# Ecstasy: XVM and the XTC Language

Version: DRAFT-20180913

*Software is music: Elegance in simplicity; harmony in vision; rhythm in motion.*

# Introduction

Imagine a system of execution. The ingredients are well known at a conceptual level: Instructions, conditionals, control flow, state, loads, and stores. Processors, virtual machines, languages: Each a tool to support the description of a desired outcome. Processes, threads, fibers, tasks, jobs: Each a picture of execution.

We create systems of executions as abstractions that exist between the definite world of digital processing and the indefinite and limitless world of programming. In the immortal words of Fred Brooks writing in *The Mythical Man Month*:

Finally, there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures.

Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.

This document describes a system of execution, the XVM[[1]](#footnote-1). It exists in the abstract as defined by this document, and much like similar technologies that have preceded it, it is expected that compliant implementations could be implemented in software, with execution efficiency provided by translation to the execution language of the underlying machine, whether it be hardware or software.

It is this simultaneous duality of purpose – to provide both a natural and immersive abstraction, and to maximize the potential efficiency of execution – that creates a dramatic tension in design. Most computer languages and their associated execution machinery are examples of the necessary trade-offs made to resolve this tension; very few examples exist without evident compromise. At one end of the spectrum are the hardware-centric models, starting with machine code, then assembly code, and progressing into the realm of increasing abstraction with languages such as FORTRAN, COBOL, and C; these programming languages allow the expression of intention using constructs that translate very directly to the mechanics of the underlying machine, thus simplifying the potential for optimization of execution in terms of both time and space. As one proceeds along the language spectrum, abstractions emerge that are less and less a reflection of the underlying machinery, and more and more the attempt to realize concepts of ideal algorithmic and structural composition. These abstraction-centric models include object-oriented languages such as Smalltalk, Java, and C#, and functional languages such as Haskell, Erlang, and Clojure; these programming languages provide powerful abstractions that allow designs and concepts to translate very directly to the mechanics of the language, optimizing for some combination of simplicity, brevity, readability, consistency, conceptual purity, predictability of execution, code re-usability, portability, speed of development, and so on.

There is nothing more crucial in understanding a system of execution than understanding how the designers of the system chose to resolve this natural tension. The two questions that must always be answered are these: (i) How does the system propose to map its capabilities to the hardware instruction set, the hardware memory model, other hardware capabilities, and the operating system(s) within which it will execute; and (ii) how does the system propose to represent its capabilities to the software developer?

To that end, the following observations and analyses are made, in an attempt to capture both the purpose and the rationale behind the decisions that are made to resolve this fundamental tension in this particular system of execution.

## On Priorities

All designs have priorities, but only some designs begin with the end in mind. When priorities are not explicitly stated, it is easy to chase the priorities that most effectively combine compulsive emotional appeal with apparent ease of implementation, in lieu of the priorities that are most valuable to the intended audience of a design. In order to create a coherent design that serves the intended audience, this specification began with a conscious discussion about priorities, and a series of explicit decisions as to what those priorities would be:

1. Correctness, aka Predictability. The behavior of a language must be obvious, correct, and predictable. This also incorporates *The Principle of Least Surprise*.
2. Security. While generally not a priority for language design, it is self-evident that security is not something that one *adds to* a system; security is either in the foundation and the substrate of a system, or it does not exist. Specifically, *a language must not make possible the access to (or even make detectable the existence of) any resource that is not explicitly granted to the running software*.
3. Composability. High-level computer languages are about composition. Specifically, *a language should enable a developer to locate each piece of design and logic in its one best and natural place*.
4. Readability. Code is written once, and referenced many times. What we call “code” should be a thing of beauty, where *form follows function*.
5. Lastly, a language must be recursive *in its design*. There is no other mechanism that predictably folds in complexity and naturally enables encapsulation. *It’s turtles, the whole way down*.

## On Hierarchical Organization

Many a software developer has referenced this saying:

*When the only tool that you have is a hammer, every problem begins to look like a nail.*

That is not to imply that a hammer is not useful. If there is one conceptual hammer that – more so than any other – has repeatedly proven its merit for managing – and *hiding* – complexity, that would be the concept of *hierarchy*. File systems are hierarchies. B\*Trees and binary trees and Patricia tries and parse trees are hierarchies. Most documents are internally organized as hierarchies, including the common HTML, XML, and JSON formats. Most graphical user interfaces are modeled as hierarchies. Many programming languages leverage hierarchy to provide nesting, information hiding, scoping, and identity. How is it that such a simple concept can be so universally useful?

First of all, hierarchical organization enables very simple *navigation*. What this means is that at any arbitrary point – called a *node* – in a hierarchy, there is a well-known set of operations that are possible, such as navigating from the current node to its *parent* node, and navigating from the current node to any of its *child* nodes. If a node does not have a parent, then it is the *root* node, and if a node does not have any child nodes, then it is a *leaf* node.

Child nodes are *contained* within their parent node. Each child is uniquely identifiable by its parent, for example by a name or some other unique attribute. A hierarchy is *recursive*; at any point in the hierarchy, from that point down is itself a hierarchy. Since a hierarchy has a single root and is recursive, each node in the hierarchy is uniquely identifiable in the hierarchy by combining the identities of each successive node starting with the root and proceeding down to the node; this identity is the *absolute path* to that node. It is possible to navigate between any two nodes in the same hierarchy by combining zero or more child-to-parent navigations with zero or more uniquely identifiable parent-to-child navigations; the sequence of these steps is a *relative path* between two nodes.

These basic attributes of a hierarchy combine in amazingly powerful ways. For example, since each node is itself the beginning of a hierarchy of any size, it is possible to refer to that entire sub-hierarchy simply by referring to that one particular node; this effectively *hides* the recursive complexity contained – or *nested* – within that node. As a result, it is possible to add, copy, move, or remove a hierarchy of any size simply by adding, copying, moving, or removing the node that is the “root” of that hierarchy.

Using a hierarchy, it is incredibly simple to construct the concept of *scope*. For example, a scope could include only a specific node, or it could include a specific node and all of its child nodes recursively to its *descendent* leaf nodes, or it could include a specific node and its *ancestor* nodes to the root node, or any other combination of inclusion and exclusion that could be described in an unambiguous manner.

These concepts are incredibly simple, yet at the same time incredibly powerful, and are leveraged liberally throughout the XVM, from managing and hiding complexity for developers, to managing memory in an execution context.

## On Predictability vs. Performance

In the course of researching language design, one preterdominant concern emerges: That of execution performance. While many different concerns are evaluated and balanced against each other in a typical design, and while the goal of performance is often explicitly ranked in a position of lesser importance, in reality there is no one goal more considered, discussed, and pursued. Regardless of the importance assigned to performance as a language goal, performance to the language designer is a flame to a moth.

Perhaps it is simply best to admit, up front, that no language can survive – let alone succeed – without amazing feats of performance. Yet performance as a goal tends to conflict with other goals, particularly with respect to manageability, serviceability, and the quality of the abstractions that are presented to the developer.

Beginning programmers often ask: “Is language *A* faster than *B*?” After all, no one wants to be using a slow language, any more than someone would want to buy a slow automobile or a slow computer. Speed is a thrill, and speed in software execution holds no less attraction than a souped-up hot rod with a throaty growl and a body-rumbling ride.

The corollary to that question is “Why is language *B* slower than *A*?” The answers to this question tend to be very illuminating. Take any two languages that compile to and execute as native machine code, and compare their performance for a given task. Despite running on the same hardware, and using the same hardware instruction set, one may be dramatically slower than the other, by a factor of 10%, or 100% (half as fast), or even 1000% (an order of magnitude slower). How is such a difference possible?

The answer lies in translation, specifically in the automated translation of idioms. A language such as C selected idioms that closely represented the underlying machine instructions of the day, which allowed programmers to write simple programs that could be almost *transliterated* from C to machine code. In other words, the language chose as its abstractions the same set of abstractions that the CPU designers were using, or abstractions that were at most one translation removed from the machine’s abstractions. This allowed for very simple compilers, and made it extremely simple to support localized optimization.

A localized optimization is an optimization in the compiled code that can be made using only information about code that is *most local to* the code that is being optimized; in other words, information outside of the scope of the small amount of code being optimized is not even considered. Many optimizations in the C language, for example, are extremely local, such that they can be performed without any information about code outside of a particular expression or statement; it is hard to imagine more localized optimizations.

However, there is a trade-off implicit in achieving such simple and direct optimizations: The abstractions provided by the language are constrained by the abstractions provided by the CPU. As one could rightfully surmise, hardware abstractions tend not to be very abstract, and the abstractions provided by hardware instruction sets tend to be only slightly better. In its early days, the C language was jokingly referred to as “assembly with macros”, because as a language, it was only slightly higher level than assembly itself.

Computing efficiency is often stated in terms of a tradeoff between time (CPU) and space (memory); one can often utilize one in order to optimize for the other, subject to the law of diminishing returns. Unfortunately, there is no single simple metric that captures computing efficiency, but if the trade-off between time and space appeared as a graph with time on one axis and space on the other, it might generally resemble the shape of the curve *y=1/x*, which most closely approaches the origin at (1,1). If there were a single computing efficiency measurement for a programming language, it could arguably be represented by the closest that this trade-off curve approaches the origin (0,0), which distance could be considered the minimum weighted resource cost. To calculate a language’s efficiency for a particular purpose, one would calculate the *inverse* of the minimum weighted resource cost.

While one can easily speak about efficiency in hypothetical terms, a benchmark is a wonderful servant but a terrible master. The path chosen in the design of the XVM is to consciously avoid limits on potential efficiency by consciously avoiding contracts whose costs are not clearly defined and understood. This approach can be explained in the inverse, by way of example in existing languages and systems: Often, features and capabilities that were considered to be “easy” implementation-wise and “free” efficiency-wise when they were introduced, ultimately emerged as the nemesis to efficiency, due to the inflexibility of the programming contracts that these features and capabilities introduced[[2]](#footnote-2).

To understand this, it is important to think of abstractions as a two-sided coin: On one side, we see the benefit of the abstraction, which allows a programmer to work with ever-larger and more powerful building blocks, while the other side of the coin represents the cost of the abstraction, which is called the *contract*. Imagine, for example, having to build something in the physical world, out of actual matter. One could conceivably build a structure out of atoms themselves, assembling the necessary molecules and arranging them carefully into perfectly formed crystalline structures. The contracts of the various elements are fairly well understood, but yet we wouldn’t try to build a refrigerator of out individual atoms.

One could imagine that building from individual atoms is the equivalent of building software from individual machine instructions, in two different ways: First, that the refrigerator is composed of atoms, just like all software is executed at some level as machine instructions; and second, that as the size and complexity of the constituent components increase, the minutiae of the contracts of the sub-components must not be surfaced in the contracts of the resulting components – those details must be hidable! This purposeful prevention of the surfacing of minutiae is called *encapsulation*, and exists as one of the cornerstones of software design. It is why one can use a refrigerator without knowing the number of turns of wire in the cooling pump’s motor, and why one can use a software library without worrying about its impact on the FLAGS register of the CPU.

Ultimately, it is the recursive composition of software that creates challenges for optimization. While low level optimizations are focused on the creation of more efficient low level code, higher level optimizations rely on explicit knowledge of what portions of a component’s contract – or the contracts of the various sub-components – can be safely ignored. In other words, the optimizer must be able to identify which contractual effects are ignored or discarded by the programmer, and then leverage that information to find alternative execution solutions whose contracts manage to cover at least the non-discarded and non-ignored contract requirements. Higher-level optimizations target the elimination of entire aspects of carefully defined behavioral contracts, and as a result, they typically require extensive information from across the entire software system; in other words, high-level optimizations tend to be non-localized to the extreme! No software has been more instrumental in illustrating this concept than Java’s Hotspot virtual machine, whose capabilities include the inlining of potentially polymorphic code by the determination that the potential for dynamic dispatch is precluded, and the elimination of specific memory barriers in multi-threaded programs as the result of escape analysis.

To enable these types of future optimizations, the contracts of the system’s building blocks must be explicit, predictable, and purposefully constrained, which is what was meant by the goal of “consciously avoiding contracts whose costs are not clearly defined and understood.” The contracts in the small must be encapsulatable in the large, which is to say that contracts must be composable in such a way that side-effects are not inadvertently exposed. It has been posited[[3]](#footnote-3) that “all non-trivial abstractions, to some degree, are leaky,” but each such leak is eventually and necessarily realized as a limit to systemic efficiency.

## On God, Turtles, Balloons, and Sandboxes

Wikipedia defines a software sandbox as follows[[4]](#footnote-4):

In computer security, a sandbox is a security mechanism for separating running programs. It is often used to execute untested or untrusted programs or code, possibly from unverified or untrusted third parties, suppliers, users or websites, without risking harm to the host machine or operating system. A sandbox typically provides a tightly controlled set of resources for guest programs to run in, such as scratch space on disk and memory. Network access, the ability to inspect the host system or read from input devices are usually disallowed or heavily restricted.

In the sense of providing a highly controlled environment, sandboxes may be seen as a specific example of virtualization. Sandboxing is frequently used to test unverified programs that may contain a virus or other malicious code, without allowing the software to harm the host device.

In the physical world, in which children play with sand, there are two common styles of sandbox. The first is constructed from four equally sized wooden planks, each stood on its long edge to form a square box, fastened in the corners, and then filled with sand. The second style is typified by a large green plastic turtle, whose “turtle shell” is the removable top that keeps the rain out, and whose “body” is the hollow bowl that keeps the sand in. Both styles hold sand and allow a child to dig tunnels and build sand-castles, but there is one major difference: When a child tunnels too deeply in the wooden-sided sandbox, the tunnel burrows past the sand and into the soil beneath, while the tunnel depth in the turtle sandbox is strictly limited by the plastic bowl.

Software sandboxes tend to mirror these physical types, in that *the dirt often lies beneath*. In other words, the sandbox attempts to protect the resources of the system, but a determined programmer will eventually be able to find a way through. The only way that a language runtime as a sandbox can ensure the protection of the underlying resources of a system is for the sandbox itself to completely lack the ability to access those resources. Thus, the purpose of the sandbox is to defend against privilege escalation:

Privilege escalation is the act of exploiting a bug, design flaw or configuration oversight in an operating system or software application to gain elevated access to resources that are normally protected from an application or user. The result is that an application with more privileges than intended by the application developer or system administrator can perform unauthorized actions[[5]](#footnote-5).

As a language runtime designer, it is not sufficient to simply distrust the application code itself; one must distrust the entire graph of code that is reachable by the application code, including all third party libraries, including the language's own published runtime libraries, and including any internal libraries that come with the runtime that are accessible. Or, put another way, if there is a possible attack vector that is reachable, it will eventually be exploited. To truly seal the bottom of the sandbox, it is necessary to disallow resource access *through* the sandbox altogether, and to enforce that limit via transitive closure.

But what good is a language that lacks the ability to work with disks, file systems, networks, and network services? Such a language would be fairly worthless. Ecstasy addresses this requirement by employing *dependency injection*, which is a form of *inversion of control*. To comprehend this, it is important to imagine the container functionality not as a sandbox, but as a balloon, and our own universe as the primordial example.

Like an inflated balloon, the universe defines both a boundary and a set of contents. The boundary is defined not so much by a location, but rather by its impermeability – much like the bottom of the green plastic turtle sandbox. In other words, the content of the universe is fixed[[6]](#footnote-6), and nothing from within can escape, and nothing from without can enter. From the point of view of someone within our actual universe, such as you the reader, there is no boundary to the universe, and the universe is seemingly infinite.

However, from outside of the universe, the balloon barrier is quite observable, as is the creation and destruction of the balloon. Religiously speaking, one plays the part of God when inflating a balloon, with complete control over what goes through that one small – and controllable – opening of the balloon.

It is this opening through which dependency injection of resources can occur. When an application needs access to a file system, for example, it supplicates the future creator of its universe by enumerating its requirement as part of its application definition. These requirements are collected by the compiler and stored in the resulting application binary; any attempt to create a container for the application will require a file system resource to be provided.

And there are two ways in which such a resource can be obtained. First of all, the resource is defined by its interface, so any implementation of that interface, such as a *mock* file system or a fully emulated – yet completely fake! – file system would do. The second way that the resource can be obtained is for the code that is creating the container to have asked for it in the same manner – to declare a dependency on that resource, and in doing so, force its own unknown creator to provide the filing system as an answer to prayer.

As the saying goes, it’s turtles all the way down. In this case, the outermost container to be created is the root of the container hierarchy, which means that if it requires a filing system, then the language runtime must inject something that provides the interface of a filing system, and that resource that is injected might even be a representation of the actual filing system available to the operating system process that is hosting the language runtime.

And here we have a seemingly obvious contradiction: What is the difference between a language that attempts to protect resources by hiding them at the bottom of a sandbox container, versus a language that provides access to those same resources by injecting them into a container? There are several differences, but let’s start with an obvious truth: Perfection in the design of security is difficult to achieve, and even harder to prove the correctness of, so it is important to understand that this design does not itself guarantee security. Rather, this design seeks to guarantee that only one opening in the balloon – and anything that is injected through that opening – needs to be protected, and the reason is self-evident: Transitive closure. By having nothing naturally occurring in the language runtime that represents an external resource, there is simply no surface area within the language runtime – other than the injected dependencies themselves – that is attackable.

Second, the separation of interface and implementation in the XVM means that the implementation of the resource is not visible within the container into which it is injected. While this pre-introduces a number of language and runtime concepts, the container implementation only makes visible the surface area of the resource injection *interface* – not of the implementation! This holds true even with introspection, and furthermore the injected resources are required to be either fully immutable, or completely independent services.

Third, this design precludes the possibility of native code within an Ecstasy application; native functionality can only exist outside of the outermost container and thus outside of the language runtime itself, and can only be exposed within the language runtime via a resource injected into a container, subject to all of the constraints already discussed.

Lastly, as has been described already, the functionality that is injected is completely within the control of the injector, allowing the requested functionality to be constrained in any arbitrary manner that the injector deems appropriate.

While it is possible to introduce security bugs via injection, the purpose of this design is to minimize the scope of potential security bugs to the design of the relatively small number of interfaces that will be supported for resource injection, and to the various injectable implementations of those interfaces.

## On Processor Performance

There exists no shortage of opinions on the topic of what aspects are the most important in a system of execution. One voice will claim that only performance matters, while another will suggest that it no longer matters at all. One voice will claim that achieving efficiencies in development is far more valuable, while another will insist that predictability and stability in execution is critical. Opinions morph with time, as the reality of the physical units of execution evolves and the conceptual units of design are ever the more realized in languages and libraries.

Nonetheless the state of the art today bears the hallmark of a path followed far beyond its logical conclusion. In 1977, John Backus raised an early warning in his ACM Turing Award lecture:

Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it.

While programming advances have largely digested and expelled the explicit concerns of store-addressing and word-at-a-time thinking, these advances have been repetitively accreted onto a burial mound whose foundation remains a von Neumann architecture. Perhaps the success of that underlying architecture is the result of natural selection, or perhaps we have only inertia to blame. In any case, the evolution of concurrent multi-processing and distributed systems has stretched the von Neumann architecture past its effective limits. Specifically, it appears that the recent growth in the extent of the now automatically-managed store has occurred at a pace well beyond the increase in performance of the heart of the von Neumann machine: the processor. Whether this imbalance can be rectified by further technological accretion or by the adoption of a fundamentally new execution architecture is yet to be seen, but regardless: The inevitable and predictable increase in performance that has become the opiate of an industry has taken a sabbatical, and may have accepted an early retirement altogether.

There has existed a loose historic alignment in the growth of processor performance, memory capacity, memory throughput, durable storage capacity, durable storage throughput and network throughput. This relatively consistent growth has allowed a general model of assumptions to be perpetuated throughout hardware architectures, operating systems, programming languages and the various resulting systems of execution. Now we find that model to be threatened by the failed assumption that processor performance will increase at a rapid and relatively predictable rate.

To maintain the façade of progress, explicit hardware parallelism has emerged as the dominant trend in increasing processor throughput. Symmetric Multi-Processing (SMP) has a relatively long history in multi-CPU systems, but adoption of those systems was hobbled both by high prices and a lack of general software support. In the 1990s, the advent of the World Wide Web propelled multi-CPU systems into the mainstream for servers, but it is the recent, seemingly instantaneous and near-universal commoditization of multi-core CPUs that has finalized the dramatic shift from a focus on processor performance to a focus on processor parallelism. Further compounding the adoption of multi-CPU and multi-core systems are various technologies for Concurrent Multi-Threading (CMT), which enables a single CPU core to execute multiple threads concurrently. In aggregate, since the turn of the millennium, parallelism has increased from one to dozens of concurrently executing threads in an entry-level server, while the performance of an individual processing unit has only increased by only a few times. Looking forward, processor performance is now expected to improve only incrementally, while the level of parallelism is continuing on an exponential curve.

Since overall processing throughput has continued to increase at a dramatic pace not dissimilar from its historic trend, this shift from performance to parallelism could be safely ignored but for one problem: The von Neumann architecture is bound to a individual processing unit, and thus has nearly halted its forward progress in terms of the throughput of a single thread of execution. This means that for the first time in software history, existing programs do not run significantly faster on newer generations of hardware unless they were built to explicitly take advantage of thread parallelism, which is to say unless they were built assuming their execution would occur on multiple von Neumann machines in parallel. Since the art of programming is expressed almost entirely in imperative terms, and since the imperative nature of programming languages is based on the von Neumann architecture, we have managed to accumulate generations of programs and programmers that are hard-wired to a model that has at least temporarily halted its forward progress.

As a result, computing devices are providing increases in processing throughput that can only be consumed by parallelism. It is obvious that this mandates support for parallelism in any new system of execution, but there is a far less obvious implication of critical importance. Parallelism increases throughput only to the extent that coordination is not required among the threads of execution, and coordination is required only for mutable resources that have the potential to be shared across multiple threads of execution. In common modern systems of execution such as Java and C#, explicit parallelism is provided by threads of execution, each representing the state of a single von Neumann machine, but those machines collectively share a single store. Compounding the coordination overhead for the store is the prevalence of automatic management of the store, referred to as Garbage Collection (GC), which unavoidably requires some level of coordination. While GC algorithms have advanced dramatically in terms of parallelism, the remaining non-parallelized (~~and possibly non-parallelizable~~[[7]](#footnote-7)) portion of GC is executed as if by a single thread, which is to say that the runtime behavior of these systems will periodically halt in order to garbage-collect the shared store. The unavoidable conclusion is that growth in the shared store without the corresponding increase in processor performance has lead to unavoidable and growing pauses in the execution of the parallelized von Neumann machines.

A series of advances in GC algorithms have thus far masked this inevitable consequence, but the advances are already showing diminishing returns, while the upward pressure on the size of the store has not abated and the dramatic progress of processor performance has not resumed. Ultimately, a single shared mutable store must be avoided, and the design of a runtime system must reflect that requirement. The XVM design is intended to fully address this challenge, and does so by decomposing the problem space along a number of naturally occurring fault lines. First, the XVM organizes scopes of linking, loading, and execution – referred to as *containers* – in a hierarchical manner; second, within that hierarchy, scope of execution is further localized into individual, conceptually-independent von Neumann machines – referred to as *services* – within which all allocations occur; and third, that only immutable state can *escape* the execution scope of a service.

These decisions allow dynamic memory allocation to be managed within (scoped to) a particular service, and the resulting garbage collection to be performed entirely within the context of individual services, without requiring the coordination of other services or containers. The two exceptions to this are escaped immutable data, and the reclamation of services and containers themselves. In the case of escaped immutable data, and precisely because the data (the objects) are immutable, the memory can be garbage collected without any coordinated halt of execution[[8]](#footnote-8). Furthermore, the escaped immutable data can be organized within the container hierarchy at the level to which the data has escaped, or alternatively can be managed in a single global immutable store.

In the case of containers and services, each mark phase of each actively executing service also marks any services that it in turn can reach, again without any coordinated halt of execution; unreachable services are then collected asynchronously.

The concept of localizing allocations in order to localize GC is not new. Systems built around an explicit threading model have employed *escape analysis* in order to determine which allocations can safely be performed using a thread local allocator (such as a slab allocator), and which allocations need to be made from a shared store. This represents a hierarchy with a fixed depth of two: Global and thread local. While a dramatic improvement over a single shared store, it still implies a global stoppage for GC execution of the shared store.

The primary benefit to GC of the localization of the store is that a significant portion of overall GC execution can be localized entirely within each thread of execution, and further localized within the bounds of a particular service. Additionally, having stores that are localized to each service enables the system to exactly measure and meter – in real time – the amount of memory that is consumed by each service, and in aggregate by each container. Lastly, a range of optimizations are available to the data managed in a thread-localized store: the memory containing the data can be allocated, accessed, manipulated, and freed efficiently without any hardware-level concurrency control, and generated native code can be optimized specifically for cache locality.

Second, by organizing memory hierarchically in a manner corresponding the runtime container model, a service or an entire container can be discarded in its entirety, because no objects outside of that hierarchical scope can have a reference into the memory managed within that scope. In other words, the cost of “yanking” an entire sub-portion of the hierarchical runtime system is nominal, and the results are deterministic.

An additional benefit is that machine code optimized for single-threaded execution has dramatic performance advantages compared to machine code that is concurrency-safe, even when concurrency control is optimized using the latest hardware capabilities such as compare-and-swap (CAS) and other “lockless” instructions. Mutable data structures that are localized to a single thread of execution allow the resulting execution to approach the theoretical maximum efficiency of the underlying hardware.

The concept of GC optimizations based on immutability is also not new. Several GC implementations have leveraged memory protection barriers to protect memory regions being compacted as if the data were immutable, allowing application execution to proceed (in the absence of a fault) with the GC operating concurrently. Significantly, explicitly immutable data can be compacted without protection, because both the old (pre-compaction) and new (post-compaction) copies of the data are valid – being identical! – thus enabling the application to continue to execute concurrently and correctly while referring to either copy arbitrarily, deferring the housekeeping task of updating any pointers that point to the lingering old copy, and deferring the recycling of the memory region that was compacted. As an added benefit, the GC of regions of data known to be immutable can be performed by any thread of execution, or even a separate thread dedicated to GC.

## On Immutability

In an object-oriented system, immutability refers to the prohibition to alter the state of an object. It turns out that many data types are naturally immutable; consider the number 42 for example[[9]](#footnote-9) – it is always the number 42! Other data types are naturally mutable; consider the concept of a *variable* for example – its very purpose is to be able to vary! Many data types are naturally immutable, and even with mutable data types, it is often desirable to be able to make specific instances immutable on demand.

The XVM explicitly supports immutability. Immutability has several benefits, notably: Predictability, thread/concurrency safety, security, and available optimizations. Predictability is one of the greatest benefits of good design, and immutability supports predictability by providing data types that are truly constant – *at runtime!* For example, when an object exposes its state, it often does so by exposing immutable data types so that its internal state cannot be directly altered; without explicit support for immutability, one of two things could occur: Either the mutable state of the object would be exposed, breaking encapsulation, or a copy (or other representation) of the mutable state would need to be created on demand to safely expose that state, which is expensive in terms of both space and time, not to mention complexity. Immutability provides a simple way to ensure that the state of an object *cannot* change, addressing each of these concerns.

With respect to thread and concurrency safety, an immutable object provides the same state regardless of the concurrency of access to the object, because changes to the state of the object are explicitly prohibited. It is precisely because of this explicit contract that the only state that can be visible to more than one thread of execution in the XVM is immutable state; immutable objects can be safely used without concurrency control and without relying on memory barriers.

Using immutability for security is powerful, but it is important to understand that security as a topic is simply another facet of predictability, and as a result, the same concepts and conclusions apply. Specifically, when immutability is an intrinsic axiom of a system, and thus cannot be circumvented, it becomes a powerful tool for reasoning about how aspects of the system will operate, and those assumptions become trusted building blocks for building secure software.

Regarding optimizations, the explicit knowledge of immutability conveys a number of significant advantages to a system of execution. As described previously, for example, explicitly immutable data enables a number of potential optimizations for the purpose of garbage collection, and immutability obviates the need for the types of concurrency control used with mutable objects. Immutability also supports both pass-by-reference and pass-by-value semantics *transparently*, because the underlying value is itself immutable, and thus the substitution of any duplicate copy of the value has no behavioral consequence. (Put another way: In the XVM, one cannot determine whether an object is being passed by reference or by value *if that object is immutable*.)

In addition to object immutability supported by the language runtime, there are two related XVM capabilities worth enumerating. The first is support for lazily initialized state as part of an otherwise-immutable data structure, specifically, by means of a function with presumed-idempotent behavior. Such a capability allows the evaluation of time-expensive computations and/or the allocation of space-expensive data structures to be deferred until actually requested. One common example is the hash function calculation for complex data types, which is often assumed to be expensive enough to defer, but whose immutable value, once computed, should be saved for subsequent use.

The second capability is language-level support for designing a non-mutating reference to an underlying data structure that may itself be mutable. In other words, it is desirable to be able to support multiple references to the same object, such that a reference explicitly omits mutating operations and the exposure of mutable state. There are many examples of mutable data that must be generally protected from mutation, but which the owner of the data may need to be able to mutate; in this case, the reference through which mutations are prohibited is called a *read-only* reference. This capability does not rely on immutability, but rather relies on the careful design of a programmer, and as a result does not provide the types of guarantees that actual immutability can confer. However, with careful use, it is a useful tool for selectively hiding mutability.

## On References

The family of languages influenced by C++ share an implicit trait: A compile-time knowledge of accessibility. For example, it is the C++ compiler that prevents access to private members, and the compiler that allows access for a friend. Subsequent languages, like Java, built on this model, and carry sufficient information in the compiled form of the language to allow the accessibility implied by the compiled code to be validated for a second time at runtime, to prevent an end-run around the language’s security features. One by-product of this design is the ability to use a single pointer to reference an object, and more specifically, a pointer that is – in C++ parlance – a Vtable\*\* (a pointer to a pointer to a virtual function table).

In the simplest terms[[10]](#footnote-10), one can imagine an object as the combination of a structure (such as a C struct) and functions that operate on that structure. For each particular type (known as a class), the Vtable\*\* approach arranges those function pointers in an array, and orders those function pointers in the array to be consistent with other compatible types, thus supporting the polymorphic behavior of compatible types (via the common sub-sections of those arrays). A pointer to the array of functions is stored in the first field of the struct, which means that a pointer to the struct will also act as a pointer to the functions associated with that struct (hence the object pointer being a Vtable\*\*). Invocation of any of those functions requires that a pointer to the struct (which pointer is named this) be passed to the function, thereby allowing the function to access and modify fields in the struct, and allowing it to invoke other functions against that same struct, all without statically binding to either the address of the struct or to the addresses of the other functions related to the struct.

From a mechanical-simplicity and efficiency standpoint, the benefits of this model are difficult to overstate; it is powerful, and it is elegant. However, there are several specific costs to account for as well. First, the type system defines accessibility in a static manner, predicated on the class of the referrer and the class of the referent, and how those two classes relate. Consider C++ protected members, which are accessible to sub-classes, or Java package-private members, which are accessible to any class within the same package (a hierarchical name-space). Second, the facets that a class exposes are statically fixed, such as the set of members declared as public, allowing any referrer that obtains a reference to exploit runtime capabilities such as Java’s *reflection* to fully examine public members, access any public data fields, and invoke any public methods. While allowing arbitrary access to public members does not seem at first to be a concern, it can be; consider that members of an interface are always public in an implementing class, which leads to the third issue: It is not possible for an object to selectively expose its members, for example providing different interfaces to different referrers.

This lack of selective exposure can lead to complexity, particularly when the goals of rich functionality and security collide. One need look no further than Java’s serialization capabilities, in which an object must be able to expose its state to a serialization mechanism, but that same state – often private in nature – must be protected from all other referrers. The solution to conflicting goals is to create exceptions to the rules. Unfortunately, each exception to a rule contributes to the complexity of the system, and faults (vulnerabilities) inevitably emerge from complexity.

Ecstasy employs a different model entirely, by *conceptually* composing each reference from two pieces of information: An object’s identity, and the type (or interface) exposed through the reference. This approach is a synthesis of several concepts:

* A *type* is defined not by its name nor by its implementation, but solely by its member capabilities, exposed as a set of methods and properties;
* A *reference* encodes both the capabilities that are selectively permitted, and the object identity against which those capabilities may be invoked;
* While each class of objects has a number of predefined sets of capabilities, such as public, protected, and private interfaces, a set can also be arbitrarily composed;
* The permission to access the capabilities of an object is part of (as if *encoded in* and *defined by*) the reference information itself, and is neither the result of static compile-time checks nor dynamic security checks;
* It is always permissible to reduce (to *mask*) the type of a reference by removing properties and/or methods, but the opposite (to *reveal*) is forbidden, except *at or above* the point in the container hierarchy where the object’s class was loaded (similar to the *ring model* in a CPU).

There are several effects of this design decision:

* Any number of different references can exist for the same underlying object, each of which can have a different set of exposed methods and properties;
* There are cases in which it will be possible for the XVM to provide a reference to a purely derivative object of an existing object *without actually allocating a derivative object*, by instead creating a references whose type is the type of the derivative object, but whose implementation is modified to rely on the original object. For example, the same *identity* can be used to represent a particular object *and* an array of objects containing only that particular object, by encoding that same identity with two different *types* into two different references[[11]](#footnote-11).

Lastly, the ability to *mask* and *reveal* portions of functionality in a reference is a fundamental security mechanism. When a reference from outside of a container is injected into that container, that reference is *masked* as the injection type (such that only the methods declared on the interface being injected will be accessible), and within that container, it is then impossible to *reveal* additional methods on that reference. However, within a container, for classes defined within that container, it is always possible to reveal additional methods on references to objects created within that container. In other words, modules loaded within a container are not protected from other modules loaded *within the same container*, and objects instantiated within a container are not protected from other objects instantiated *within the same container*. What does this mean in practice? Among other things, it means that within the container that defines a class and instantiates an object of that class, terms like private and protected are useful only for hiding complexity, not for securing the contents of an object.

## On Portability

Similar to the goals of the Java Virtual Machine specification, portability is a primary consideration for the XVM. Specifically, the XVM design targets the efficient implementation on three major instruction sets (x86/x64, ARM A32/A64, and wasm[[12]](#footnote-12)) and five major operating systems (Android, iOS, Linux, macOS, Windows); it is assumed that support for any other modern environment is practical as a result.

|  |  |  |
| --- | --- | --- |
| Instruction Set | Operating System | Example |
| ARM | Android | Phone, tablet, desktop, kiosk, embedded device |
| ARM | iOS | iPhone, iPad, Apple TV |
| x86/x64 | Linux macOS Windows | Desktop  Server |
| wasm | \* | Browser |

The XVM specification defines a portable binary (aka *byte code*), and the behavioral requirements to execute such a binary. The XVM is not constructed around nor constrained by a particular word size, pointer size, endian byte ordering, or signed/unsigned support of the host CPU; all encodings in the portable binary are either octet (8-bit byte) or variable-length integers.

The portable binary was designed as an input for native compilation, such as a Just In Time (JIT) compiler. Additionally, care was taken in the design of the XVM to ensure that it allowed for Ahead Of Time (AOT) compilation. One explicit non-requirement of the portable binary is efficient interpretation; while an XVM interpreter is possible (one was implemented as a proof-of-concept during development of the XVM specification), the design of the portable format does not lend itself to efficient interpretation.

There is one root[[13]](#footnote-13) module, the Ecstasy.xtclang.org module, which provides the fundamental building blocks of the type system; all XVM types are derived from the contents of this root module. Furthermore, a module cannot contain native (external) code, so the root module contains a complete implementation of the types contained therein. This implies a completely self-referencing type system composition, which while a correct assumption, is also an impossible conclusion[[14]](#footnote-14).

While the root module itself is identical (portable) across hardware and software platforms, the implementation of the XVM varies – potentially dramatically – across the same. The XVM is responsible for replacing select portions of the self-referential implementation in the root module with native implementations that support the specified contracts; however, *which* portions get replaced will vary depending on a combination of the goals of a particular XVM implementation and the capabilities of the hardware and software combination that the XVM is targeting. Furthermore, *which* portions get replaced is (and *should be*) unknown by a programmer working in Ecstasy.

On one hand, it is highly desirable to be able to provide a truly minimal set of native implementations in the execution system itself, and to be able to implement all other data types and operations by composing from that minimal set. This approach permits a minimalistic implementation of the XVM. Similarly, it is highly desirable to be able to provide a richer set of native implementations as part of the execution system itself, in order to leverage additional capabilities provided by the runtime environment, the operating system, and the underlying hardware. In other words, the ability to trade off complexity, size, and performance is based on the ability to define standardized capabilities as abstract data types that can be implemented either as an native part of the execution system or in software that is executed by the execution system. While this design aspect is visible in as few places as possible, it deeply permeates the design of the XVM specification and the root module.

To accomplish this goal, the root module provides a rich set of abstract types, defined as part of the XVM specification, and a reference implementation for all aspects of those types is provided. It is *possible* to create one XVM implementation that relies *almost* entirely on the reference implementation of these types (thus requiring only a nominal set of operations to be implemented in native code), and another XVM implementation that directly implements a dramatically larger portion of the root module in native code (hopefully achieving higher performance with less memory usage).

With the advent of the CLI specification, best known in the Microsoft .NET CLR incarnation, and to a lesser extent in the open source Mono project, a new facet was added to the concept of virtual machine portability: Explicit multiple-language support. There are inevitable trade-offs in creating an execution system that is intended to support multiple languages; generally, an execution system is optimized for a specific language, and the implementation of any other language trades off between execution performance and the adherence to the “foreign” language specification.

One can conceivably implement any programming language on any Turing-complete execution system, resorting to full runtime interpretation if necessary; the obvious trade-off is correctness versus space/time efficiency. With respect to multiple language support, the primary goal of the XVM is to enable the efficient implementation of a *specific class of language* that can be described by its attributes: modular, object-oriented, composable, single-dispatch, immutable message passing, concurrent, thread-safe, with automatic memory management. Alternatively, the class of language could be described as an incremental evolution of the Java and C# languages, with support for type composition, a reified generic type system with transitive closure, a single (object) type system, and a recursively consistent (hierarchical) runtime model, while explicitly eschewing support for concurrent access to mutable data. While it is desirable that other classes of languages be efficiently implementable on the XVM, it is an explicit non-requirement that the semantics of those languages be intrinsically supported by the XVM where those semantics differ from those of the targeted specific class of language. Or, more succinctly: The XVM is designed to run the Ecstasy language, and its design allows for relatively efficient execution of languages with similar runtime models, but its design contains nothing that explicitly supports languages other than Ecstasy.

## On Startup Time

One of the lesser design goals for the XVM is to be able to achieve near instantaneous startup time. As simple as this goal sounds, the complexity of the type system works against it.

Consider a side effect of the core JVM type system being largely implemented on top of the JVM: The large number of classes required for the simplest “hello world” application. Just as in the children’s song[[15]](#footnote-15) that iterates over the body a bit at a time, like “*the knee bone’s connected to the shin bone*” and so on, the JVM is forced to load a relatively large graph of its type system as a result of the dependencies among its most basic types. Loading a type likely requires I/O, parsing, validation, internal type definition, the execution of any specific type initialization code, and the recursive loading of types on which the loaded type depends. While loading a relatively large graph of a type system is acceptable for a long-running process (which can amortize that loading cost over hours, days or months), it is far more intrusive a cost for a short-running process.

The XVM design further exacerbates this condition by guaranteeing – with transitive closure – the validation of an entire type system before it is activated within a runtime container. That means that the entire type system must be loaded and evaluated before the first line of code can be executed.

To address this challenge, several aspects of the XVM design are relevant:

* Within a given container, the type system of that container is immutable; loading the type system within a container thus always results in an identical type system;
* The elimination of global data structures (and in particular, mutable data structures) dramatically simplifies coupling across the type system, simplifying initialization;
* Dependency injection largely eliminates lazily-initialized language sub-systems, such as I/O;
* AOT compilation allows a container’s type system, or even an entire application, to be compiled to executable code and related structures in advance; and
* Explicit version support in the portable binary allows the runtime to detect changes that violate any of the above assumptions.

The design is intended to allow an XVM implementation to instantiate a new container with a specified set of modules, with a cost that is *on the same order of magnitude* as loading and initializing a shared library in a running C/C++ application.

## On Openness

Undoubtedly, the largest change in software in the past two decades has been cultural and not technological. The term “open” had long been used for marketing commercial software that had some slight yet often only theoretical potential for interoperability with other commercial software. Today, most core software components, libraries, operating systems – and even many applications – are available in complete source code form for use under an *open source* or *software libré* license, and many of the specifications and standards – including languages and execution systems – that enable interoperability are similarly open and available. From an economic standpoint, it would appear that the demand for a fundamental set of software standards and components being available as a public good now out-weighs the cost of creating and managing that public good (even in some cases lacking any consistent centralized authority!), and the cost of reverting to private goods for that fundamental set of software standards and components is unacceptable for all but the most especial of requirements.

It is in this spirit that the XVM specification is made available, with its ideas and concepts inspired from others’ open work, and – if any prove worthwhile – its ideas and concepts freely available for re-use and recycling as the reader sees fit.

## Credits

None of these ideas occurred in a vacuum. The initial inspiration for this effort was the Java Virtual Machine, which was groundbreaking in its timely fusion of brilliance and pragmatism, and which is described in beautiful detail by the Java Virtual Machine Specification and the related Java Language Specification. Older languages such as C, C++, Python, and Erlang left their own indentation on our thinking as well. Many concepts of the XVM will be familiar to Smalltalk programmers, who to this day righteously espouse that all subsequent programming advances were already present in and perfected by Smalltalk, which unfortunately met its untimely demise as the result of a vast right-wing conspiracy. More recently, languages such as C#, Ruby, Scala, Go, Ceylon, and Kotlin have stretched our imaginations, each with their own pragmatic preview of a potential programming future, and with systems of execution such as the Sun Hotspot JVM, Microsoft CLR, Google V8, and LLVM managing to occasionally exceed even the scope of imagination with their practical ingenuity.

# Background

Ecstasy and the XVM exist in a technology field filled with existing knowledge an experience. It was our intention, in designing Ecstasy, to leverage as many existing concepts and design idioms as we could from the most-used languages and technologies, so that someone with existing knowledge in field would feel immediately at home both reading and writing in the Ecstasy language.

This section attempts to correlate technical terms and concepts used in the Ecstasy langue and the XVM with similar terms and concepts that are already broadly used in the field.

## A Brief Introduction to Object-Oriented Programming

A number of languages are described as Object-Oriented, which means that the language is intended to allow the structure of a program to be neatly divided into discrete units that each define a specific portion of the program’s data and the logic that is closely associated with that data; these units are referred to as *classes*. To explain some terms courtesy of an overused analogy, consider a dog named Spot and a cat named Fluffy: Spot is an object whose class is dog, while Fluffy is an object whose class is cat. The term *class* represents the definition of the thing, while the term *object* represents one of those things; sometimes the term *instance* is used, as in: Spot is *an instance of* dog.

Now that the obligatory animal analogy has been made, let us instead consider a graphics system on a device or computer, starting with the concept of color. One representation of color is called RGB, in which three separate numeric values for red, green, and blue are mathematically combined into a single numeric value that represents the color; perhaps the simplest implementation of RGB values represents the three color channels as 8-bit integers, and the combined RGB value as a 24-bit integer. In an object-oriented language, one can define a class named Color whose properties are Red, Green, Blue, and RGB. The notion of *encapsulation* allows the detail of how that information is stored inside a Color object to be hidden; for example, a Color could have three separate integers that it stores its red, green, and blue information in, or it could have one integer that it stores the combined RGB value in, or it could use some other representation of the data altogether. Regardless, from any point of view outside of the Color class, the *state* of a color object is easily represented by the *properties* of the object in a manner that is consistent with the description of what a color is, such that values of the Red, Green, Blue, and RBG properties are consistent.

Another valuable aspect of object-oriented programming is *polymorphism*, which is when different classes of objects share some common programming *interface*, and thus are substitutable to some degree. Extending our previous example, a second class representing color could be created in a manner that would be substitutable in some situations for the first, by providing the same programming interface as the Color class provides, but potentially with a different data structure and implementation. For example, some JPEG and video formats use the YCbCr representation of color that is composed of a luma value (Y), and chroma values for both blue difference (CB) and red difference (CR). Since a YCbCr value can be converted to and from an RGB value, a second Color class (perhaps named YCbCrColor) could be defined that provides properties of Luma, BlueDiff, and RedDiff, but also provides properties of Red, Green, Blue, and RGB – just like the original Color class. Again, as with the previous example, encapsulation allows the class to hide the specific manner in which the object manages its internal information (also called its *state*) behind its programming interface, while exposing that information as both an YCbCr value *and* as an RGB value. Polymorphism allows an instance of the new YCbCrColor class to be used where the program was expecting an instance of the original Color class.

One way in which the YCbCrColor class could have been defined is through the use of *inheritance*, which is when a new class is defined by starting with the definition of an existing class, and then adding things to it. Inheritance is quite powerful, particularly because it allows the re-use of existing class definitions, and because it provides a natural means of supporting polymorphism. In our example, if YCbCrColor is inherited from Color, then it would automatically inherit the entire programming interface of Color, and an instance of YCbCrColor could be transparently used in place of an instance of Color, because YCbCrColor *is a* Color. However, as powerful as inheritance may seem, it has often been perceived to be overused and abused as a tool; as a result, developers have started to “favor object composition over class inheritance”[[16]](#footnote-16).

While inheritance provides an “*is a*” relationship, composition provides a “*has a*” relationship. For example, instead of the YCbCrColor class inheriting from the Color class, it could simply have a property of the Color class; in other words, the programming interface for YCbCrColor could include a way to obtain a separate object that *is a* Color value, just like it includes a way to obtain Luma, BlueDiff, and RedDiff values.

A third option is to use interface inheritance, which allows the programming interface from an existing class to be re-used while not inheriting other aspects of the existing class definition, such as the manner in which data is structured and managed. For example, a programming interface called Color could be defined that has a single property RGB, whose value is in the form used natively by the device, such as a 24- or 32-bit integer or floating point value. Any number of class definitions could then *implement* the Color interface, thus allowing instances of those various classes to be utilized any time a Color is required. For example, the Color class from our original example could be renamed as RGBColor, and both it and the YCbCrColor class could be made to implement the new Color interface. Similarly, the interface could be implemented by additional class definitions, such as a YUVColor class (the color value used in PAL televisions), and a YIQColor class (the color value used in NTSC televisions).

Conceptually speaking, a programming interface is composed of the accessible *properties* (state) of a class and the *methods* (behavior) that the class exposes; collectively, these are the *members* of the interface. Some programming languages allow any class to act as a substitute for another class, as long as it has the same members as the class that it is substituting for; this is an example of *dynamic typing*, and is often specifically referred to as *duck typing* because “if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.” Other languages require that a class explicitly declare the names of the programming interfaces for which it can act as a substitute; generally speaking, this is an example of *static typing*. However, it is important not to approach terms such as static and dynamic typing too rigidly, because these concepts represent a continuum of possibilities, as opposed to some single binary choice. Furthermore, every programming language has many different aspects, each of which can be located at a completely different point on that continuum.

In object oriented programming, each class is a specific data type; in our examples, each of the various color-related classes and programming interfaces is its own separate data type. When a programmer wishes to create a class (e.g. a Array) that will manage, or contain, or operate upon some unspecified second class (e.g. the class of things that will go into the Array), that second class can be represented by a place-holder, which is called a *type parameter* (e.g. an ElementType). A class that has one or more type parameters is called a *parameterized type*, or simply a *generic type*. We have already discussed “*is a*” and “*has a*” relationships; a generic type supports an “*of a*” relationship. Consider the array, which is perhaps the most basic of data structures, supported even by assembly languages. An array is a contiguous sequence of a specific number of elements of a specific type, each of which can be loaded from or stored into the array using the element’s index, which is an absolute position in the array.

While static typing makes it possible to specify that one must pass a single Color to a particular method, consider a method that accepts multiple Color objects to be passed to it in an array. Without generic types, the programmer would have to define the method to accept *any* Array (whose contents are of an unspecified type), or alternatively could create a specific type – perhaps ArrayOfColors – that can hold only colors. A generic type is an alternative to creating a separate specific type (such as ArrayOfColors) for each possible “*of a*” relationship; instead, a single type (Array) is defined with a place-holder, such as Array<ElementType> which uses ElementType as a place-holder for some unspecified type. Using a generic type would allow the method in our example to be defined to accept a parameter of type Array<Color>, which is read as “an Array of Color”. The benefit of generic types is the extension of type safety (and any other facilities that rely on or can benefit from type knowledge) to any classes that support one or more “*of a*” relationships. This is also referred to as *parametric polymorphism*, which refers to both the *parameterized type* aspect, and polymorphism, which is a direct benefit of using parameterized types (generic types) as the basis for generic programming.

One of the valuable capabilities provided by inheritance, and perhaps one of the causes for the overuse of inheritance, is the ability to define state and behavior in one class that is then inherited by many different classes. Elimination of redundant code (*a la* “cut and paste”) is a noble goal indeed, and is at the core of the “Don’t Repeat Yourself” (DRY) principle[[17]](#footnote-17), which states: “Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.” In addition to inheritance, you can see this principle at work in programming interface definitions, polymorphism, and generic types. The concept is well described by Steve Smith[[18]](#footnote-18):

Of all the principles of programming, Don't Repeat Yourself (DRY) is perhaps one of the most fundamental. The developer who learns to recognize duplication, and understands how to eliminate it through appropriate practice and proper abstraction, can produce much cleaner code than one who continuously infects the application with unnecessary repetition.

Every line of code that goes into an application must be maintained, and is a potential source of future bugs. Duplication needlessly bloats the code-base, resulting in more opportunities for bugs and adding accidental complexity to the system. The bloat that duplication adds to the system also makes it more difficult for developers working with the system to fully understand the entire system, or to be certain that changes made in one location do not also need to be made in other places that duplicate the logic they are working on.

Repetition in logic can take many forms. Copy-and-paste[..] is among the easiest to detect and correct. Many design patterns have the explicit goal of reducing or eliminating duplication in logic within an application. In fact, the formulation of design patterns themselves is an attempt to reduce the duplication of effort required to solve common problems and discuss such solutions.

A powerful mechanism for achieving DRY in object-oriented programming is the *mix-in* (or *mixin*), which in many ways is like a class definition, but one that can be “mixed into” (i.e. “added to”, “applied to”, or “incorporated into”) other class definitions. Using the concepts of generic programming, a mixin *applies to* some yet-to-be-specified second class that provides a specified programming interface, allowing the logic of the mixin to be constructed around the abilities of that programming interface. Then, any time that mixin’s logic is desired in another class, the mixin can be applied to that class; while various other mechanisms provide “*is a*”, “*has a*”, and “*of a*” relationships, a mix-in satisfies the “*needs a*” relationship. Continuing our earlier color example, some graphics systems support the notion of color transparency, which is sometimes expressed in the range of 0% to 100%, but is often encoded as an 8-bit value prepended to a 24-bit RGB value, creating a 32-bit integer value. A ColorTransparency mix-in could be defined that applies to any Color, adding a Transparency property that holds a value representing the transparency level, and overriding the RGB property to incorporate the transparency information into the resulting value.

This “brief” introduction was not intended to be an exhaustive explanation of object oriented programming; hopefully, it has introduced a number of terms and concepts, and with enough context to provide a basis for understanding some of the constructs in the Ecstasy language.

In the end, these concepts are simply mechanisms for organizing functionality together into logical units, in manners designed explicitly to minimize redundancy and errors, and to maximize readability and understandability. Regardless of the language, the computer underneath still executes in the same manner, and its concepts are far simpler, and its mechanisms far fewer in number. Each discrete piece of logic is, at some level, an assembly language function, and accessing that logic is, at some level, nothing more than a call instruction. It is that world that each of these object oriented constructs must translate into, such that every conceivable operation can be expressed as nothing more than a function call, and the magic, if any, is in doing so efficiently without the developer ever being aware of that original intent.

## Types

A system of execution has at its core a concept of data types. The XVM relies on a core set of types that are – by self-reference – necessary to bootstrap the XVM and those types themselves; these types, being intrinsic to the XVM, are referred to as *intrinsic types*, and are defined in the root module, Ecstasy.xtclang.org. As described previously, the real implementation of these types may differ from implementation to implementation of the XVM specification, but there is an ironclad guarantee: That these types are always available, and provide the capabilities as described by their source code.

The XVM uses a pure object type system, and only an object type system. As the words would imply, a pure object type system does not include support for types that are *not* object types; traditionally such non-object types are known as *primitive types*. The lack of a primitive type system is familiar to Smalltalk programmers, and has emerged as a general trend in newer object-oriented languages. There are several reasons for the emergence of pure object type systems, but the fundamental reason is that it dramatically simplifies a language by reducing the number of intrinsic type systems – typically from two (objects and primitives) to just one (objects). The result is uniformity and simplicity, and thus elegance.

However, there are legitimate technical reasons why primitive types still exist in most contemporary languages. First of all, they are compact: While an object often consumes 8 or 16 bytes of memory at a minimum, a primitive byte is still one byte. Second, an object is likely to incur an additional memory dereference on every access, because objects tend to be accessed via a *pointer*; any virtual behavior incurs at least one more dereference on top of that, because the *function pointer* for the behavior is looked up in a *virtual function table* (Vtable). Third, it is relatively easy to generate efficient native code for a high-level language that uses primitive types, as if the code had been written in a primitive language such as C. Finally, there are native hardware accelerators, such as Single-Instruction Multiple-Data (SIMD) co-processors[[19]](#footnote-19) and CPU instruction sets that require values to be arranged in a **very** explicit layout composed solely of explicitly primitive (CPU-native) types.

Even though raw performance concerns have diminished as the processing throughput of CPUs has increased, it is still crucial for the success of an execution system to carefully consider performance trade-offs. Nonetheless, there are two fundamental realizations that enable the abandonment of a primitive type system without significant concern for a negative performance impact. First, the use of runtime profiling information to optimize code has opened new possibilities by illustrating how conceivable-yet-daunting performance optimizations can be safely realized; this trend is successfully and dramatically illustrated by the Sun (now Oracle) Hotspot JVM. Second, in a pure object-oriented type system, primitive types are translated to objects that are immutable and whose identity corresponds only to the represented value; this knowledge unleashes a slew of potential optimizations. For example, multiple copies of the object can be safely created without concern of identity confusion, since any two objects corresponding to an identical primitive value are in fact the same *singleton* object whether or not their location in memory is the same; as a result, it is conceivable to arbitrarily optimize to using a native representation on the stack, in a register, or as an immediate value, thus achieving the same performance benefits in the executing native code as would have been possible with an explicit primitive type system.

The loss of support for a primitive type system without the loss of potential performance is not accidental. In order to achieve this, there are several common programming concepts that must be hidden[[20]](#footnote-20) such as memory location and method of allocation, and several programming concepts that must be made explicit such as immutability and equality (sameness). These concepts are not limited to those types formerly known as primitive; the associated benefits can be realized with any intrinsic or user types that share these same attributes.

## Type Compatibility

*Type compatibility* is the ability to use a value of a particular type in a situation that requires a specific (and potentially different) type. Three common examples of a *type requirement* occur when (i) an L-Value (e.g. a variable or a property) is being assigned to, (ii) an argument is being passed to a method or function, and (iii) a value is being returned from a method or function. In each case (an L-Value, a parameter, and a return value), there is a specific type that is required, and a value (an expression of a particular type) that is being provided. In each case, the type of the provided value must be *assignment compatible* with the required type.

When possible, the compiler (and the verifier[[21]](#footnote-21)) will attempt to prove (i) that the type of the value is guaranteed to be type compatible, in which case there will be no additional operations required at runtime, or (ii) that the type of the value can not be assignment compatible, which allows an error to be reported. It is self-evident that proving type constraints as either correct or incorrect at compile (or verify) time is desirable, because such an answer is binary and absolute.

However, many type constraints are not testable until the code is actually in the process of executing, because run-time types are often **more** specific than compile-time types:

* A compile-time type specifies a type constraint, but at run-time, a *narrower* type (such as a sub-class) can be used without violating that constraint.
* At compile-time, a type may only be known as a type parameter (a place-holder for an actual type), which means that the run-time type can be any type that does not explicitly violate the type constraint of the type parameter.

One might wonder how being **more** specific with a type could pose a potential type conflict at run-time, but the potential run-time conflicts are a natural consequence of *type covariance*, in both parameterized types and auto-narrowing types, each of which is described in subsequent sections. Type conflicts can occur when a widening conversion hides covariance, or when the assumed type of a value is tested with a type assertion:

* It is possible to provide a more-specific run-time type where a less-specific compile-time type is specified; in such a case, known as a *widening conversion*, the actual run-time type is often unknowable at compile-time, and thus any type conflicts caused by covariance[[22]](#footnote-22) (such as a narrower type required for a parameter) can only be detected at run-time.
* It is possible to explicitly *type-assert* (often referred to as a *type-cast* in other languages), which means to test for a specific *narrower* type at run-time than the known compile-time type; failure of the type assertion results in the run-time raising an exception.

In both cases, type safety is ensured, but because the run-time types are not knowable at compile-time, that type safety is verified at run-time.

(TODO discuss the “shims” concept, and the ability to avoid their use)

-- TODO --

thaConsider an expression whose type is *E*, and a required type *R*.

of an: Types are either compatible, or they are not. Type compatibility indicates that a reference to a value of type V is compatible with can be (i) assigned *as-is* to a variable or a property of a specific type, (ii) passed *as-is* as an argument of a specific parameter type, or (iii) returned *as-is* as a value of a specific return type.

Given two types, *A* and *B*, it is possible to determine if type *B* is *assignment compatible* with type *A*:

* It may be known that type *B* is assignment compatible with type *A*, because *A* and *B* refer to the same identical type.
* It may be known that type *B* is *assignment incompatible* with type *A*, because an attribute of type *A* *conflicts with* an attribute of type *B*, such that no value of type *B* nor any derivative thereof could possibly be assignment compatible with type *A*.
* Assignment compatible
  + Narrowing assignment
  + Widening assignment

Incompatible types

Narrowing by type assertion

Reference widening

Widening with latent type assertion

Narrowing occurs when

Additionally, a type is *assignment compatible* if a reference can be (i) assignedto a variable or a property of a specific type, (ii) passedas an argument of a specific parameter type, or (iii) returnedas a value of a specific return type, either *as-is* or as the result of an *automatic conversion*.

## Parameterized Types

The XVM provides extensive type-safe support for parameterized types, generally referred to in programming languages as *generic types*. A parameterized type declares one or more named *type parameters*, which act as types of their own, but which ultimately act as a level of indirection to an underlying type. For example, imagine a List type that declares an ElementType type parameter; it is then possible to have a type which is a List of String, and another type which is a List of Int.

It is possible to determine whether a parameterized type *produces* and/or *consumes* the types declared by each of its type parameters. Imagine three types used for handling an imaginary Error type: a Log of Error type, a Generator of Error type, and a List of Error type; one could reasonably guess that the Log type *consumes* the Error type (because a caller passes Error objects to it), that the Generator type *produces* the Error type (because it emits Error objects), and that the List type both consumes and produces the Error type (because those objects can be added to the list, and the list can also be queried for the objects already in it).

We posit that type *A* consumes another type *B* if and only if there exists at least one method from type *A* that:

* Has at least one parameter of type *B*;
* Has at least one parameter that produces type *B*; or
* Has at least one return type that consumes type *B*.

We posit that type *A* produces another type *B* if and only if there exists at least one method from type *A* that:

* Has at least one return type of type *B*;
* Has at least one return type that produces type *B*; or
* Has at least one parameter that consumes type *B*.

The determination of whether a type produces and/or consumes another type is useful for predicting errors at compile time, by helping to differentiate likely-correct uses of generic types from likely-incorrect uses of the same.

## Related Types

TODO XTC parent/child inner/outer

## Auto-Narrowing Types

TODO XTC this / parent / child

## Objects

In a traditional imperative language like C, complex data types are typically represented as *structs*; a struct is considered a complex data type because it is composed of any number of *fields*, each of which has its own data type. It is possible both to instantiate a struct “on the stack” without dynamic allocation of memory, and to instantiate a struct “on the heap” using dynamic allocation of memory. Generally, *pointers* to those structs are used as an efficient mechanism to pass a reference-to-a-struct to a function, to store a reference-to-a-struct in a variable or any data structure, etc., without actually having to copy or move the struct itself. Following a pointer to get to the pointed-to struct is called de-referencing the pointer.

In an object-oriented language like C++ (which itself extends the C language), a *class* is a data type that extends the concept of a struct, but by default hides the fields of the struct as *private* members. Like C, it is possible to pass a C++ object using a pointer to the object – which would require de-referencing the pointer to access the object – but it is also possible to pass a C++ *reference* to the object, thus partially hiding the concept of a pointer. Furthermore, code that is part of the definition of the class is automatically invoked when an instance of the class is constructed and destructed. All of these aspects of C++ are designed to support and encourage encapsulation, which is a key tenet of object-oriented languages. C++ also enables the definition of the class itself to be parameterized using *templates*; these templates are used to extend otherwise-generic class definitions with additional compile-time type safety and type-specific optimizations.

Java removed a number of the C and C++ language capabilities altogether:

* Structs;
* Pointers;
* Multi-dimension arrays;
* Control over the memory layout of classes;
* Stack-allocation of objects;
* Creating multiple objects with a single allocation;
* Explicit object destruction and deallocation; and
* Derivative patterns such as RAII.

The result was an elegant compromise that allowed developers to focus on a pure object type system, with a few obvious exceptions, such as:

* The non-object type system represented by primitive values;
* The not-entirely-pure object type system represented by array types, including some “split personality” types like “arrays of primitive values”;
* Global functions (functions not related to object instances) retained via the static keyword;
* Non-virtualized object instantiation (since an exact type must be specified following the new keyword).

Java does not expose pointers to the programmer; instead, Java has *object references*. These act very similarly to pointers, but with a few notable differences:

* The reference is an opaque data structure, in that its only value is the referent itself;
* The memory address of the referent object is not exposed;
* The reference itself cannot be manipulated or modified (e.g. pointer arithmetic);
* A reference is strongly typed and (mostly) type safe;
* There is exactly one reference value that does not have a referent, which is the null value, which has well defined behavior.

On the other hand, like C and C++, Java retained the ability to compare two references for reference equality, but then added a few additional related and (in retrospect) unwise contracts, including:

* Java references are associated with a native *hash-code value* that originally was thought to be “free”, since the value was the memory address (or a direct transformation thereof); maintaining this contract ended up adding an extra four bytes to every object (which was eventually optimized out of many objects at a high cost of optimization complexity).
* Java references served as the basis for Java’s native (i) mutual exclusion, (ii) thread parking, and (iii) thread unparking mechanisms, referred to as (i) synchronized methods and blocks, (ii) wait, and (iii) notify; again, this had the unintended consequences of making every object a potential mutex and parking queue, adding some amount of storage overhead to the header of every object, and adding dramatic optimization complexity to the JVM to reduce the overhead of those unintended consequences.

(Java is truly an engineering marvel, but the greatest feats of the Java team are found in the ways that they have managed to overcome the seemingly insurmountable challenges of their own making. While Google’s V8 engine has accomplished much the same for the morass of JavaScript, the V8 team can at least disclaim the various design flaws that they are so ingenuously[[23]](#footnote-23) compensating[[24]](#footnote-24) for.)

The XVM uses objects and references, in much the same manner as the JVM and the CLR before it.

## Terms

**Object** – Every value, every input, every output – quite literally every *thing* in the XVM is an object. Objects are *managed* by the XVM: They are *instantiated*, they exist for a period of time, and they are *garbage collected* when they are no longer used. An object is a conceptually discrete unit of state and behavior.

**Class** – Every object is “*of a*” class, and that class defines the state that can be held by an object, and the behavior of an object.

**Reference** – Every reference is “*to an*” object (the *referent*). Every interaction with an object occurs through a reference to that object, including interactions that an object has with itself. A reference has a *type*, and that type defines the set of interactions that are possible with the referent object through the reference.

**Type** – A type defines a set of interactions that are possible with an object, expressed in the form of properties and methods.

*Every reference has a type. Every object has a class.*

this – Every object has a reference to itself; the reference to self is the object’s this reference[[25]](#footnote-25).

**Meta** – Every object has an associated representation of its design-time and run-time meta-data; this representation is provided by the object’s *meta* object.

**Struct** – Every object has an associated representation of its structure that contains its state; this representation is the object’s *struct* object. For each property of the object[[26]](#footnote-26), there exists a *field* in the object’s struct that contains the reference held by the property.

**Template** – The unit of design as provided by a developer. Templates are combined to form a class, using information encoded in the template, and following rules defined in this specification. A template is “of a” template category; there are eight template categories, corresponding to the keywords module, package, class, const, enum, service, mixin, and interface. A template has an identity, and defines properties and methods. A template can also define constants, functions, and can contain child templates.

**Property** – A property is a discrete unit of object state; an object’s state is accessed and modified via properties. A property has a constraint type and an identity, and holds a reference to the property’s value. The property value is stored in the object’s structure in a *field*, and is TODO virtual on the object

**Method** – A method is a named piece of functionality – a *behavior* – that can be applied to an object; that object is called a *target object*. As a result of combining multiple templates in order to form a class, a method may represent a *sequence* of method implementations, because each template can contribute an implementation for a given method. (This concept is called a *virtual method*, and the sequence of method implementations is called a *virtual call chain*.) A method, when *bound* to a target object, produces a *function*.

super – For each method implementation in a virtual call chain, the next implementation in the chain is called the *super method* (which implies that the last method implementation in the chain does not have a super method). In each method implementation that has a super method, a function reference named super is available to invoke the super method.

**Constant** – A constant is a named immutable value. The constant may be a *compile-time* constant, if its value is determinable by the compiler, or it may be a *run-time* constant, if its value is computed at run-time, which occurs no later than the first time that the value is used.

**Function** – A function is an *implementation* that can be *invoked*; it represents a sequence of *instructions*. A named function can be defined as part of a template, in a manner similar to a method; unlike a method, a function is not virtual, and a function does not get bound to a target object. A function takes some number of *parameters*, and yields some number of *return values*. When one or more function parameters are *bound* to argument values, a new function is produced (with that many fewer parameters). A function, when *invoked*, executes the instructions in the implementation, and yields the return values.

## Meta Model

While the intrinsic type system is intended to provide the fundamental building blocks from which the software developer can construct their own arbitrarily elaborate types, it also provides the software developer with a definition of the XVM itself, as it is running, including the type system. In other words, the type system includes intrinsic types that define and describe the type system. The use of a self-referential and self-defining type system is not new, but Ecstasy and the XVM carry the concept to its (recursively) logical conclusion.

Based on the concepts already discussed, it should come as no surprise that a class in the XVM is itself an object, and the same applies to all of the constituent pieces of a class, such as methods, properties, and so on. Even a function is an object, which is an incredibly simple and powerful way of expressing a potential action (the function) as a “thing” that can be stored off, passed around, compared with other functions, and so on – all in a transitively closed, type-safe manner. Obviously, a type itself is an object, which itself has a type, which itself is an object, and so on to infinity.

There are a few aspects of the XVM that are treated as if they were closely guarded secrets – things that in theory could easily be exposed through a meta-model, but are not. Things like how an object is laid out in memory, and how that memory is managed – whether it is a dynamic allocation, or on an execution stack, and so on. As described previously, these are conscious choices, because by hiding this information, the XVM is free to organize (and even dynamically reorganize) the native structures and code that constitute a running system.

One of these closely guarded secrets is the structure of an object reference, and that is because it is desirable to avoid native structures and pointers altogether for many common types, such as a Boolean or an Int. By hiding what a reference *is*, the runtime can pretend that there is an object when in reality there is none, and can pretend that there is a reference to that object, when in reality there may not be. In other words, the XVM can choose to implement an object as an object, but it is also free to implement an object in any way that it finds more desirable, such as implementing an object as a machine-native value[[27]](#footnote-27).

Amazingly, that object (that does not actually exist), referenced via a reference (that does not actually exist), can have code that is running with a this (of that non-existent reference), being passed around from place to place, having its virtual methods executed – and all of the language and runtime contracts still hold!

And while the details of the object’s structural organization are purposefully hidden, and while the references themselves are not directly manipulable, the XVM meta-model carefully exposes the concepts of the *referrer* (the origination point of a reference) and the *referent* (the target object of a reference), and magic ensues. Every value, every instantiated object, every method or function result, every property, every calculation – every *object*! – only exists because it is referenced; once it is no longer referenced, it ceases to exist. We often describe that *state of being referenced* as “someone holding onto an object”, and in reality that simply means that a reference to that object is stored in at least one reachable place in the system, be that a variable in a method, a property on another object, an element in an array, and so on. That “place where the reference is stored” is called a Ref.

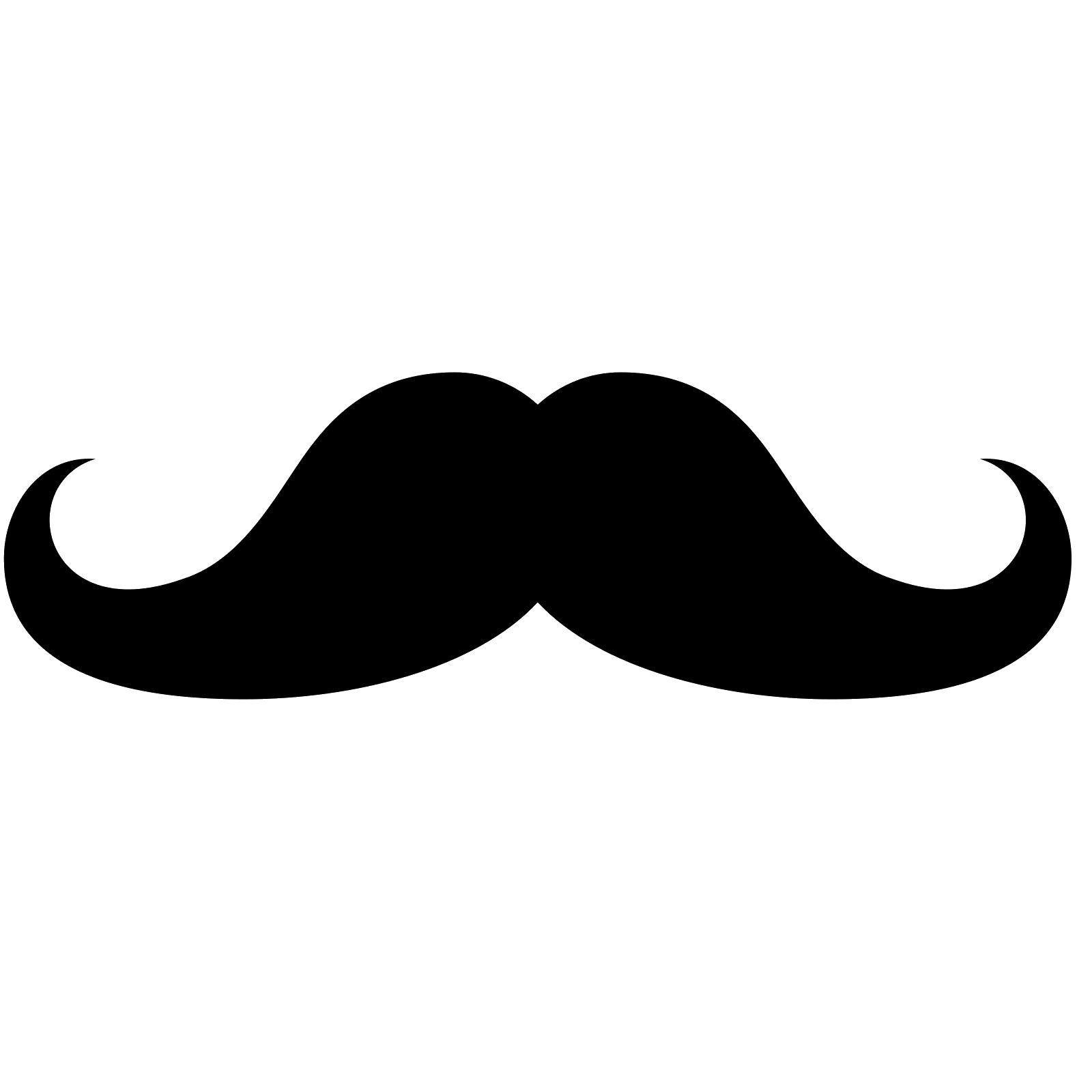
The magic is that each Ref is itself an active, extensible, programmable part of the XVM, and not just some passive address in memory. Every place that an object can be held, and every place that an object can be accessed, there is a Ref, and to operate on a reference, one must ask the Ref to get it, and to change the reference, one must ask the Ref to set it. When a local variable is declared, the declaration is specifying the class of the Ref that will hold the variable’s value. When a property is declared, the declaration is specifying the class of the Ref that will hold the property’s value – and the property declaration may even be declaring a new class of Ref that exists just to manage that one property’s value!

As already described with respect to objects themselves, the contracts for Ref are carefully designed, so that the runtime can pretend that there is a Ref when in reality there may be none, and in the case that a developer ever asks for a reference to the Ref itself, it will always be there.

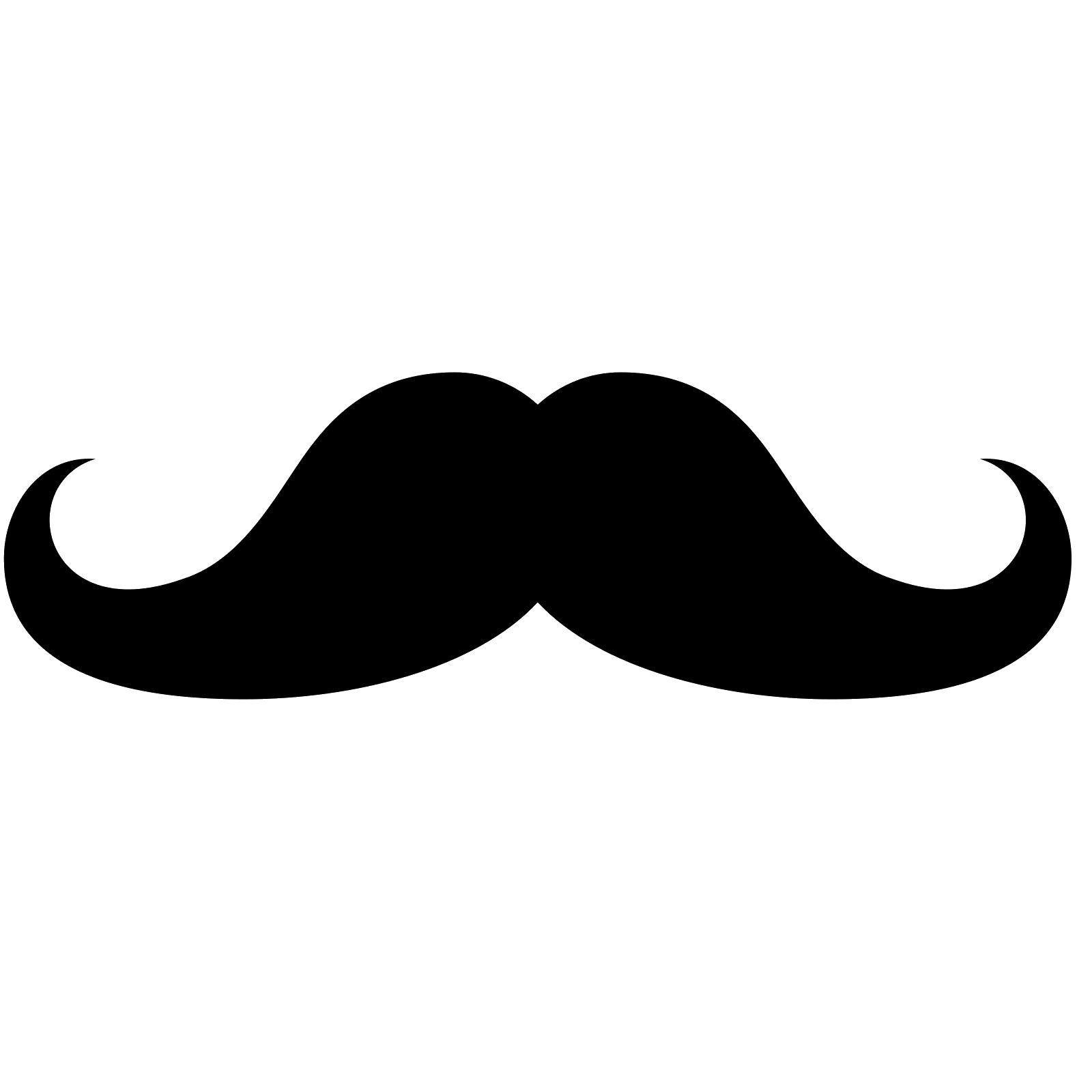
## Mixins and Annotations

The discussion of Ref is an ideal segue to the topic of mixins, and their use in annotations. To understand the concept of a mixin, it helps to start by considering the traditional concept of class inheritance. Inheritance starts with something that we don’t wish to alter (or ruin!), yet it’s something that we wish to use as the basis for something new and different, so instead of altering the original, we place a “pane a glass” over it, and do our work there:

**+**



**=**



By composing in this manner, we still have the original, and we also have a new *derivative* work (the “Monad Lysander”) that exists – without destroying or defiling the original! If these were classes, we would describe the original as the *base class*, and the result as the *derived class*, with the relationship being that the derived class *inherits from* the base class.

Implementation inheritance has long been regarded as an incredibly powerful tool, but it is also regarded as being one of the most fragile forms of composition in an object-oriented system. In the case of Monad Lysander, all of that careful movembrish work is tightly coupled to the intricate underlying details of Mona Lisa. Should we wish to enhance any other masterpiece in a similar manner, inheritance would force us to reproduce all of that painstaking follicular work, almost certainly relying on the “cut & paste” design pattern.

Remembering back to the goals of Ecstasy, we posited: “*a language should enable a developer to locate each piece of design and logic in its one best and natural place*.” That “pane of glass” represents that one best and natural place, but only if it can be applied to any face in any picture, and that ability is what defines a *mixin*.

*A mixin is a template that can be incorporated into any other template that satisfies the mixin’s type constraint, using that type as its joinery into the resulting class.*

In our example, the *type constraint* is “has an upper lip without a mustache”, and that is the point to which the mixin is affixed. Since the mixin’s type constraint is known, the compile-time type for its this is the type specified in the type constraint.

Unlike inheritance, in which a template specifies the template from which it must derive, a mixin can apply to any template that *is of* the necessary type. This allows the mixin – that one “*piece of design and logic*” – to be used over and over again, anywhere that it is needed. Furthermore, any number of mixins can be incorporated into a resulting class, allowing common functionality to be split out into discrete units (each a mixin), and then aggregated together as that functionality is needed.

But perhaps most importantly, a mixin can be incorporated into an existing class *by annotation* to form a new class on the spot; consider this line of Ecstasy code:

return new **@Persistent** Employee(id, name);

And at this point, we must digress, because it’s simply too obvious that one could benefit by adding capabilities – such as persistence – in a business application to business classes – such as an “Employee”. While novel, indubitably clever, and intriguingly interesting, such a solution is also eminently predictable. What makes mixins amazing is that a mixin can be incorporated into a class that is not even known.

Consider a variable, for example. We can prove that a variable exists, precisely because it holds a value in a predictable manner. We can even obtain the Ref object for that variable, and since we have stated as an axiom that an object has a class, we know therefore that the variable itself – being an object! – has a class. But the developer does not “new” that class to make a variable; rather, the existence of that variable just happens – exactly when, and where, and *as* it is needed.

And here, incorporated into that unknown class, is where the mixin performs its magic:

**@Future** HttpResult result = svc.handle(request);

This declaration is *not* incorporating a Future mixin into an HttpResult; rather, it declares that *the class of the variable itself* incorporates the Future mixin. Indeed, the result variable[[28]](#footnote-28) (augmented with its Future functionality) is now capable of being treated as an asynchronous future:

**&**result.handle(e -> genErrorPage(request, e))

.passTo(result -> send(request, result));

The same annotation capability is also used to declare properties. A number of Ref-altering annotations (e.g. @Future, @Atomic, @Lazy, @Soft, and @Weak) are included in the root module, but if the need arises, it is startlingly simple to write a new Ref-altering annotation of your own.

Lastly, annotations can apply to either the variable itself, or to the type contained in the variable; in this example, the variable has a @Lazy mixin, and it holds a reference to an @Unchecked Int object:

@Lazy(() -> 2+2) @Unchecked Int x;

# Ecstasy: The XTC Language

The XTC[[29]](#footnote-29) language is the reference language implementation for the XVM.

## Lexical Structure

XTC code is at its most fundamental level a sequence of Unicode code-points, referred to as *characters*. When a sequence of these characters is stored in a file, it is typically stored in an *encoded* format, for example using the UTF-8 encoding. A number of possible encodings exist for sequences of Unicode characters, but the details of those encodings are outside of the scope of this specification. All encodings must support at least the Unicode code-points in the range U+0000 through U+007F, corresponding to the ASCII character set, and encodings may support up to the entire range of legal Unicode code-points, which at the time of writing is U+0000 through U+10FFFF.

The XTC source code being lexically analyzed is called the *Input*. For purposes of lexical analysis, the *Input* is considered to be a character stream, and that stream is read from left to right.

The first stage of lexical analysis tracks the *line number* and the *line offset* in order to identify exact locations within the source code; both the line number and line offset are zero-based. Any *location* within the source can be identified by a combination of the line number and the line offset, while a selection of the source is identified by a starting location (inclusive) and an ending location (exclusive). The location tracking uses the definition of the *LineTerminator* sequence to determine when to increment the line number and reset the line offset:

*LineTerminator:*

U+000A

U+000B

U+000C

U+000D U+000Aopt

U+0085

U+2028

U+2029

Each *RawCharacter* in the character stream is evaluated to determine if it is the beginning of a *LineTerminator* sequence; if it is, then reading past the *LineTerminator* sequence causes the line number to be incremented and the line offset to be reset to zero. Reading any other *RawCharacter* (that is not the beginning of a *LineTerminator* sequence) causes the line offset to be incremented and does not change the line number. Note that in the case of the two-character *LineTerminator* sequence also known as “CR/LF”, the line number and line offset are undefined in between the contiguous characters U+000D and U+000A.

The second stage of lexical analysis translates Unicode character escape sequences into characters. A Unicode character escape sequence can occur at any point within the source, and must specify a legal Unicode code-point.

*UnicodeCharacterEscapeSequence:*

\u *hex-digit* *hex-digit* *hex-digit* *hex-digit*

\U *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit*

Here are a few examples of how specific characters could be encoded using a Unicode character escape sequence:

|  |  |  |
| --- | --- | --- |
| Character | Code Point | Escape Sequence |
| $ | U+0024 | \u0024 or \U00000024 |
| ¢ | U+00A2 | \u00A2 or \U000000A2 |
| € | U+20AC | \u20AC or \U000020AC |
| 𤭢 | U+24B62 | \U00024B62 |

Since the stream is being analyzed as if it is being read left to right, one character at a time, the character that results from a Unicode character escape sequence is not further evaluated to determine if it could be part of another Unicode character escape sequence; for example, the Unicode code-point U+005C is the character ‘\’, but the source “\u005cu005c” translates simply to the six characters ‘\’, ‘u’, ‘0’, ‘0’, ‘5’, ‘c’, and not further to the single character ‘\’. Similarly, since the Unicode character escape sequences are evaluated after the line number and line number processing, the Unicode character escape sequence “\u000a” does not affect the line number; instead, since the escape sequence is six characters long, it increases the line offset by six, even though a subsequent stage of lexical analysis would only encounter a single character. Note that neither a recursively encoded format nor an escapable format is supported; “\uu0024” is translated simply as the seven characters ‘\’, ‘u’, ‘u’, ‘0’, ‘0’, ‘2’, ‘4’, and “\\u0024” is translated as the two characters ‘\’, ‘$’.

The third stage of lexical analysis converts the residual *Input* stream of characters into a stream of *InputElements*, which are *WhiteSpace*, *Comments,* and *Tokens*, prefaced with an optional *BeginningOfFile* serving as Unicode byte-ordering indicator, and terminated with an optional *EndOfFile*; the *WhiteSpace* and *EndOfFile* elements are discarded.

*Input:*

*BeginningOfFileopt InputElementsopt EndOfFileopt*

*BeginningOfFile:*

U+FEFF

*InputElements:*

*InputElement*

*InputElements InputElement*

*InputElement:*

*WhiteSpace*

*Comment*

*Token*

*EndOfFile:*

U+001A

Whitespace is defined as the union of the traditional ASCII whitespace characters and the three Unicode “separator” categories. Here is the entire table of whitespace characters:

|  |  |  |  |
| --- | --- | --- | --- |
| Code Point | Decimal | Name | Description |
| U+0009 | 9 | HT | Horizontal Tab |
| U+000A | 10 | LF | Line Feed |
| U+000B | 11 | VT | Vertical Tab |
| U+000C | 12 | FF | Form Feed |
| U+000D | 13 | CR | Carriage Return |
| U+001C | 28 | FS | File Separator |
| U+001D | 29 | GS | Group Separator |
| U+001E | 30 | RS | Record Separator |
| U+001F | 31 | US | Unit Separator |
| U+001A | 26 | SUB | End-of-File, or “control-Z” |
| U+0020 | 32 | SP | Space |
| U+0085 | 133 | NEL | Next Line |
| U+00A0 | 160 | &nbsp; | Non-breaking space |
| U+1680 | 5760 |  | Ogham Space Mark |
| U+2000 | 8192 |  | En Quad |
| U+2001 | 8193 |  | Em Quad |
| U+2002 | 8194 |  | En Space |
| U+2003 | 8195 |  | Em Space |
| U+2004 | 8196 |  | Three-Per-Em Space |
| U+2005 | 8197 |  | Four-Per-Em Space |
| U+2006 | 8198 |  | Six-Per-Em Space |
| U+2007 | 8199 |  | Figure Space |
| U+2008 | 8200 |  | Punctuation Space |
| U+2009 | 8201 |  | Thin Space |
| U+200A | 8202 |  | Hair Space |
| U+2028 | 8232 | LS | Line Separator |
| U+2029 | 8233 | PS | Paragraph Separator |
| U+202F | 8239 |  | Narrow No-Break Space |
| U+205F | 8287 |  | Medium Mathematical Space |
| U+3000 | 12288 |  | Ideographic Space |

Since the longest possible translation is used at each stage of lexical analysis, *WhiteSpace* and *Comments* are often necessary for separating two tokens that could otherwise be translated as a single, larger token.

*WhiteSpace:*

*WhiteSpaceElement*

*WhiteSpace WhiteSpaceElement*

*WhiteSpaceElement:*

*SpacingElement*

*LineTerminator*

*SpacingElement:*

U+0009

U+001C

U+001D

U+001E

U+001F

U+0020

U+00A0

U+1680

U+2000

U+2001

U+2002

U+2003

U+2004

U+2005

U+2006

U+2007

U+2008

U+2009

U+200A

U+202F

U+205F

U+3000

*LineTerminator:*

U+000A

U+000B

U+000C

U+000D U+000Aopt

U+0085

U+2028

U+2029

*Comment:*

*EndOfLineComment*

*EnclosedComment*

*EndOfLineComment:*

// *InputCharactersopt*

*InputCharacters:*

*InputCharacter*

*InputCharacters InputCharacter*

*InputCharacter:*

*RawCharacter* except *LineTerminator*

*EnclosedComment:*

/\* *EnclosedCommentTail*

*EnclosedCommentTail:*

\* *AsteriskCommentTail*

*NotAsteriskCharacter EnclosedCommentTail*

*AsteriskCommentTail:*

/

\* *AsteriskCommentTail*

*NotAsteriskOrSlashCharacter* *EnclosedCommentTail*

*NotAsteriskCharacter:*

*RawCharacter* except \*

*NotAsteriskOrSlashCharacter:*

*RawCharacter* except / or \*

*RawCharacters:*

*RawCharacter*

*RawCharacters RawCharacter*

*RawCharacter:*

any legal Unicode code-point except *BeginningOfFile* or *EndOfFile*

It should be obvious that the “control-Z” character used for the *EndOfFile* production, which is an historical anachronism, is only going to be permitted to occur at the end of the file. Similarly, the Unicode byte-ordering indicator character used for the *BeginningOfFile* production is only permitted to occur at the beginning of the file. Note that the *BeginningOfFile* production can be utilized in a number of different Unicode encodings, including UTF-8, UTF-16, and UTF-32 encoded files, and in either little-endian or big-endian byte order for both the UTF-16 and UTF-32 formats, and thus represents a number of different byte sequences that one may encounter at the beginning of a file. Any program operating on XTC source files should recognize and support no less than these five Unicode encodings and the plain ASCII encoding.

Since the stream is being analyzed as if it is being read left to right, and since the *EndOfLineComment* is seeking only for a *LineTerminator* or the end of the stream, any “//”, “/\*”, or “\*/” character sequences encountered while producing an *EndOfLineComment* will be ignored. Likewise, since the *EnclosedComment* is seeking only for the “\*/” character sequence, any “//” or “/\*” character sequences encountered while producing an *EnclosedComment* will be ignored. By logical inference, comments cannot be nested.

## Lexical Literals

There are several fundamental literal forms that are handled by the lexical stage of compilation, including numeric literals and character string literals.

For numeric literals, Ecstasy supports base-2 (bit/binary), base-8 (octal), base-10 (digit/decimal), and base-16 (hexit/hexadecimal) values. Ecstasy requires a special prefix (the digit ‘0’ followed by a non-digit character) to indicate a value uses a base (radix) other than base-10[[30]](#footnote-30). Fractional values can also be specified in base-2, base-8, base-10, and base-16. Floating point values can have an exponent expressed in base-2, base-8, base-10, or base-16, and a floating point value that has both a mantissa and an exponent can use a different base for each.

*NumericLiteral:*

*IntegerLiteral*

*FloatingPointLiteral*

*IntegerLiteral:*

*Signopt UnsignedIntegerLiteral*

*Sign:* one of

+ -

*UnsignedIntegerLiteral:*

*BitLiteral*

*OctLiteral*

*DigitLiteral*

*HexitLiteral*

*BitLiteral:*

0 *BinaryIndicator Bits*

*BinaryIndicator:* one of

B b

*Bits:*

*Bit BitsOrUnderscoresopt*

*Bit:* one of

0 1

*BitsOrUnderscores:*

*BitOrUnderscore*

*BitOrUnderscores BitOrUnderscore*

*BitOrUnderscore:*

*Bit*

\_

*OctalLiteral:*

0o *Octals*

*Octals:*

*Octal OctalsOrUnderscoresopt*

*Octal:* one of

0 1 2 3 4 5 6 7

*OctalsOrUnderscores:*

*OctalOrUnderscore*

*OctalOrUnderscores OctalOrUnderscore*

*OctalOrUnderscore:*

*Octal*

\_

*DigitLiteral:*

*Digits*

*Digits:*

*Digit DigitsOrUnderscoresopt*

*Digit:* one of

0 1 2 3 4 5 6 7 8 9

*DigitsOrUnderscores:*

*DigitOrUnderscore*

*DigitOrUnderscores DigitOrUnderscore*

*DigitOrUnderscore:*

*Digit*

\_

*HexitLiteral:*

0 *HexIndicator Hexits*

*HexIndicator:* one of

X x

*Hexits:*

*Hexit HexitsOrUnderscoresopt*

*Hexit:* one of

0 1 2 3 4 5 6 7 8 9 A a B b C c D d E e F f

*HexitsOrUnderscores:*

*HexitOrUnderscore*

*HexitOrUnderscores HexitOrUnderscore*

*HexitOrUnderscore:*

*Hexit*

\_

*FloatingPointLiteral:*

*IntegerLiteral Exponent*

*Signopt FractionalLiteral Exponentopt*

*FractionalLiteral:*

0 *BinaryIndicator Bitsopt* . *Bits*

0 o *Octalsopt* . *Octals*

*Digitsopt* . *Digits*

0 *HexIndicator Hexitsopt* . *Hexits*

*Exponent:*

*ExponentIndicator IntegerLiteral*

*ExponentIndicator:*

*DecimalExponentIndicator*

*BinaryExponentIndicator*

*DecimalExponentIndicator:* one of

E e

*BinaryExponentIndicator:* one of

P p

There are three forms of character string literals in Ecstasy: The single character literal (in single quotes), the normal character string literal (in double quotes), and the free-form literal (inside of a two-dimensional textual frame). The first two are relatively self-explanatory for anyone with knowledge of contemporary languages, but the last is something altogether insane: Using the Unicode “box drawing” block of characters, and assuming a fixed-width font (for purposes of column alignment), a block can exist in Ecstasy code that contains free-form, multi-line text; for example:

String s = ╔═════════════════════╗  
 ║<html><body> ║  
 ║you could paste an ║  
 ║entire file in here ║  
 ║</body></html> ║  
 ╚═════════════════════╝;

The purpose of the free-form literal is to support the inclusion of entire text files, as-is, within source code files. This is particularly useful for including HTML, XML, JSON, and other templates as part of an application, located (embedded) right next to the logic with which they are used, without requiring elaborate concatenation in the source code itself. On the other hand, this capability does strongly indicate a need for a language-aware IDE that can manage the formatting of the embedded text.

*CharacterLiteral:*

' *SingleCharacter* '

*SingleCharacter:*

*InputCharacter* except \ or '

*CharacterEscape*

*StringLiteral:*

" *CharacterStringopt* "

*FreeformLiteral*

*CharacterString:*

*StringCharacter*

*CharacterString StringCharacter*

*StringCharacter:*

*InputCharacter* except \ or "

*CharacterEscape*

*CharacterEscape:*

\ \

\ "

\ '

\ b

\ f

\ n

\ r

\ t

*FreeformLiteral  
 FreeformTop FreeformLines FreeformBottom  
  
FreeformTop  
 Whitespaceopt FreeformUpperLeft NoWhitespace FreeformHorizontals NoWhitespace* **→**

*FreeformUpperRight Whitespaceopt LineTerminator*

*FreeformLines  
 FreeformLine  
 FreeformLines FreeformLine  
  
FreeformLine  
 Whitespaceopt FreeformVertical FreeformChars FreeformVertical Whitespaceopt LineTerminator  
  
FreeformChars  
 FreeformChar  
 FreeformChars FreeformChars  
  
FreeformChar  
 InputCharacter except FreeFormReserved or LineTerminator  
  
FreeformBottom  
 Whitespaceopt FreeformLowerLeft NoWhitespace FreeformHorizontals NoWhitespace* **→**

*FreeformLowerRight*

*FreeFormReserved  
 FreeformUpperLeft  
 FreeformUpperRight  
 FreeformLowerLeft  
 FreeformLowerRight  
 FreeformHorizontal  
 FreeformVertical  
  
FreeformUpperLeft  
 U+250C* ┌ *U+250D* ┍ *U+250E* ┎ *U+250F* ┏ *U+2552* ╒ *U+2553* ╓ *U+2554* ╔ *U+256D* ╭ *FreeformUpperRight  
 U+2510* ┐ *U+2511* ┑ *U+2512* ┒ *U+2513* ┓ *U+2555* ╕ *U+2556* ╖ *U+2557* ╗ *U+256E* ╮ *FreeformLowerLeft  
 U+2514* └ *U+2515* ┕ *U+2516* ┖ *U+2517* ┗ *U+2558* ╘ *U+2559* ╙ *U+255A* ╚ *U+2570* ╰ *FreeformLowerRight  
 U+2518* ┘ *U+2519* ┙ *U+251A* ┚ *U+251B* ┛ *U+255B* ╛ *U+255C* ╜ *U+255D* ╝ *U+256F* ╯ *FreeformHorizontals  
 FreeformHorizontal  
 FreeformHorizontals NoWhitespace FreeformHorizontal  
  
FreeformHorizontal  
 U+2500* ─ *U+2501* ━ *U+2504* ┄ *U+2505* ┅ *U+2508* ┈ *U+2509* ┉ *U+254C* ╌ *U+254D* ╍ *U+2550* ═ *FreeformVertical  
 U+2502* │ *U+2503* ┃ *U+2506* ┆ *U+2507* ┇ *U+250A* ┊ *U+250B* ┋ *U+254E* ╎ *U+254F* ╏ *U+2551* ║

## Lexical Symbols, Keywords, and Identifiers

Symbols are one or more characters (using the *“longest possible translation”* principle) that have a meaning in the language:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| : | { | << | ?: | &= | < |
| ; | } | >> | = | |= | <= |
| , | [ | >>> | += | ^= | > |
| . | ] | & | -= | &&= | >= |
| .. | + | | | \*= | ||= | ++ |
| ... | - | ^ | /= | := | -- |
| @ | \* | ~ | %= | ?:= | -> |
| ? | / | ! | <<= | == | \_ |
| ( | % | && | >>= | != |  |
| ) | /% | || | >>>= | <=> |  |

The following table contains all of the pre-defined keywords and identifiers (again, using the *“longest possible translation”* principle):

|  |  |  |
| --- | --- | --- |
| allow | function | static |
| as | if | struct |
| assert | immutable | super |
| assert:always | implements | switch |
| assert:once | import | this |
| assert:test | import:embedded | this:module |
| assert:debug | import:required | this:private |
| avoid | import:desired | this:protected |
| break | import:optional | this:public |
| case | incorporates | this:service |
| catch | instanceof | this:struct |
| class | interface | this:target |
| conditional | into | throw |
| const | is | TODO |
| construct | mixin | try |
| continue | module | typedef |
| default | new | using |
| delegates | package | val |
| do | prefer | var |
| else | private | void |
| enum | protected | while |
| extends | public | with |
| finally | return |  |
| for | service |  |

The last lexical element is the identifier. An identifier is composed of a sequence of characters[[31]](#footnote-31), the first of which must be a letter or an underscore, and the subsequent characters can include numbers and currency symbols:

|  |  |  |  |
| --- | --- | --- | --- |
| Category | Unicode | Start | Trail |
| Letter | Lu Ll Lt  Lm Lo | ✔ | ✔ |
| Mark | Mn Mc Me |  | ✔ |
| Number | Nd Nl No |  | ✔ |
| Currency | Sc |  | ✔ |
| Underscore | U+005F | ✔ | ✔ |

The following table contains all of the pre-defined keywords and identifiers (again, using the *“longest possible translation”* principle):

*Identifier:*

IdentifierStart IdentifierFinish*opt*

*IdentifierStart:*

*Letter*

*\_*

*IdentifierFinish:*

*IdentifierTrail*

*IdentifierFinish IdentifierTrail*

*IdentifierTrail:*

*IdentifierStart*

*Mark*

*Number*

*Currency*

## Definite Assignment

There are several mechanisms that are used to identify potential “unassigned variable” errors at compile time, and to help the Ecstasy compiler produce the appropriate code for capabilities such as variable *capture*. The following terms will be used:

* **Definitely Assigned** – given a variable, and given a specific point in the code, the variable has provably been assigned a value by the time that point in the code has been reached; it is a compile-time error and a runtime exception to access a variable that has not been assigned.
* **Definitely Unassigned** – given a variable, and given a specific point in the code, the variable has provably *never* been assigned a value by the time that point in the code has been reached.
* **Effectively Final** – given a variable, the variable is provably assigned once and only once for the lifetime of the variable.
* **Explicitly Final** – a manner of declaring a variable such that it is a compile-time error and run-time exception for the variable to be assigned more than once.
* **Permissibly Unassigned** – a manner of declaring a variable such that the definite assignment rules are ignored at compile time for that one variable.
* **Local Variable** – a named variable declared within a method or function body, and which is created (if and as needed) each time that the method or function is executed.
* **Lexically Scoped** – a local variable begins its lexical scope at the point in the source code at which it is declared, and continues to the end of the statement within which it is declared.
* **Structurally Scoped** – a local variable *begins* its structural scope at the point in the source code at which it is declared; *suspends* its structural scope at any point that declares a nested class (including an anonymous inner class), property, or method/function (including a lambda); *resumes* its structural scope at the conclusion of any such declaration; and *continues* its structural scope to the end of the statement within which it is declared. (In other words, a variable’s structural scope is the portion of a variable’s lexical scope for which the compiler produces code for the body of the method/function that declared the local variable.)
* **Variable Capture** – the mechanism by which access to a local variable is made possible from a lambda expression or an anonymous inner class, when that local variable is lexically scoped but not structurally scoped.

Definite assignment is enforced for constants, properties, and local variables:

* It is relatively simple to prove definite assignment at compile time for constants, since they are required to have a value specified as part of their declaration.
* Properties that have a field may have an initial value specified as part of their declaration, or may be initialized by a constructor. It is a compile-time error for a class to have a constructor that does not result in a value being assigned to each property that has a field. It is a runtime exception for an object to be instantiated in a manner that does not assign a value to each property that has a field.
* Local variables must be definitely assigned before they are used. While local variables are typically assigned as part of their declaration, it is possible for a local variable to be declared without being assigned. Additionally, there are complexities associated with control flow mechanisms (loops, conditionals, etc.) that must be factored in to the determination of definite assignment.

To this end, the compiler tracks three different attributes (definitely assigned, definitely unassigned, and effectively final) as it relates to each variable, and each statement and expression is specified in terms of how it affects these three attributes. These three attributes form the *variable assignment state*, or VAS[[32]](#footnote-32). Specifically, by tracking these three different attributes for every variable, the compiler is able to determine for each statement and expression:

* Whether the variable is definitely unassigned *before* the statement or expression;
* Whether the variable is definitely unassigned *after* the statement or expression;
* Whether the variable is definitely assigned *before* the statement or expression;
* Whether the variable is definitely assigned *after* the statement or expression.

For loops, conditional statements and expressions, and expressions of type Boolean, the compiler is able to further determine:

* Whether the variable is definitely unassigned after the statement or expression *when true*;
* Whether the variable is definitely unassigned after the statement or expression *when false*;
* Whether the variable is definitely assigned after the statement or expression *when true*;
* Whether the variable is definitely assigned after the statement or expression *when false.*

Lastly, the compiler is able to determine which local variable are effectively final.

The following acronyms are used:

* VAS – the Variable Assignment State;
* VAST – the Variable Assignment State *when true*; and
* VASF – the Variable Assignment State *when false*.

# Expressions

An *expression* is generally a language construct that has a type and yields a value. The number 42 is a *LiteralExpression*, for example, that has some numeric type and yields a value of that type that is the number 42, while the cliché foo() function or method call is an *InvocationExpression* and yields whatever value(s) that the foo() function or method returns.

In Ecstasy, while the most common case is that each expression yields a single value, it is possible for an expression to yield zero values (a *void* expression), or multiple values. The expression has a compile-time type for each value that it will yield. When an expression is required that yields *n* values, any expression that yields at least *n* values (and whose types are compatible) can be used, with the additional values being discarded.

An expression may have an *implicit type*[[33]](#footnote-33), which is the compile-time type of the value that the expression can be proven to yield. An expression may also be able to produce (or convert to) another type, if it is requested to do so. In some cases, an expression may be able to determine the type that it should produce, based upon a requested type; this is called *type inference*, and the resulting type is the *inferred type*.

In common compiler terminology, an expression yields either an *L-Value* or an *R-Value*. An L-Value represents a variable to which a value can be assigned, while an R-Value represents a value. Some expressions, such as a *NameExpression* (used for a variable name, among other things) or an *ArrayAccessExpression*, can represent either an L-Value or an R-Value, depending on how the expression is being used.

By default, an expression does not change the VAS: A variable is unassigned after an expression if it is unassigned before the expression; assigned after an expression if assigned before the expression; and assumed to be effectively final after the expression if assumed to be effectively final before the expression. Any behavior that differs will be documented for that expression form.

There are a number of terms that are used to describe how an expression type is determined, and potentially converted:

* A *uni-implicit type* is a type determined from a single input type, such that the result type is the same as the input type, except when the input type is the type of an enumeration value, in which case the result type is the type of the enumeration itself.
* A *bi-implicit type* is determined in a manner similar to a uni-implicit type, except that it has two types as inputs. If either (after adjusting any enumeration value type to its enumeration type) is assignable to the other, then that assignable-to type issued as the result type.
* A *sequence-implicit* type is determined in a manner similar to a bi-implicit type, except that has any number of types as inputs.
* Assignment conversion is used when an expression of type T1 is being assigned to type T2, T1 “is a” T2 is false, and there exists a non-ambiguously best @Auto conversion method on T2 that yields a type that “is a” T1. The assignment conversion uses that @Auto conversion method in order to obtain the result.

An expression is said to be *short-circuiting* if the expression may *arc* (dislocate its execution) to a *ground* point, without yielding a value. The ground point is provided by an enclosing expression or statement, but not all expressions and statements provide a ground point, and in many cases, a short-circuiting expression is not allowed to occur. To differentiate between compile-time and run-time behavior:

* An expression may *arc* at runtime, which means that the short-circuit mechanism is actually triggered as part of the runtime evaluation of the expression; and
* An expression is said to *short-circuit* iff the compiler determines that the expression has the ability to arc at runtime.

Syntactically, the *Expression* construct represents the trailing “else” (or *ground*) for a short-circuiting expression:

*Expression:*

*ElseExpression*

## ElseExpression

The grounding “else” expression is a fairly new construct in programming languages; syntactically, it is similar to the “else” branch of a ternary expression:

a : b

The implicit type of the expression is the bi-implicit type of the type of expression a and b.

The expression yields the result of the expression a. If the expression a arcs, then the expression yields the result of the expression b.

The expression short-circuits if expression b short-circuits.

Definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS before b is the VAS after a. (TODO)
* The VAS after the expression is the join of the VAS after a and the VAS after b.

Note that the “else” operator groups to the right, which is different from most other binary operators:

*ElseExpression:*

*TernaryExpression*

*TernaryExpression* : *Expression*

That means that a : b : c is treated the same as a : (b : c) . (The second form of the construct cannot use *Expression* on the left side, because that would recurse infinitely.)

## TernaryExpression

A ternary expression is a well-known short-hand form of an “if statement”, but in the form of an expression:

a ? b : c

The type of the expression a must be Boolean. The implicit type of the expression is the bi-implicit type of the type of expression b and the type of expression c. The types of expressions b and c may use assignment conversion if necessary to the type of the expression.

If the expression a yields True, then the expression yields the result of the expression b. If the expression a yields False or short-circuits, then the expression yields the result of the expression c.

The expression short-circuits if either expression b or expression c short-circuits.

Definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS before b is the VAST after a.
* The VAS before c is the VASF after a.
* The VAS after the expression is the join of the VAS after b and the VAS after c.
* If the type of the expression is Boolean, then:
  + The VAST after the expression is the join of the VAST after b and the VAST after c.
  + The VASF after the expression is the join of the VASF after b and the VASF after c.

Note the unusual requirement for whitespace before the ‘?’ operator, which differentiates this form of expression from the *NotNullExpression*:

*TernaryExpression:*

*ElvisExpression*

*ElvisExpression Whitespace* ? *ElvisExpression* : *TernaryExpression*

## ElvisExpression

An Elvis expression is a relatively new occurrence in languages, and is in the form:

a ?: b

The type of the expression a must be Nullable[[34]](#footnote-34); in other words, it must be possible to assign Null to an L-Value of the type of the expression a. The implicit type of the expression is the bi-implicit type of the *non-*Nullable type of expression a and the type of expression b; if there is no bi-implicit type, then a type intersection of the *non-*Nullable type of expression a and the type of expression b is used instead. The types of expressions a and b may use assignment conversion if necessary to the type of the expression.

If the expression a yields a value other than Null, then the expression yields the result of the expression a. If the expression a yields Null or short-circuits, then the expression yields the result of the expression b. In other words, it is very similar in its behavior to the following, with the primary difference being that the expression a is only evaluated once in the Elvis expression form:

a == Null ? b : a

The expression short-circuits if expression b short-circuits.

Definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS before b is the VAS after a.
* The VAS after the expression is the join of the VAS after a and the VAS after b.
* If the type of the expression is Boolean, then:
  + The VAST after the expression is the join of the VAST after a and the VAST after b.
  + The VASF after the expression is the join of the VASF after a and the VASF after b.

Note that the Elvis operator groups to the right, which is different from most other binary operators:

*ElvisExpression:*

*OrExpression*

*OrExpression* ?: *ElvisExpression*

That means that a ?: b ?: c is treated the same as a ?: (b ?: c) .

## OrExpression

The logical “or” expression is a well-known expression form:

a || b

The type of the expressions a and b must both be Boolean. The implicit type of the expression is Boolean.

If the expression a yields True, then the expression yields True[[35]](#footnote-35). If the expression a yields False, then the expression yields the result of the expression b.

The expression short-circuits if either expression a or b short-circuits.

Definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS before b is the VASF after a.
* The VAS after the expression is the join of the VAST after a and the VAS after b.
* The VAST after the expression is the join of the VAST after a and the VAST after b.
* The VASF after the expression is the VASF after b.

The OrExpression groups to the left, so a || b || c is treated as (a || b) || c :

*OrExpression:*

*AndExpression*

*OrExpression* || *AndExpression*

## AndExpression

The logical “and” expression is a well-known expression form:

a && b

The type of the expressions a and b must both be Boolean. The implicit type of the expression is Boolean.

If the expression a yields False, then the expression yields False[[36]](#footnote-36). If the expression a yields True, then the expression yields the result of the expression b.

The expression short-circuits if either expression a or b short-circuits.

Definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS before b is the VAST after a.
* The VAS after the expression is the join of the VASF after a and the VAS after b.
* The VAST after the expression is the VAST after b.
* The VASF after the expression is the join of the VASF after a and the VASF after b.

The AndExpression groups to the left, so a && b && c is treated as (a && b) && c :

*AndExpression:*

*BitOrExpression*

*AndExpression* && *BitOrExpression*

## BitOrExpression, BitXorExpression, and BitAndExpression

The bitwise “or”, “xor”, and “and” expressions are well-known expression forms:

a | b

a ^ b

a & b

The type of the expressions a must have an unambiguously single best “or”/“|” operator[[37]](#footnote-37) defined (or “xor”/“^” for BitXorExpression, or “and”/“&” for BitAndExpression) that takes an argument of the type of expression b; the implicit type of the expression is the return type of that operator.

The expression yields the result of the invocation of the operator method against the target reference yielded by expression a, using the value yielded by expression b as the argument to that method.

The expression short-circuits if either expression a or b short-circuits.

Definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS before b is the VAS after a.
* The VAS after the expression is the VAS after b.

These expressions group to the left, so a | b | c is treated as (a | b) | c :

*BitOrExpression:*

*BitXorExpression*

*BitOrExpression* | *BitXorExpression*

*BitXorExpression:*

*BitAndExpression*

*BitXorExpression* ^ *BitAndExpression*

*BitAndExpression:*

*EqualityExpression*

*BitAndExpression* & *EqualityExpression*

## EqualityExpression

Comparison for purposes of equality uses the well-known expression forms:

a == b

a != b

The type of the expressions a and b must both be compatible for purposes of equality. In the simplest case[[38]](#footnote-38), this means that there must be a bi-implicit type of the expressions a and b, and that type must have an equals() function[[39]](#footnote-39) that matches a specific signature pattern:

static <CTT extends *This*> Boolean equals(CTT v1, CTT v2)

Where CTT is the compile-time bi-implicit type of the expressions a and b, and *This* is the class containing the equals() function. The implicit type of the expression is Boolean.

The type of expression a can also be used to infer the type of the expression b, such as in the following example:

enum Color {Red, Green, Blue};  
Color c = chooseAnyColor();

// because the left-hand type is of type Color, the right-hand

// side does not need to be qualified e.g. "Color.Red"

if (c == Red)

{

c = Blue;

}

The expression short-circuits if either expression a or b short-circuits.

The expression uses the default left-to-right definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS before b is the VAS after a.
* The VAS after the expression is the VAS after b.

The EqualityExpression groups to the left, so a == b == c is treated as (a == b) == c :

*EqualityExpression:*

*RelationalExpression*

*EqualityExpression* == *RelationalExpression*

*EqualityExpression* != *RelationalExpression*

## RelationalExpression

Comparison for purposes of relation uses a number of well-known expression forms, and introduces a few new operators:

a < b

a > b

a <= b

a >= b

a <=> b

a is b

a as b

The operators and their result types are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Op | Type of b | Result Type |
| Less Than | < | *bi-implicit* | Boolean |
| Greater Than | > | *bi-implicit* | Boolean |
| Less Than/Equals  *(Not Greater Than)* | <= | *bi-implicit* | Boolean |
| Greater Than/Equals  *(Not Less Than)* | >= | *bi-implicit* | Boolean |
| Order Comparison  (*Spaceship* operator) | <=> | *bi-implicit* | Ordered |
| Test type | is | Type | Boolean |
| Assert type | as | Type | b |

The rules for the traditional comparison operators (<, >, <=, >=) closely match the rules for the EqualityExpression above, including both input and output types:

* The type of the expressions a and b must both be compatible for purposes of equality;
* There must be a bi-implicit type of the expressions a and b;
* That bi-implicit type must have an compare() function that matches a specific signature pattern[[40]](#footnote-40);
* The type of expression a can also be used to infer the type of the expression b;
* The implicit type of the expression is Boolean.

The spaceship operator (<=>) is named for its similarity to the representation of spaceships in the old text-mode Star Trek game in BASIC. This operator is used to compare to values, to determine if the first is less than (<), equal to (=), or greater than (>) the second. The rules are identical to the comparison operators above, except that the implicit type of the expression is Ordered[[41]](#footnote-41).

The last two operators are is and as. The is form of the expression tests whether the value yielded by the expression a is of the type specified in the type expression b; the implicit type of the expression is Boolean. The as form of the expression also tests whether the value yielded by the expression a is of the type specified in the type expression b; the implicit type of the expression is the type specified in the type expression b.

Object o = foo();

Int n = 0;

if (o is Int)

{

n = o as Int;

}

The expression short-circuits if either expression a or b short-circuits.

The expression uses the default left-to-right definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS before b is the VAS after a.
* The VAS after the expression is the VAS after b.

The *RelationalExpression* groups to the left[[42]](#footnote-42), so a < b < c is treated as (a < b) < c :

*RelationalExpression:*

*RangeExpression*

*RelationalExpression* < *RangeExpression*

*RelationalExpression* > *RangeExpression*

*RelationalExpression* <= *RangeExpression*

*RelationalExpression* >= *RangeExpression*

*RelationalExpression* <=> *RangeExpression*

*RelationalExpression* is *TypeExpression*

*RelationalExpression* as *TypeExpression*

## RangeExpression, ShiftExpression, AdditiveExpression, and MultiplicativeExpression

Each of these expressions in the “binary operator” form of:

a *op* b

Each operator, the associated default operator method name, and a brief description of the operator conceptual usage are as follows:

|  |  |  |
| --- | --- | --- |
| Op | Method | Description |
| .. | through | Range (or interval) of values |
| << | shiftLeft | Integer bit shift |
| >> | shiftRight | Integer bit shift (signed) |
| >>> | shiftAllRight | Integer bit shift (unsigned) |
| + | add | Numeric addition |
| - | sub | Numeric subtraction |
| \* | mul | Numeric multiplication |
| / | div | Numeric division |
| % | mod | Numeric modulo |
| /% | divmod | Numeric division and modulo |

The exact meaning of each operator is determined by each class that chooses to implement the operator[[43]](#footnote-43).

The type of the expression a must have an unambiguously single best operator defined that takes an argument of the type of expression b; the implicit type of the expression is the return type of that operator. (The operator for the *divmod* expression has two return types – one for the dividend and one for the modulo.)

The expression yields the result of the invocation of the operator method against the target reference yielded by expression a, using the value yielded by expression b as the argument to that method.

The expression short-circuits if either expression a or b short-circuits.

Definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS before b is the VAS after a.
* The VAS after the expression is the VAS after b.

These expressions group to the left, so a + b + c is treated as (a + b) + c :

*RangeExpression:*

*ShiftExpression*

*RangeExpression* .. *ShiftExpression*

*ShiftExpression:*

*ShiftExpression*

*ShiftExpression* << *AdditiveExpression*

*ShiftExpression* >> *AdditiveExpression*

*ShiftExpression* >>> *AdditiveExpression*

*AdditiveExpression:*

*MultiplicativeExpression*

*AdditiveExpression* + *MultiplicativeExpression*

*AdditiveExpression* - *MultiplicativeExpression*

*MultiplicativeExpression:*

*PrefixExpression*

*MultiplicativeExpression* \* *PrefixExpression*

*MultiplicativeExpression* / *PrefixExpression*

*MultiplicativeExpression* % *PrefixExpression*

*MultiplicativeExpression* /% *PrefixExpression*

## PrefixExpression

Each of these expressions is in the “unary operator” form of:

*op* a

Each operator, the associated default operator method name, and a brief description of the operator conceptual usage are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op | Method | Required Type | Result Type | Description |
| + |  | Number | Same as a | Numeric positive |
| - | neg | Has @Op | @Op return | Numeric negative |
| ! | not | Boolean | Boolean | Boolean not |
| ~ | not | Has @Op | @Op return | Bitwise not |
| ++ | nextValue() | Sequential | nextValue() return | Pre-increment L-Value |
| -- | prevValue() | Sequential | prevValue() return | Pre-decrement L-Value |

This group of operators is non-uniform in several different ways:

* There is no runtime behavior for the unary plus operator; after the compiler verifies that the type of expression a is a Number, the operator is discarded and the entire expression is replaced with the expression a.
* The pre-increment and pre-decrement operators act on L-Values, not on R-Values; this is the first expression that we have encountered that modifies an L-Value.
* The pre-increment and pre-decrement operators do not invoke an @Op method to implement the behavior; rather, the type that they require (the type of the value held by the L-Value) is simply the Sequential interface, and the compiled code causes either the nextValue() method (for ++) or the prevValue() method (for --) to be invoked in order to calculate the new value to store in the L-Value.
* The Boolean “not” operator is the same operator as the bitwise “not” operator, except that it uses the “!” symbol for readability. The Boolean “not” operator requires the type of expression a to be Boolean, and the expression yields a Boolean. (As a result, it is possible to use the “~” operator on a Boolean value, although this practice is discouraged in order to maintain readability.)

The meaning of both the “+” and “!” operators is tightly specified; the exact meaning of each other operator is determined by each class that chooses to implement the operator.

For the “-” and “~” operators, the type of the expression a must have an unambiguously single best operator defined that takes no arguments; the implicit type of the expression is the return type of that operator.

For the “++” and “--” operators, the type of the expression a must be of type Sequential; the implicit type of the expression is the return type of the nextValue() or prevValue() method, which is also of type Sequential (and by convention, is always of the type of the expression a itself).

The expression short-circuits if the expression a short-circuits.

Definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS after the expression is the VAS after a, except that for the “++” and “--” operators, the variable denoted by the L-Value expression a is marked as *not* effectively final.

These expressions group to the right, so !!!a is treated as !(!(!a)) :

*PrefixExpression:*

*PostfixExpression*

+ *PostfixExpression*

- *PostfixExpression*

! *PostfixExpression*

~ *PostfixExpression*

++ *PostfixExpression*

-- *PostfixExpression*

## PostfixExpression

Postfix expressions take quite a few different forms; there is no simple rule. The only similarity is that the expressions each begin with an expression a. As a result, each of these will be defined in their own section.

These expressions group to the left, so a.x.y.z is treated as ((a.x).y).z :

*PostfixExpression:*

*PrimaryExpression*

*PostfixExpression* ++

*PostfixExpression* --

*PostfixExpression* ( *Argumentsopt* )

*PostfixExpression* [ *ArrayIndexIndicatorsopt* ]

*PostfixExpression NoWhitespace* ?

*PostfixExpression* .&*opt Name TypeParameterTypeListopt*

*PostfixExpression* .new *TypeExpression ArgumentList*

*PostfixExpression* .is( *TypeExpression* )

*PostfixExpression* .as( *TypeExpression* )

## PostfixExpression: Post-Increment and –Decrement

These two well-known expressions use the “unary operator” forms:

a++

a--

These expressions closely mirror the rules of the pre-increment and pre-decrement expressions:

* The post-increment and post-decrement operators act on L-Values, not on R-Values.
* The post-increment and post-decrement operators do not invoke an @Op method to implement the behavior; rather, the type that they require (the type of the value held by the L-Value) is simply the Sequential interface, and the compiled code causes either the nextValue() method (for ++) or the prevValue() method (for --) to be invoked in order to calculate the new value to store in the L-Value.

The type of the expression a must be of type Sequential; the implicit type of the expression is the return type of the nextValue() or prevValue() method, which is also of type Sequential (and by convention, is always of the type of the expression a itself).

The expression short-circuits if the expression a short-circuits.

Definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS after the expression is the VAS after a, except that the variable denoted by the L-Value expression a is marked as *not* effectively final.

## PostfixExpression: Invocation

This uses well-known expression form, with a variable number of parameters:

a()

a(p0)

a(p0, p1, pn)

The type of the expression a must be either Method or Function. If the type of expression a is Method, then expression a must be a *NameExpression* indicating the name of the method[[44]](#footnote-44).

(TODO – define 3 aspects: bind target, bind args, call function, and describe requirements and resulting expression type)

The expression short-circuits if the expression a short-circuits. It is a compile-time error for an expression used as an argument to short-circuit[[45]](#footnote-45).

Definite assignment rules:

* The VAS before a is the VAS before the expression.
* The VAS before p0 is the VAS after a.
* For any *i*, where *i*>0, the VAS before p*i* is the VAS after p*i-1*.
* The VAS after the expression is the VAS after pn, (which is the VAS after a if there are no parameters).

*Arguments:*

*Argument*

*Arguments* , *Argument*

*Argument:*

*NamedArgumentopt ArgumentExpression*

*ArgumentExpression:*

?

< *TypeExpression* >?

*Expression*

*NamedArgument:*

*Name* =

## PostfixExpression: ArrayExpression

TODO a[] a[?,?] a[1,2]

## PostfixExpression: NotNullExpression

TODO a?

## PostfixExpression: dot NameExpression

TODO .name

## PostfixExpression: dot NewExpression

TODO .new …

## PostfixExpression: dot RelationalExpression

TODO .is(T) / as(T)

## PrimaryExpression

Primary expressions also take quite a few different forms. As a result, each of these will be defined in their own section.

These expressions do not group:

*PrimaryExpression:*

( *Expression* )

new *TypeExpression ArgumentList AnonClassBodyopt*

throw *TernaryExpression*

TODO *TodoFinishopt*

&*opt* construct*opt QualifiedName TypeParameterTypeListopt*

*StatementExpression*

*SwitchExpression*

*LambdaExpression*

\_

*Literal*

# Statements

A *statement* is a language construct that is used to declare various language structures, make assignments, or provide flow of control; unlike an expression, a statement does *not* have a type and does *not* yield a value. Statements include a broad set of capabilities, such as: module and class declarations, method declarations, property and variable declarations, property and variable assignments, for loops, if statements, and the return from a method or function.

Each statement that is defined by the language corresponds to a well-defined set of execution rules, which are detailed in the sections below. Each statement must be *reachable*, which means that in the course of execution, it must be possible for the execution to proceed to the statement, whether or not in reality that particular statement is ever actually executed; unless otherwise noted, it is an error for a statement to be unreachable.

Certain statements, by their nature, interrupt the normal sequential flow of statement execution, and are said to not *complete*. A statement that can complete is thus any statement that can be reasoned to allow execution to continue on to the next statement, which in turn makes the next statement reachable. The rules for statement completion are defined for each statement, but it is self-evident that a statement that cannot be reached does not complete.

Statements may also introduce *scope*. Scope represents a balanced (i.e. it both opens and closes) domain for declaration.

* Structural scope – Any statement or expression (such as a DeclarationStatement or a LambdaExpression) that results in an XVM structure (for example, a class, property, or method) represents a *structural scope*; each structure provides a scope within which further nested structures may exist. A nested structure is identified using the identity of its structural scope plus its own *local* identity, (for example, its name).
* Local variable scope – A nested scope within a block of execution that allows variables to be declared, whose names are visible only with that scope, and whose lifetimes are limited by the extent of that scope.
* A statement (or expression) can provide neither, either, or both of the above two forms of scope. A structural scope can additionally act as a capturing scope, which allows it to *capture* local variables (including this) from the containing scope.

Generally, when the term *scope* is used without specifying one of the explicit terms above, then “local variable scope” is implied.

The following sections will detail each of these statement forms:

*Statement:*

*DeclarationStatement*

*AssertStatement*

*AssignmentStatement*

*BreakStatement*

*ContinueStatement*

*DoStatement*

*ExpressionStatement*

*ForStatement*

*IfStatement*

*ImportStatement*

*LabeledStatement*

*ReturnStatement*

*StatementBlock*

*SwitchStatement*

*TryStatement*

*UsingStatement*

*VariableStatement*

*WhileStatement*

## DeclarationStatement

A declaration statement is a statement that is used to define part of an Ecstasy type system. Generally speaking, this is how classes and types and their constituent members are declared and defined in Ecstasy. There are eight different specialized forms of a class: module, package, class, const, enum, service, mixin, and interface. Constituent members include classes, properties, constants, methods, functions, and type definitions (typedefs).

A declaration statement always acts as a structural scope. A declaration statement occurring within a method body also acts as capturing scope.

Unlike other statements, a declaration statement is not required to be reachable. A declaration statement completes if and only if it is reachable.

*DeclarationStatement:*

*ModuleDeclarationStatement*

*PackageDeclarationStatement*

*ClassDeclarationStatement*

*ConstDeclarationStatement*

*EnumDeclarationStatement*

*ServiceDeclarationStatement*

*MixinDeclarationStatement*

*InterfaceDeclarationStatement*

*TypedefDeclarationStatement*

*PropertyDeclarationStatement*

*MethodDeclarationStatement*

## StatementBlock

A statement block represents a sequence of statements.

A statement block is executed by executing each statement in the statement block in sequence.

The first statement in the statement block is reachable if the statement block is reachable; each subsequent statement is reachable if the statement immediately preceding it completes. The statement block completes if every statement in the block completes.

The VAS provided to the first statement in the statement block is the VAS provided to the statement block itself. The VAS for each subsequent reachable statement is the VAS from the completion of the previous statement in the statement block. The VAS at the completion of the statement block is the VAS from the completion of the last statement in the statement block.

A statement block provides a local variable scope for the statements in the statement block.

*StatementBlock:*

{ *Statementsopt* }

*Statements:*

*Statement*

*Statements Statement*

## IfStatement

An if statement is a *conditional* statement; a conditional statement allows some portion of code to be executed depending on a particular condition. Specifically, an if statement allows the conditional execution of a block of code, zero or one times, with an optional alternative block of code (or another if statement) to execute if and only if the first block of code is *not* executed.

The execution of an if statement begins with the evaluation of the IfCondition, which yields a Boolean value, and that value controls whether the body of the statement will be executed:

* The IfCondition can be an expression. In this case, the expression must yield at least one value, and the first value must be of type Boolean, which will be used as the result of the IfCondition. Any additional values are silently discarded.
* Alternatively, the IfCondition can use a MultipleOptionalDeclaration, which defines one or more assignments that result from an expression. The expression must yield at least two values, and the first value must be of type Boolean, which will be used as the result of the IfCondition.
* In either case, if the IfCondition expression short-circuits, then False will be used as the result of the IfCondition.

This simple example illustrates several of the possibilities discussed above, and uses a simple Boolean expression for each of the two IfConditions in the two if statements:

@Inject X.io.Console console;

if (hot)

{

console.println("This porridge is too hot!");

}

else if (cold)

{

console.println("This porridge is too cold!");

}

else

{

console.println("This porridge is just right.");

}

The MultipleOptionalDeclaration is used when the expression yields values that are also necessary within the body of the if statement, as in this example, where the Boolean value yielded from an Iterator.next() call is consumed by (becomes the result of) the IfCondition, and the String value yielded from that same call is assigned to a new variable named “s”, which can then be used in the “then” body of the if statement, but is not assigned in the else body of the if statement:

void printNext(Iterator<String> iter)

{

@Inject X.io.Console console;

if (String s : iter.next())

{

console.println("next string={s}");

}

else

{

console.println("iterator is empty");

}

}

The MultipleOptionalDeclaration construct is quite flexible. Assume an interface FooBar and two variables fb1 and fb2 of that type, where fb2 is also Nullable:

interface FooBar

{

(Boolean, String, Int, Int) foo();

conditional (String, Int, Int) bar();

}

FooBar fb1;

FooBar? fb2;

The method foo() always returns four values, but the method bar() may return four values *or* a single False value, because it declares a *conditional return*. In both cases, the first value is of type Boolean, and so can be used in an if statement:

if (fb1.**foo**())

{

// ...

}

if (fb1.**bar**())

{

// ...

}

Furthermore, since each declares a total of four return values, the MultipleOptionalDeclaration allows up to three assignments from each:

if (String s, Int x, Int y : fb1.foo())

{

console.println("foo()=True, s={s}, x={x}, y={y}");

}

else

{

console.println("foo()=False, s={s}, x={x}, y={y}");

}

But in the case of a conditional return from bar(), the else clause does not have access to any of the return values beyond the first Boolean:

if (String s, Int x, Int y : fb1.**bar**())

{

console.println("bar()=True, s={s}, x={x}, y={y}");

}

else

{

// the following line will **not** compile, because bar() has

// a conditional return, and it returned False, so the

// variables s, x, and y are **not** definitely assigned:

// console.println("bar()=False, s={s}, x={x}, y={y}");

console.println("bar()=False");

}

Lastly, a short-circuiting expression will yield the same control flow as a False value for the IfCondition, so if the IfCondition can short-circuit, then the assignments implied by the MultipleOptionalDeclaration cannot be assumed to have occurred before the else clause:

if (String s, Int x, Int y : fb2**?**.foo())

{

console.println("foo()=True, s={s}, x={x}, y={y}");

}

else

{

// the following line will **not** compile, because "fb2**?**"

// is a short-circuiting expression, so the variables

// s, x, and y are **not** definitely assigned:

// console.println("foo()=False, s={s}, x={x}, y={y}");

console.println("foo()=False");

}

The IfCondition construction is also used in the while statement and the do statement.

The IfCondition is reachable if the if statement is reachable; the IfCondition completes if it is reachable and the expression completes or short-circuits. The “then” statement block is reachable if the IfCondition completes and the value of the IfCondition is not the constant False. The else clause is reachable if the IfCondition completes, and the value of the IfCondition is not the constant True; in the case that the else clause is not present, then the else clause is assumed to be reachable and assumed to complete if an else clause with an empty statement block would be reachable and would complete. The if statement completes if the “then” statement block completes or if the else clause completes.

If either the “then” statement block or the else clause (if present) is unreachable because the IfCondition is the constant value False or True, that unreachability is **not** considered to be an error. For example:

if (False)

{

// this code is unreachable, which would normally be a

// compile-time error, but in order to preserve historical

// anachronisms, this unreachable code is permitted

foo();

}

Definite assignment rules:

* The VAS before the IfCondition is the VAS before the if statement.
* The VAS before the “then” statement block is the VAST after the IfCondition.
* If the IfCondition can short-circuit, then the VAS at each possible point of short-circuiting is joined with the VASF after the IfCondition before it is provided to the else clause.
* The VAS before the else clause is the VASF after the IfCondition.
* In the case that the else clause is not present, then the VAS after the else clause is the VAS before the else clause.
* The VAS after the if statement is the VAS after the “then” statement block joined with the VAS after the else clause.

The if statement provides a local variable scope for any declarations in the IfCondition; that scope exists to the end of the if statement, which is to say that both the “then” statement block and the else clause are nested within that scope. The “then” statement block and the optional else statement block each naturally provide a local variable scope, because they are statement blocks.

*IfStatement:*

if ( *IfCondition* ) *StatementBlock ElseStatementopt*

*ElseStatement:*

else *IfStatement*

else *StatementBlock*

*IfCondition:*

*Expression*

*MultipleOptionalDeclaration* : *Expression*

*MultipleOptionalDeclaration:*

*SingleOptionalDeclaration*

*MultipleOptionalDeclaration* , *SingleOptionalDeclaration*

*SingleOptionalDeclaration:*

*Assignable*

*VariableTypeExpression Name*

*Assignable:*

*Name*

*Expression* . *Name*

*Expression ArrayIndexes*

*VariableTypeExpression:*

val

var

*TypeExpression*

## WhileStatement

A while statement is a *looping* statement; a looping statement allows some portion of code to be executed repeatedly. Specifically, a while statement allows the conditional execution of a block of code, zero or more times.

The execution of a while statement begins with the evaluation of the IfCondition, which controls whether the body of the loop will be executed. An IfCondition may yield more than one value, but the first value must be of type Boolean, and that Boolean value provides the indication of whether the body of the while statement (a statement block) will be executed: If the condition evaluates to True, then the body is executed; otherwise, if the condition evaluates to False or short-circuits, then the while statement completes. When the body completes (*if* it completes), then the while statement executes from the beginning (it loops).

Execution of a break statement that applies to the while statement causes the while statement to complete.

Execution of a continue statement that applies to the while statement causes the while statement to execute from the beginning.

A while statement labeled by a LabeledStatement provides two read-only label variables that expose the state of the loop:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| first | Boolean | True iff this is the first iteration of the loop |
| count | Int | The count of completed iterations of the loop |

These label variables are explicitly name-scoped using the label name; for example:

PrintNames:

while (String name : nameIterator.next())

{

if (!PrintNames.first)

{

console.print(", ");

}

console.print(name);

}

The IfCondition is reachable if the while statement is reachable; the IfCondition completes if it is reachable and the expression completes or short-circuits. The statement block is reachable if the IfCondition expression completes (i.e. if the IfCondition can complete without the expression short-circuiting) and the value of the IfCondition is not the constant False. The while statement completes if the IfCondition completes and is not the constant True, or if a break statement that applies to the while statement is reachable.

Definite assignment rules:

* The VAS before the IfCondition is the VAS before the while statement.
* The VAS before the statement block is the VAST after the IfCondition.
* If the IfCondition can short-circuit, then the VAS at each possible point of short-circuiting is joined with the VASF after the IfCondition.
* The VAS after the while statement is the VAS after the statement block (REVIEW) joined with the VASF after the IfCondition.

The while statement provides a local variable scope for any declarations in the IfCondition; that scope exists to the end of the while statement, which is to say that the statement block is nested within that scope. The statement block naturally provides a local variable scope, because it is a statement block.

*WhileStatement:*

while ( *IfCondition* ) *StatementBlock*

## DoStatement

The do statement is very similar to the while statement, except that it always executes its body (the statement block) at least one time.

Execution of the do statement begins with the execution of the body. After the execution of the body, the do statement evaluates the IfCondition, which controls whether the loop will be repeated. If the condition evaluates to True, then the do statement executes from the beginning (it loops); otherwise, if the condition evaluates to False or short-circuits, then the do statement completes.

Execution of a break statement that applies to the do statement causes the do statement to complete.

Execution of a continue statement that applies to the do statement causes the do statement to advance its execution to the IfCondition.

Like the while statement, a do statement labeled by a LabeledStatement provides two read-only label variables that expose the state of the loop:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| first | Boolean | True iff this is the first iteration of the loop |
| count | Int | The count of completed iterations of the loop |

The statement block is reachable if the do statement is reachable. The IfCondition is reachable if the statement block completes, or if a continue statement that applies to the do statement is reachable; the IfCondition completes if it is reachable and the expression completes or short-circuits. The do statement completes if the IfCondition completes and is not the constant True, or if a break statement that applies to the do statement is reachable.

Definite assignment rules:

* The VAS before the statement block is the VAS before the do statement.
* The VAS before the IfCondition is the VAS after the statement block.
* The VAS after the do statement is the VAS after the IfCondition.

The do statement provides a local variable scope that includes both the statement block and the IfCondition; this scope is used by the statement block **in lieu of** its own local variable scope, such that variables declared within the statement block are still be in scope when the IfCondition executes. This unusual arrangement allows the loop’s IfCondition to reference information otherwise available only within the statement block.

*DoStatement:*

do *StatementBlock* while( *IfCondition* );

## ForStatement

The for statement is generally used to loop *over* some range of values or the contents of some data structure, executing a block of code once in each loop iteration. It is, in many ways, a construct of convenience, intended to obviate the several additional lines of code that would be necessary to accomplish the same result with a while or do statement. As such, its purpose is to assist both in the writeability and readability for a number of patterns common in looping code, while reducing errors.

There are two forms of the ForStatement, based on two different forms of the ForCondition. The first is the traditional C-style for condition that has a separate initialization, test, and modification section. The second is the for-each style that specifies a range of values or a value of a *container type* whose values are to be iterated. These two forms will be explored separately, below.

*ForStatement:*

for ( *ForCondition* ) *StatementBlock*

*ForCondition:*

*VariableInitializationListopt* ; *Expressionopt* ; *VariableModificationListopt*

*MultipleOptionalDeclaration* : *Expression*

*VariableInitializationList:*

*VariableInitializer*

*VariableInitializationList* , *VariableInitializer*

*VariableInitializer:*

*VariableTypeExpressionopt Name*= *Expression*

*VariableModificationList:*

*VariableModification*

*VariableModificationList* , *VariableModification*

*VariableModification:*

*Assignment*

*Expression*

## ForStatement: C-style

The C-style for statement uses a three-part ForCondition, any part of which can be left blank:

* An initialization portion, which is performed when the statement is entered;
* A test portion, which is performed at the beginning of each iteration;
* A modification portion, which is performed at the end of each iteration.

The initialization portion is used to configure the initial state of the loop. Any number of declarations and assignments can be included in this portion; for example:

x=0, Int i=3, String s="hello", nums[0]=0, Boolean f=False

The test portion is an expression that yields a Boolean value; for example:

x<100 && s=="hello"

Lastly, the modification portion allows a number of actions to be performed at the end of each loop iteration:

++x, foo(), i \*= 2, f = !f

The above examples are absolutely atrocious in terms of readability; they are intended solely to illustrate the richness of the capabilities of the for statement. For readability, the initialization portion should only be used to declare and initialize (i) loop control variables and (ii) variables local to the loop whose state is carried over from one iteration to the next. Similarly, the modification portion should be limited, as much as possible, to modifying only loop control variables.

In general terms, this style of for statement can be re-written as a while statement:

for (A; B; C)

{

D;

}

The above example could be re-written as a while statement inside of a statement block:

{

A;

while (B)

{

D;

C;

}

}

Or, considering the possibility of a continue statement inside of the for statement, the above example would be more correctly re-written as a combination of an enclosing statement block, a labeled while statement, and a StatementExpression containing an if statement referencing a label variable:

{

A;

L: while ({

if (!L.first)

{

C;

}

return B;

})

{

D;

}

}

That alternative is ugly enough to fully validate the *raison d'être* of the C-style for statement.

Execution of the for statement begins with the execution of the VariableInitializationList, if any. Each comma-separated item in the initialization section is executed left-to-right and permitted to short-circuit; when a short-circuit occurs, execution continues with the next item. Then the test section, if specified, is evaluated: If it yields the Boolean value False, or it short-circuits, then the for statement completes; otherwise, if it yields the Boolean value True, or if no test portion is specified, the statement block body of the for statement is executed. After the execution of the body, the for statement executes the VariableModificationList, if any; like the initialization section, each comma-separated item in the modification section is evaluated left-to-right and permitted to short-circuit, and a short-circuit causes execution to continue with the next item. Finally, the for statement loops back to the test section.

Execution of a break statement that applies to the for statement causes the for statement to complete.

Execution of a continue statement that applies to the for statement causes the for statement to advance its execution to the VariableModificationList.

Like the while statement, the for statement labeled by a LabeledStatement provides two read-only label variables that expose the state of the loop:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| first | Boolean | True iff this is the first iteration of the loop |
| count | Int | The count of completed iterations of the loop |

The VariableInitializationList of VariableInitializers is reachable if the for statement is reachable. The first VariableInitializer is reachable if the VariableInitializationList is reachable. If a VariableInitializer short-circuits, then the VariableInitializer completes. Each subsequent VariableInitializer is reachable if the VariableInitializer preceding it completes. The VariableInitializationList completes if it is reachable and its last VariableInitializer completes; if there is no VariableInitializationList (i.e. if the list is empty), then the VariableInitializationList is considered to complete if it is reachable.

The test expression is reachable if the VariableInitializationList completes. If the test expression short-circuits, then the test expression completes as if it evaluated to the value False. If the test expression is absent, then the test expression completes as if it were the constant value True. The statement block body of the for statement is reachable if the test expression completes and is not the constant value False.

The VariableModificationList is reachable if the body completes or if a continue statement that applies to the for statement is reachable. The first VariableModification is reachable if the VariableModificationList is reachable. If a VariableModification short-circuits, then the VariableModification completes. Each subsequent VariableModification is reachable if the VariableModification preceding it completes. The VariableModificationList completes if it is reachable and its last VariableModification completes; if there is no VariableModificationList (i.e. if the list is empty), then the VariableModificationList is considered to complete if it is reachable.

The for statement completes if the test expression is completes and is not the constant True, or if a break statement that applies to the for statement is reachable.

Definite assignment rules:

* The VAS before the VariableInitializationList is the VAS before the for statement.
* The VAS before the first VariableInitializer is the VAS before the VariableInitializationList.
* The VAS before the each subsequent VariableInitializer is the VAS after the previous VariableInitializer.
* The VAS after the VariableInitializationList is the VAS after the last VariableInitializer. (In the case of an empty VariableInitializationList, the VAS after the VariableInitializationList is the VAS before the VariableInitializationList.)
* The VAS before the test expression is the VAS after the VariableInitializationList.
* If the test expression can short-circuit, then the VAS at each possible point of short-circuiting is joined with the VASF after the test expression.
* The VAS before the statement block is the VAST after the test expression.
* The VAS before the VariableModificationList is the VAS after the statement block.
* The VAS before the first VariableModification is the VAS before the VariableModificationList.
* The VAS before the each subsequent VariableModification is the VAS after the previous VariableModification.
* The VAS after the VariableModificationList is the VAS after the last VariableModification. (In the case of an empty VariableModificationList, the VAS after the VariableModificationList is the VAS before the VariableModificationList.)
* The VAS after the for statement is the VASF after the test expression joined with the VAS from each reachable break statement that applies to the for statement.

The for statement provides a local variable scope for any declarations in the ForCondition; that scope exists to the end of the for statement, which is to say that the statement block is nested within that scope. The statement block naturally provides a local variable scope, because it is a statement block.

## ForStatement: “for each”-style

The “for-each” style of the for statement is used to loop over a range of values, or the contents of a *container type*, such as a sequence, list, array, or map. For example:

for (String name : ["Tom", "Chris", "Caroline"])

{

console.println(name);

}

The supported types of the expression, in order of precedence, are:

* Iterator – The for statement loops through the contents of the Iterator.
* Range – The for statement loops through the contents of the Range by its Sequential ElementType (**not** using an Iterator).
* Sequence – The for statement loops through the contents of the Sequence using an Int index (**not** using an Iterator).
* Map – The for statement creates an Iterator over the keys Set (or the entries Set, if the Entry or the value thereof is required) of the Map, and then loops through the contents of the Iterator.
* Iterable – The for statement requests an Iterator from the Iterable, and then loops through the contents of the Iterator.

Additionally, if the value from any of the above is a Tuple, then the MultipleOptionalDeclaration permits assignment from the *fields* of the Tuple. Consider the following example, in which a list contains tuples of first and last names:

String format(List<Tuple<String, String>> names)

{

String result = "";

Append: for (String first, String last: names)

{

if (!Append.first)

{

result += ", ";

}

result += "{first} {last}";

}

return result;

}

The for statement operating on an Iterator or an Iterable container and labeled by a LabeledStatement provides two read-only label variables that expose the state of the loop:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| first | Boolean | True iff this is the first iteration of the loop |
| count | Int | The count of completed iterations of the loop |

The for statement operating on a Range or Sequence and labeled by a LabeledStatement provides three read-only label variables that expose the state of the loop:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| first | Boolean | True iff this is the first iteration of the loop |
| last | Boolean | True iff this is the last iteration of the loop |
| count | Int | The count of completed iterations of the loop |

The for statement operating on a Map<KeyType,ValueType> and labeled by a LabeledStatement provides three read-only label variables that expose the state of the loop:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| first | Boolean | True iff this is the first iteration of the loop |
| entry | Entry | The current Map.Entry |
| count | Int | The count of completed iterations of the loop |

The ForCondition is reachable if the for statement is reachable; the ForCondition completes if it is reachable and the expression completes or short-circuits. The statement block is reachable if the ForCondition expression completes (i.e. if the ForCondition can complete without the expression short-circuiting). The for statement completes if the ForCondition completes.

Definite assignment rules:

* The VAS before the ForCondition is the VAS before the for statement.
* The VAS before the statement block is the VAST after the ForCondition.
* If the ForCondition can short-circuit, then the VAS at each possible point of short-circuiting is joined with the VASF after the IfCondition.
* The VAS after the for statement is the VAS after the statement block (REVIEW) joined with the VASF after the ForCondition.

The for statement provides a local variable scope for any declarations in the ForCondition; that scope exists to the end of the for statement, which is to say that the statement block is nested within that scope. The statement block naturally provides a local variable scope, because it is a statement block.

*If you have built castles in the air, your work need not be lost; that is where they should be. Now put the foundations under them. – Thoreau*

1. XVM is an abbreviation for the XTC Virtual Machine [↑](#footnote-ref-1)
2. Such contracts are *software legacy*, meaning that the contracts cannot be altered without disclaiming the past; in other words, altering the contracts will break everything. [↑](#footnote-ref-2)
3. <https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/> [↑](#footnote-ref-3)
4. <https://en.wikipedia.org/wiki/Sandbox_(computer_security)> [↑](#footnote-ref-4)
5. <https://en.wikipedia.org/wiki/Privilege_escalation> [↑](#footnote-ref-5)
6. The law of conservation of mass and energy, for example. [↑](#footnote-ref-6)
7. Since this was penned, Azul Systems (azul.com) commercialized a true, non-blocking software solution for large shared-mutable stores in their Java Virtual Machine implementation. [↑](#footnote-ref-7)
8. For example, an asynchronous copy-compacting collector allows multiple physical copies to represent the same object. Each individual service can perform its own mark phase, followed by a copy-compact phase (which can be performed by any thread, including a daemon), and any memory that is no longer used can be freed by the next mark phase. [↑](#footnote-ref-8)
9. See <http://hitchhikers.wikia.com/wiki/42>, [↑](#footnote-ref-9)
10. This example is intended to be illustrative, and should not be viewed as an authoritative explanation of object orientation unless you are attempting to annoy a Smalltalk programmer. [↑](#footnote-ref-10)
11. In this case, the term *type* is being used to indicate both the interface type and the implementation class. [↑](#footnote-ref-11)
12. WebAssembly is a portable machine code for web browsers: <http://webassembly.org/> [↑](#footnote-ref-12)
13. *“Root”* in the sense that it has no dependency on any other module. [↑](#footnote-ref-13)
14. Without expanding into a formal proof, the XVM itself provides no data types (such as primitive types) from which new types can be composed; thus, the root module defines types composed only from other types in the same module, which is by definition infinitely recursive. [↑](#footnote-ref-14)
15. This specification would lack provable transitive closure without a link to Dem Bones: <https://en.wikipedia.org/wiki/Dem_Bones> [↑](#footnote-ref-15)
16. From the book “Design Patterns: Elements of Reusable Object-Oriented Software”, GoF *et al* [↑](#footnote-ref-16)
17. From the book “The Pragmatic Programmer”, by Andy Hunt and Dave Thomas [↑](#footnote-ref-17)
18. With edits, licensed using [Creative Commons Attribution 3](mailto:https://creativecommons.org/licenses/by/3.0/us/) by Steve Smith [↑](#footnote-ref-18)
19. Most notably CUDA and OpenCL for driving massively parallel graphics processors (GPUs) [↑](#footnote-ref-19)
20. “*Hiding*” simply means that these aspects must not appear either explicitly *or implicitly* in a programming language contract. [↑](#footnote-ref-20)
21. The verifier is a mechanism that validates compiled code as part of the loading and linking process. Its purpose is to verify that the rules enforced at compile-time still hold when the code is being prepared for execution, which is important since modules can be modified independently of each other, and possibly in incompatible ways. [↑](#footnote-ref-21)
22. A common example of covariance is found in Java arrays of reference types; for example, an array of String is “instance of” an array of Object, since String is “instance of” Object. [↑](#footnote-ref-22)
23. <https://www.youtube.com/watch?v=UJPdhx5zTaw&t=48s> [↑](#footnote-ref-23)
24. <http://benediktmeurer.de/2017/12/13/an-introduction-to-speculative-optimization-in-v8/> [↑](#footnote-ref-24)
25. This is a simplification: There are actually five separate this references, each with a specific purpose. [↑](#footnote-ref-25)
26. This is a simplification: A property may or may not have a field in the object’s struct; a property that does not have a field is called a *calculated* property, which is to say that a property that is calculated may not require a field. [↑](#footnote-ref-26)
27. That same machine-native value can even be used to represent an array containing that object, or a tuple containing that object, or a function returning that object, all by varying the implied type in a reference, which itself may not even exist. The possibilities are mind-bending. [↑](#footnote-ref-27)
28. For a property or variable v of type T, the expression &v obtains a reference to the variable itself, which is of type Var<T> (or just Ref<T> if the property is read-only). The read/write interface Var extends the read-only interface Ref. [↑](#footnote-ref-28)
29. XTC is an abbreviation for XVM Translatable Code, and is also known as “Ecstasy” [↑](#footnote-ref-29)
30. Many languages allow an octal literal value to be specified by simply starting a value with ‘0’; this is one of the archaic constructs that Ecstasy does not support. A sequence of digits beginning with 0 is treated as base-10. [↑](#footnote-ref-30)
31. It is expected that these rules will be modified to allow the use of the standardized set of common emojis; see <https://unicode.org/emoji/charts/full-emoji-list.html> [↑](#footnote-ref-31)
32. Vapid Acronym Syndrome: While the use of acronyms is generally avoided, this one will be used extensively enough that the shortening to an ugly acronym will be well worth it. [↑](#footnote-ref-32)
33. Quite often, when the arity of the expression is not relevant to the point being discussed, the singular form will be used, instead of always saying “type or types”. [↑](#footnote-ref-33)
34. enum Nullable {Null} [↑](#footnote-ref-34)
35. This is traditionally called “short circuiting boolean logic”, but that term is avoided to reduce confusion with respect to the concept of “short-circuiting expressions” already introduced. [↑](#footnote-ref-35)
36. This is traditionally called “short circuiting boolean logic”, but that term is avoided to reduce confusion with respect to the concept of “short-circuiting expressions” already introduced. [↑](#footnote-ref-36)
37. An operator is a method annotated by @Op. Either the method name must match the default name for the operator method (in this case “or”), or the @Op annotation requires a String parameter for the operator itself (in this case “|”). [↑](#footnote-ref-37)
38. The complexity increases for annotated types and relational types. [↑](#footnote-ref-38)
39. Since the root Object class declares an equals() function, all types support comparison for purpose of equality. [↑](#footnote-ref-39)
40. static <CTT extends *This*> Ordered compare(CTT v1, CTT v2) [↑](#footnote-ref-40)
41. enum Ordered {Lesser, Equal, Greater} [↑](#footnote-ref-41)
42. For various self-explanatory reasons, it is unlikely to be used in this manner. [↑](#footnote-ref-42)
43. While you can easily implement operators on your own classes, you should not even consider doing so until you fully understand why it is a very bad thing to do. The full explanation will be included in the next version of this document. [↑](#footnote-ref-43)
44. Invocation of a method via a *reference* of type Method is not supported by the language; it is supported instead by the API of the Method class itself. [↑](#footnote-ref-44)
45. Short-circuiting is meant to improve readability and comprehensibility. Being able to short-circuit (prevent) an invocation *from within the argument list* accomplishes just the opposite. [↑](#footnote-ref-45)