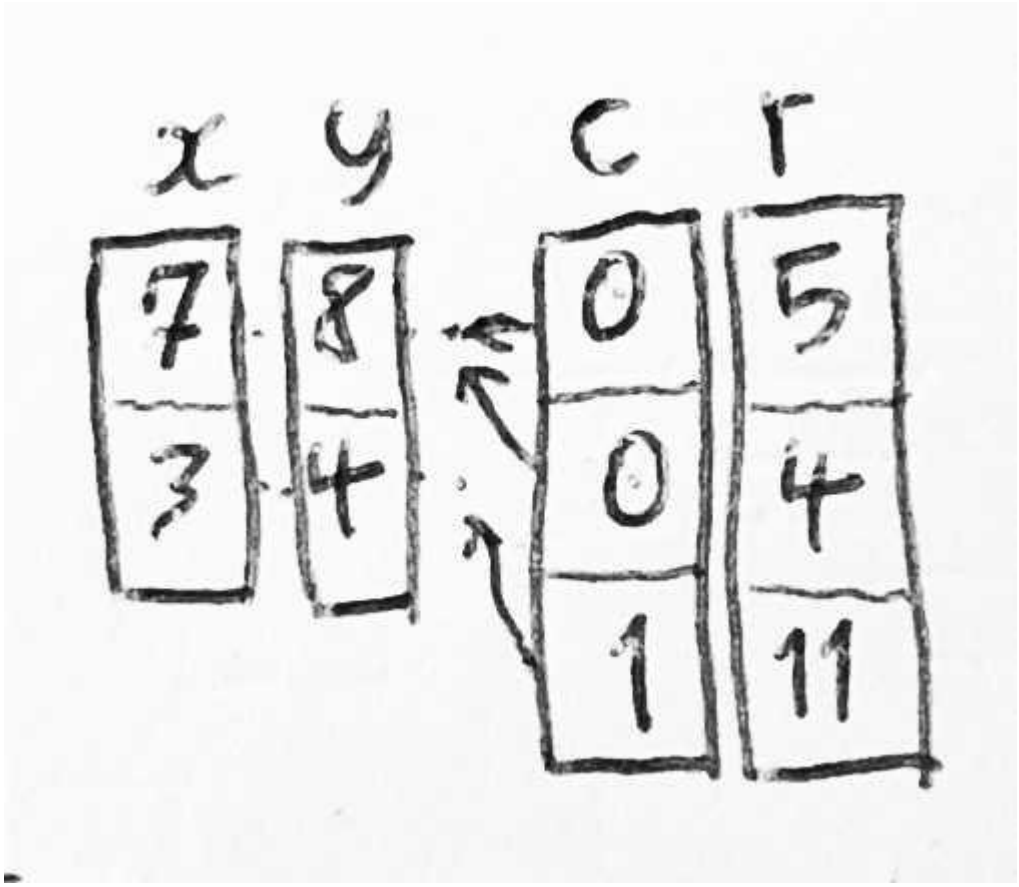
PARALLEL ARRAYS, THE FORTRAN MEMORY MODEL: MEMORY IS A BUNCH OF ARRAYS



FORTRAN was designed for numerical modeling of physical phenomena, which was one of the earliest uses for computers, usually known as “scientific computing”. At the time, scientific computers were distinguished from the “business computers” that COBOL was designed for by a number of features: they used binary instead of decimal; they didn’t have bytes, only fixed-length words; they supported floating-point math; and they featured faster computation and slower I/O.



Typically, these models involve a lot of linear algebra on large numerical arrays, which has to be computed as fast as possible, and that's what FORTRAN (later Fortran) was optimized for: efficient use of multidimensional arrays. FORTRAN not only didn't have recursive subroutines, pointers, or records, at first it didn't have subroutines at all! When it did get them, which I think was in FORTRAN II, they had parameters, which could be arrays, which was something ALGOL 60 still couldn't figure out how to do properly. Since arrays were the only non-primitive type, the only possible element types for the arrays were primitive types such as integers or floating-point numbers.



In the parallel-array memory model that evolved in this FORTRAN world, if you have several entities of the same type, each one will be identified by an integer offset that is valid in any of several arrays, and finding a particular attribute of an entity involves indexing the array for that attribute with the index of that entity. If we retreat from the 1950s a little bit and allow ourselves a primitive character type that we can make arrays out of, but sticking purely to parallel arrays for data structuring, then the previous example of finding the middle name of an accountholder could proceed in any of the following ways:

1. IM = IMDNAM(ICCHLD(IACCTN))
   IA = ISTR(IM)
   IE = ISTR(IM+1)
   After these four array-indexing operations, the account holder's middle name is in characters [IA, IE) of the CCHARS array.
2. IM = IMDNAM(IACCTN), then proceed as before, if you don't have a separate set of arrays for the attributes of account holders, as you very well might not.

3. Instead of IMDNAM, use CMDNAM, an N × 12 array of characters, with one 12-character column for the middle name of each account holder.

In this memory model, a subroutine can access any index in an array that has been passed as an argument or otherwise shared with it, reading or writing it any number of times at random.

This is what is meant by "programming FORTRAN in any language": nearly every programming language has arrays of primitive types. Even in assembly language or Forth, arrays are not hard to make. Awk, Perl4, and Tcl additionally give you dictionaries, which are not first-class objects because those languages are not object-graph languages, but those dictionaries work fine in place of arrays for storing attributes of entities, and allow you to identify the entities by strings instead of integers.

An amusing detail of this is that, at the machine level, in the simple case, parallel arrays generate almost exactly the same code as struct members referenced through pointers, as in the nested-record model. For example, here's `b->foo`, where `b` is a pointer to a `struct` with a 32-bit `foo` member:

```
  40050c:   8b 47 08                mov    0x8(%rdi),%eax
```

And here's `foos[b]`, where `b` is an index into a `foos` array of 32-bit things:

```
  400513:   8b 04 bd e0 d8 60 00    mov    0x60d8e0(,%rdi,4),%eax
```

In both cases we add an immediate constant, representing the attribute we're interested in, to a variable in a register, indicating which entity we're looking at. The only machine-code-level difference is that, in the second case, we multiply the index by the item size, and the immediate constant is a bit larger.

(You'll note that the instruction format is different, though, and not all CPU architectures support such large constant offsets; the array's base address, or a sufficiently large struct-field offset, may have to be loaded into a register on some machines.)

Adam N. Rosenberg advocates always programming in the parallel-arrays style, and explains his reasoning at book length. I am not convinced that this is a good idea, but he gives the best modern defense of the idea that I've seen.

Briefly, parallel arrays are cache-friendly, support different visibility for different attributes, support setting watchpoints, provide a sequence that can be meaningful, and support multidimensional indexing (where an attribute is a property of a tuple of entities, rather than just one entity). I would add that they also allow you to write subroutines that abstract over attributes, since they reify each attribute at run-time: you can write a `sum` function or a `covariance` function that can be applied to arbitrary attributes.

The second point here is particularly interesting: although parallel arrays don't allow you to make an entity private to a particular part of your program, they *do* allow you to make an *attribute* private.

I don't like parallel arrays because they're so error-prone. The compiler can't tell which arrays a given index is or isn't valid for, and neither can the debugger or garbage collector. Parallel arrays mean a type error — storing an identifier for one kind of entity in a variable

that is expected to hold identifiers for some other kind of entity — will very frequently give you incorrect answers instead of a compile-time or run-time error message. (Especially if you don't have array bounds checking turned on.). Also, because parallel entities reify attributes rather than entities at run-time, creating and deleting entities is error-prone, subroutines tend to have a large number of parameters (increasing the cost of abstraction and thus the size of subroutines), and you often suffer the same kind of arbitrary limits that you suffer with the nested-record model.

Parallel arrays are very easy and efficient to serialize, particularly if you aren't concerned about portability to different machine architectures, but they tend to require you to serialize or deserialize an attribute for all entities it applies to at the same time.

Fortran is not the only thing to encourage you to organize your memory as parallel arrays. Octave, Matlab, APL, J, K, PV-WAVE IDL, Lush, S, S-Plus, and R are all significant parallel-array-oriented programming languages; Numpy, Pandas, and OpenGL are parallel-array-oriented libraries; and as I explained above, Perl4, awk, and Tcl are to some extent parallel-dictionary-oriented. Some of these reduce the risk of parallel arrays getting out of sync by making the arrays immutable once populated, or at least encouraging the creation of new arrays; Pandas, K, and the parallel-dictionary variant reduce the risk of type errors by encouraging you to index your arrays by things that aren't integers.

Because parallel arrays facilitate pushing loops over sets of entities into leaf functions, they tend to eliminate interpretation overhead as such, so APL was able to do high-performance numerical work even in an interpreter in the 1960s. Various features of modern hardware increase the pressure to use parallel arrays to get better performance: the increasing gap between CPU speed and memory speed, the SIMT architecture of GPUs, and the SIMD instructions that have been added to CPUs to increase the ratio of ALU silicon to control silicon. As a result, game programmers have been returning to parallel arrays, under the names of "data-oriented design" and (sort of) "entity systems". Numerical ("scientific") computing has never abandoned parallel arrays.

Interlude: why is there no Lua, Erlang, or Forth memory model?

So those three memory models correspond roughly to three basic kinds of data structures: the record, the linked list, and the array, each one giving one of them. But we know that there are a few other data structures that are just as ubiquitously applicable: Lua is organized around finite maps (dictionaries; it calls them "tables"), Erlang is organized around active shared-nothing processes that enqueue messages to each other (one variant of Hewitt and Agha's actor model), and Forth is organized around stacks.

I don't really know the answer. Lua and Erlang are basically object-graph oriented. Forth programs often just use parallel arrays, although the Forth dictionary is invariably a linked list. It seems like you probably *could* build a system where the program's view of memory was as a hash table, or some kind of non-graph-structured society of actors rather than an object graph, or two or more stacks. Linda was an exploration of non-graph-structured communication among actors through a tuple space, and maybe that could be extended into a whole language.

But at the end I'm going to explore some other alternatives.

Pipes, the magnetic tape memory model: the moving finger writes, and, having writ, moves on

Unix pipes are the simplest kind of memory of all (and they have their hardware analogies). The only operations they support are writing a byte (or, as an optimization, several bytes), reading a byte, and perhaps closing (from either end). (Actual magnetic tape actually wrote and read in blocks, but we'll ignore that.) You would not, generally, read and write on the same pipe in the same program.

This kind of append-only storage turns out to be entirely adequate for some algorithms; MapReduce is not far from operating in this fashion, but also the typical problem of tokenization with lex, for example, uses just such a minimal interface to its input.

Python's iterators and generators, the C++ STL's forward iterators, D's forward ranges, and Golang's channels are all examples of such pipes or channels, with their purely sequential data access.

What would a programming language memory model based entirely on pipes look like? You would have operations to read data items from pipes (of primitive types only, maybe) and write them. Consider the example programming language from the introduction. Given pipes with `empty`, `get`, and ``put subroutines, you could write a merge function in it that you could use for mergesorting, although it's not quite as easy as in Python or whatever:

```
def merge(in1, in2, out) {
    have1 := 0;
    have2 := 0;
    while !empty(in1) || !empty(in2) {
        if 0 == have1 && !empty(in1) {
            val1 := get(in1);
            have1 := 1;
        }
        if 0 == have2 && !empty(in2) {
            val2 := get(in2);
            have2 := 1;
        }
        if 0 == have1 {
            put(out, val2);
            have2 := 0;
        } else {
            if 0 == have2 || val1 < val2 {
                put(out, val1);
                have1 := 0;
            } else {
                put(out, val2);
                have2 := 0;
            }
        }
    }
}
```

Note that the above function, and the mergesort including it that I'm too lazy to write at the moment, only assumes that there are at least four pipes, the pipes have enough capacity

for the data, and that it can distinguish between them; it doesn't need to create pipes, destroy pipes, or pass pipes through pipes. The `in1`, `out`, etc., arguments don't need to be some kind of first-class pipe object; they could just be integers. (And indeed that's how Unix programs handle Unix pipes: by file descriptor index.) Or they could in fact be first-class pipe objects that can be passed as arguments but not passed through pipes themselves.

You could imagine a multithreaded control flow system in which you could fork off different threads, and threads that try to read from empty streams block until data becomes available. You'd have to use pipes instead of arrays or records; probably you'd have to use one stream for each attribute. Then the runtime could schedule the different pipe-processing threads as you saw fit, perhaps running some of them on different machines.

The π-calculus is a language that sort of uses only pipes; it's a kind of concurrent channel-oriented alternative to the λ-calculus. As Jeannette Wing explains,

Let P and Q denote processes. Then

- P | Q denotes a process composed of P and Q running in parallel.
- a(x).P denotes a process that waits to read a value x from the channel a and then, having received it, behaves like P.
- ā 〈x〉 .P denotes a process that first waits to send the value x along the channel a and then, after x has been accepted by some input process, behaves like P.
- (νa)P ensures that a is a fresh channel in P. (Read the Greek letter "nu" as "new.")
- !P denotes an infinite number of copies of P, all running in parallel.
- P + Q denotes a process that behaves like either P or Q.
- 0 denotes the inert process that does nothing.

The example π-calculus code she gives is:

!incr(a, x).ā 〈x+1〉 | (νa)($\overline{\text{incr}}$ 〈a, 17〉 | a(y))

which fires up a "server" with an infinite number of processes, each of which listens on the channel incr for one message, decomposes it into a and x, then sends x+1 on the channel a, then terminates; all of this in parallel with two other parallel processes which share a fresh channel named "a", which one of them sends on the "incr" channel along with 17, while the other awaits for a message to be sent on that channel, binding it to y.

The π-calculus is probably not going to become a practical programming system, and neither will other similar process calculi; but it suggests that it is at least possible to program using only pipes as memory. Note, though, that the above version of it smuggled in tuples and labeled graphs of processes! And if you don't do that, I'm not certain it's universal.

"Flow-based programming", as embodied by the NSA project known as Apache NiFi, is a different approach to generalizing the Unix pipes-and-filters paradigm.

Directories, the Multics memory model: memory is a string-labeled tree with blob leaves

The Unix (or Windows, or MacOS, or Multics) hierarchical filesystem is another kind of organization of memory, usually durable memory made of spinning rust, and shell scripts, in particular, use it extensively. In its basic form, it's a restriction of the object-graph model in which each node has a single unique parent and is either a "directory", which is a string-indexed dictionary of links to child nodes, or a "regular file", which is a mutable blob (sequence of bytes). A significant enhancement is a third kind of node, a "symbolic link", which is a path to follow, either from the root of the tree or from its parent directory, to find the desired node — if, in fact, it exists at all.

Typically, in a hierarchical filesystem, to represent the attributes of an entity, you serialize them into a regular file, along with many other entities! But that's mostly because the system-call interfaces are so slow and clumsy, and because the per-node overhead is typically on the order of hundreds of bytes. You can certainly, instead, use a directory for each entity, storing the value of its attribute "x" in a file called "x" within it. This is what Unix does for storing information about users, for example, or different versions of a software package, or how to compile different software packages.

The hierarchical filesystem occupies a middle ground when it comes to local variables. Typically, any tree node is accessible by traversing the tree, but you can be pretty sure that software you have running over in one part of the filesystem isn't going to notice what files you've created in some other part of the system; and, in most cases, you can add new files to directories without breaking the software that's already using those directories, although you have no guarantee of this.

The unique-parent property makes serialization and deserialization relatively straightforward.

There have been a variety of programming systems that more or less work this way. MUMPS, which still runs the US Veteran's Administration, is more or less this (although typically "files" are limited to 4096 bytes, and a node can be both a "directory" and a "regular file"). IBM's IMS database, which is also still in substantial use, has a very similar data model; the nodes are called "segments", and there is a schema imposed on the database. And, a few years back, Mark Lentczner began to develop a modern, object-oriented programming environment called Wheat, based on this model for developing web apps as a better alternative to PHP; I may have a bit of a personal attachment to it.

In Wheat, each function's activation record is a "directory", with its variables as (dynamically typed, rather than blob) "files" in it. Some subtrees of the "filesystem" are persistent, but not others; access to remotely hosted data is transparent. Essentially, Wheat sought to eliminate the impedance mismatch between the persistent, hierarchically-named, globally-accessible resources of the Web, and the program's own conception of memory.

Relations, the SQL memory model: memory is a collection of mutable multivalued finite functions

This is, in a sense, the most abstract of all the memory models.

A "multivalued function" is usually called a "relation" in mathematics. cos is a function: for any θ, cos(θ) has a single well-defined value. $\cos^{-1}$ is a relation: $\cos^{-1}(0.5)$ has many values, although by convention we turn it into a function by picking just one. You could think of cos as a set of ordered pairs such as (0, 1), (π/2, 0), (π, -1), (3π/2, 0), and so on, and the inversion operation is then just a matter of reversing the order of the pairs: (1, 0), (0, π/2), (-1, π), (0, 3π/2), and so on.

The relations we consider here, however, are more general than binary relations: rather than consisting of sets of 2-tuples, they're sets of n-tuples for some fixed n. You could consider, for example, the ternary relation between an angle, its cosine, and its sine: (0, 1, 0), (π/2, 0, 1), (π, -1, 0), and so on.

Most of the systems for relational programming have only limited support for infinite relations like cos, because it's very easy to construct undecidable problems with them.

To be more concrete, here's a section from the `permissions.sqlite` table for my Firefox install:

```
sqlite> .mode column
sqlite> .width 3 20 10 5 5 15 1 1
sqlite> select * from moz_hosts limit 5;
id   host                 type        permi  expir  expireTime       a  i
---  -------------------  ----------  -----  -----  ---------------  -  -
1    addons.mozilla.org   install     1      0      0
2    getpersonas.com      install     1      0      0
5    github.com           sts/use     1      2      1475110629178
9    news.ycombinator.com  sts/use     1      2      1475236009514
10   news.ycombinator.com  sts/subd    1      2      1475236009514
```

For the relational model, each of these columns is in some sense a function of the primary key, which happens to be the `id` column on the left; so you could say host(1) is `addons.mozilla.org`, host(2) is `getpersonas.com`, type(5) is `sts/use`, and so on. So far, this is just a hash table.

Where this breaks from the usual kind of functional programming, though, is that you can go either way: rather than merely asking for host(9), you can ask for host^{-1}('news.ycombinator.com'), which turns out to be multivalued:

```
sqlite> select id from moz_hosts where host = 'news.ycombinator.com';
id
---
9
10
```

Furthermore, you can compose these multivalued functions together:

```
sqlite> .width 0
sqlite> select min(expireTime) from moz_hosts where host = 'news.ycombinator.com';
min(expireTime)
---------------
1475236009514
```

Typically to represent an entity in SQL, you use a row in a table (aka a tuple in a relation), and to represent its relationships with other entities, you identify each entity by some unique set of attributes, called a key, like the id in the above table, and then include the key of one entity in some other column of the other. Returning to our example of the middle name of the account-holder, we could say this:

```
select accountholder.middlename
from accountholder, account
where accountholder.id = account.accountholderid
and account.id = 3201
```

or perhaps just leave off that last clause and get the middle names of all account holders, rather than just one.

SQL is not the only realization of the relational model, but it is by far the most popular. It recently accidentally became Turing-complete and thus capable of taking over the world.

This might seem like an irrelevant diversion from real programming languages like Lisp, FORTRAN, and C, since SQL as a programming language is more of a curiosity than a practical utility. But consider the example programming language I described at the beginning. If augment it with invoke SQL statements, store the primitive-typed results in its variables, and iterate over the result sets, it becomes a practical programming system, if perhaps a somewhat clumsy one. (In fact, what I just described is more or less PL/SQL.)

This is not about changing the denoted or expressed types in the language. Even if the programming language can still only handle numbers as variable meanings or expression values, once it is able to store those numbers in relations and fetch them back again via queries, its power increases immensely.

SQL suffers from some limitations: table names exist in a global namespace, and both rows and columns in a table are globally accessible (you can't have private rows or columns, although in a sense the result of a query is a private table, and can be used as such); each column can typically only hold primitive data types like numbers and strings (at least modern databases no longer make the strings fixed-width like COBOL); and it is usually painfully slow to run.

Looked at a certain way, SQL operations, and thus most of its benefits and limitations, map very closely to those of parallel arrays:

```
for (int i = 0; i < moz_hosts_len; i++) {
    if (0 == strcmp(moz_hosts_host[i], "news.ycombinator.com")) {
        results[results_len++] = moz_hosts_id[i];
    }
}
```

Or, perhaps more clearly showing where SQL saves you effort, doing a sort-merge join with parallel arrays is far from unmanageable; instead of this SQL:

```
select accountholder.middlename
from accountholder, account
where accountholder.id = account.accountholderid
```

You can do something like this in C with parallel arrays:

```
int *fksort = iota(account_len);
sort_by_int_column(account_accountholderid, fksort, account_len);
int *pksort = iota(accountholder_len);
sort_by_int_column(accountholder_id, pksort, accountholder_len);
int i = 0, j = 0, k = 0;
while (i < account_len && j < accountholder_len) {
    int fk = account_accountholderid[fksort[i]];
    int pk = accountholder_id[pksort[j]];
    if (fk == pk) {
        result_id[k] = fk;
        result_middle_name[k] = accountholder_middlename[pksort[j]];
        k++;
        i++;                // Supposing accountholder_id is unique.
    } else if (fk < pk) {
        i++;
    } else {
        j++;
    }
}
free(fksort);
free(pksort);
```

Here `iota` is something like the following:

```
int *iota(int size) {
    int *results = calloc(size, sizeof(*results));
    if (!results) abort();
    for (int i = 0; i < size; i++) results[i] = i;
    return results;
}
```

And `sort_by_int_column` can be written more or less as follows:

```
void sort_by_int_column(int *values, int *indices, int indices_len) {
    qsort_r(indices, indices_len, sizeof(*indices),
            indirect_int_comparator, values);
}

int indirect_int_comparator(const void *a, const void *b, void *arg) {
    int *values = arg;
    return values[*(int*)a] - values[*(int*)b];
}
```

It happens that SQL implementations use a variety of tricks under the covers to achieve reasonable efficiency on difficult queries, at the cost of efficiency on easy queries, but that doesn't affect the abstraction seen by the user program. Well, not much.

People often say SQL is a declarative language, in which you need not say how a result is computed, only what result is desired. I think this is only partly true, and declarativeness is not a binary predicate that is true or false of a language; it is a matter of the level of abstraction of the description, which is an infinite continuum, and in practice concerns that are supposedly below the level of abstraction at which you are programming inevitably creep in, if only for efficiency reasons.

Perhaps more interesting, if less pure, realizations of the relational model are Prolog and miniKANREN, which combine relational programming with object-graph recursive data

structures and achieve truly astonishing power. A fairly simple miniKANREN program can, for example, generate an infinite series of programs that output their own source code ("quines") in a reasonable amount of computation time.

The general field of constraint programming, in which you specify some properties of the answer you want ("constraints" it must fulfill), and then the system undertakes to search for the answer, is experiencing major growth right now as SAT and SMT solvers improve dramatically year by year.

NOTES

Essentials of Programming Languages proposes to analyze programming languages first according to what types of values that variables can denote and what types expressions may evaluate to, which is a great deal more informative than merely analyzing syntax, but I feel that it mostly glosses over this deeper layer.

[2] I'm calling these six conceptualizations "memory models", even though that term used to mean a specific thing that is related to 8086 addressing modes in C but not really related to this. (I'd love to have a better term for this.)