TONAL: ThoughT ON A Language

Or, The Sound Principles of Artificial Philosophy

Contents

List of Figures

List of Tables

Acknowledgements

Lisa Lippincott

Alexander Stepanov

Bjarne Stroustrup

Sean Parent

Part I

Introduction

Linguistics, botanics, physics, musics. Science is what is left once we take into account all the ways one can be wrong.

1 Motivation

The TONAL programming language tries to answer the question - what if a language aims from the start to have compile-time programming and generic programming as its primary focus?

Why focus on compile-time and generic programming?

Performance is important now, more than ever, as software exceeds the capability of hardware. Programmers of the past wrote amazing programs on, and for, machines that are less powerful than today's phones, but they did so taking a lot of unsafe and unportable and unexplainable shortcuts. Many modern languages trade performance for portable safety and end up showing that it is impossible to put that toothpaste back into the tube. As soon as a language gains a slow feature, it gets used, depended on, and thus becomes impossible to rollback.

Primarily focusing on compile-time and generic programming is to force the design of the language to properly solve the issue of high-level programming, as it relates to performance, rather than just kicking the can down the street. Languages that add compile-time and generic programming as an afterthought (or for backwards compatibility reasons) get the worst of all worlds. To that end, TONAL aims to make compile-time and generic programming so primary, that it is indistinguishable from normal programming.

Otherwise, if compile-time and generic programming is too hard to use, programmers will find performance is too hard to consider as default. Compile-time and generic programming is required for performance because it is simply faster to do whatever is known at compile-time than waiting to run the final program.

Good performance is good usability. Good usability for compile-time and generic programming is good performance. TONAL takes after C++ in its focus on reducing unnecessary overhead from the machine, but without the constraints of C preventing a simpler syntax.

On the topic of usability, TONAL is not motivated by pet peeves. Very often, a new language has to reinvent bits of syntax - new ways of writing loops, or printing a string, seemingly because the language designers dislike minor things about the languages they've used. They're usually ad-hoc and special compiler only constructs that can't be customized. On the other side of the spectrum, like LISP, every high level feature is created by macros on top of compiler features, giving a programmer any option. Languages like C++ introduce features like range-for loops and structured bindings with hooks that allow the programmer to work their own types into the language.

TONAL takes after LISP in only having a small number of primitive forms, but takes after C++ in that the primitive forms are complete enough - and customizable enough - to cover most code structures without requiring macros. To achieve a look that doesn't feel ad-hoc, the syntax is focused such that the advanced constructs - like compile-time and generic programming - are not distinguishable from the simple constructs. That means forgoing the use of most special characters commonly used in programming languages, because experience in C++ shows that having a lot of special characters makes it harder to write generic code that fits together syntactically. The focus on compile-time programming means macros are not needed, because generic function calls are not expensive at compile-time.

In the early experimentation of TONAL, from the syntax reduction emerged the classification of syntactical elements into the scale degrees of music, particle physics and basic grammar. Anyone who has had trouble remembering the differences between declarations, definitions, statements, expressions; parameters, arguments; would appreciate the difficulty of trying to talk about code, let alone teach it. While teachability wasn't an initial goal of TONAL, the effort to make performance, compile-time and generic programming ergonomic, reducing syntax and special characters, made it possible to pair syntax elements with corresponding systematic names.

TONAL is suited to novices and experts alike. Rather than forced simplicity of some languages aimed at novices, that then requires experts to work around limitations, it makes advanced language constructs of other languages into regular constructs. TONAL aims to make novices into experts by providing a gentler ramp to advanced concepts. Novices do not remain novices for long, but anaemic language constructs stay around forever, so there's no point in reducing the power of a language.

A unified syntax for novices and experts means that rewriting of code is reduced. There isn't a massive syntactical change from concrete, tweaked code to abstract code to take advantage of performance or compile-time or generic programming. As novices become experts, their old code doesn't need to be discarded, but similarly as experts become experts in more language features old and new, their effort isn't wasted. Novice code should look like expert code should look like new paradigm code.

This lends itself well to backwards compatibility, which is required for any serious language. People are happy to break other people's code, but not happy when their own code breaks. No language can maintain a critical mass of users if they leave due to regular breakage, so a language must not break anyone's code for a longer amount of time than the coding style is used.

2 Influence

It must be said up front that TONAL is influenced most by C++. Syntax doesn't matter, but the mindset. The language is shaped by experience. C++ is not the best language, but it gets the job done, and the jobs run faster. It doesn't break your code just because a lot of vocal people hate the features you depend on. It doesn't add new features because they're popular at the time; mostly features with a demonstrable benefit gets added - as libraries if possible. It doesn't mandate features that are not supported by hardware, such as garbage collection, various co-processing units, even tiered memory architectures.

C++ also represents decades of learning. Any fool can create a new language (eg, TONAL), but there is a treasure trove of knowledge gained from practical, industrial experience that would be foolish to ignore. Plenty of languages and their designers think that merely having a new language with slightly different syntaxes for printing, or for-loops, or declaring variables, will somehow avoid all the problems that plague any programming language. C++ is a living document that is a testament to all its successes, failures, inventions, imports, compromises and dead ends, and in that TONAL should regard itself as inheriting all those lessons.

TONAL should also learn from C++ its collaborative, disciplined approach to features being added, deprecated, removed, modified or re-specified. Despite the accusations of design by committee, the C++ committees, composed of various Working Groups that are charged with looking into broad areas of focus, only consider proposals coming from the community, they do not design proposals. There may be many competing proposals, but ultimately the one that is selected is one that plays well with the rest of the language.

C++ is an international standard, written in English, with multiple independent implementations, with no reference implementation owned by a single entity. TONAL aims to be vendor neutral, but taking lessons from the C++ standardization experience, TONAL should ultimately be defined by a comprehensive, open, collaborative test-suite that tests for conformance as well as non-conformance. English standards are imprecise compared to test-suites and are often bogged down with incomprehensible descriptions. The test-suite would need to be delicately curated to avoid standardizing quirks of an implementation. Certain implementors may think they can monopolize the language by standardizing their implementation, but that actually works against them as they would be standardizing a quirk of their implementation that they will then be forced to support in the duty of backwards compatibility.

One of the major influences of C++, and inherited by TONAL, are the principles espoused by Elements of Programming by Alexander Stepanov and Paul McJones, and From Mathematics To Generic Programming by Alexander Stepanov and Daniel Rose. Every language claims to be derived from fundamental unifying principles. Unfortunately programs have to model the real world and the real world is not easily reduced, especially not into abstract ideas. Such languages eventually break from their clean derivations and introduce widely used concepts in an ad-hoc way. Even if a language is completely consistent to its core, it still remains the challenge of programmers to build their programs from the core principles of the problem domain. This may or may not be compatible with the language's core principles; for example, not everything fits into one strict type hierarchy. Instead, programmers often/should do what mathematicians do - start with concrete, specialized prototypes/examples and then refactor (by naming, splitting up, extracting) and generalize into a solution composed of parts that fit well together, but can be useful for other problems and problem domains.

Instead of forcing programmers to adhere to some object model and hierarchy, or worse yet, a framework, TONAL takes the lessons of abstract algebra to help programmers do what they already do. Abstract algebra is about what notation means and how it behaves under its own rules; programmers, of any language, are really just creating specific notation. Whether that notation is implemented as objects, functions, design patterns, idiomatic code - without special syntax - ideas from abstract algebra gives a way to uniformly encode common rules about how that notation behaves. The programmer is free to decide how to implement their notation because TONAL does not enforce any particular implementation as the one paradigm to rule them all and be confident that the rest of the language, and indeed 3rd party libraries, can easily interoperate through the ideas of abstract algebra.

Programming languages are trending towards more functional styles, because they have demonstrated, tangible benefits. Many logic bugs do arise from unexpected side-effects, and functional languages on the whole do not modify values, but create new values from given arguments. In the same vein, TONAL follows suit with the default immutability of functions. Functional languages also treat all functions as first class, in contrast to languages like C++ where free functions are not the same as capturing lambdas (implicit function objects) or full-fledged function objects. It creates a friction to writing generic code that needs to treat all function-like things the same. TONAL follows the functional style by not differentiating between free functions and (sometimes stateful) function objects. That is the minimal language support needed (or rather, it gets out of the way), and from there, functional composition can be implemented in libraries with no cost.

Functional programming leads very well into declarative style programming languages. LISP and XML use S-expressions. S-expressions merely express the structure of a program, and the program processors are then free to traverse that structure and interpret the meaning and apply transformations eg, code generation, macro expansion, optimizations. C++ in fact can also use S-expressions, a technique called expression templates, in order to encode computations for lazy evaluation. That allows for whole expression factorizations like that in maths which then leads to optimization opportunities that may even take into account hardware capabilities.

Declarative style is all about intent, which is more about what code means to us than what it means to the machine. Intention matters the most in the frontend of the language which is where the high level constructs impose conceptual structure - structure that doesn’t actually exist in the bits and bytes of computer architecture. Intent allows a programming language to understand what a program is supposed to mean, and not just a series of instructions to blindly follow. High level constructs that reflect intent can be eliminated from the eventual program due to guarantees about program behaviour that cannot be inferred from low level code. For example, resumable functions - sometimes called coroutines in other languages - can be optimized away, in contrast to manual jumps and state-machine management (often achieved with messy macros) to emulate a resumable function that the language has no idea about.

In recent years, language design has placed a lot more emphasis on defaults. When it comes to intention, the non-specification of an intention says just as much as specifying an intention. C++ is mutable by default, unless specially marked; Rust is immutable by default unless specially marked. Both have their reasons - C++ maintains semantic backward compatibility with C, whereas Rust recognizes that it is often safer to make things immutable and so doesn't make the programmer go out of their way to specify the common option. Defaults represent the common intention. Like C++ but without the C backwards compatibility (or in fact C++ backwards compatibility), defaults are chosen that can be implemented with zero overhead, or can be eliminated when given compile-time information. As it so happens, this also means the defaults are those which are safe, because zero overhead can only be achieved with compile-time information; an understanding - by the compiler - of the high level guarantees of a feature.

Query languages are prime examples of languages all about intent - what we want to find, but not how we find. XQuery FLWOR expressions are something that influences TONAL as a design goal. The language should be powerful enough to build up to things like FLWOR, LINQ. SNOBOL4 pattern matching is the other major declarative influence on TONAL. Regular expressions are not powerful enough - hampered by heavy use of symbol characters, and there are too many different flavours of

BNF, while SNOBOL4 has only one de-facto standard. Both XQuery FLWOR and SNOBOL4 pattern matching have the benefit of being embedded in an actual programming language, so we know that there isn't much of a barrier to implementing it as part of a programming language instead of relying on external processors to generate code.

Perhaps the most surprising influence on TONAL is the graphical formal specification language DRAKON. Ergonomics is an important aspect of DRAKON that is not formalized in other graphical languages. C++ has an uncontested reputation for having difficult syntax. LISP syntax is, on the surface, very simple, but heavy reliance on macros, many different ways to declare variables, and a proliferation of mini-languages. And of course, graphical languages like traditional flowcharts, SDL, UML, and most others, have been unwieldy in never really being able to capture the fluid nature of software development without being inundated with large criss-crossing diagrams with many graphical primitives. DRAKON has relatively few primitives and a small set of powerful layout rules that make it easy to create readable diagrams. In a similar way, TONAL avoids the extremes of LISP and C++ and chooses primitives that covers every aspect of successful languages with minimal syntax.

Ergonomics is also a concern when allowing for third party tooling. The simpler a language is to parse, especially without the text transformations hidden by macros, the easier it is for anyone to whip up a quick tool to get whatever information they need from source. It would also make it easier for implementations to ship tools when there is a common, non-technical, vocabulary that captures all aspects of the language.

3 Scope

An important question for all implementations of programming languages is: what should be left out?

Can we afford to leave it out? Can we find a way of bringing it back in later?

What can't we bring back later? What can't we afford to leave out?

As was discussed in Motivation on page 12, performance is something that is hard, if not impossible, to get back once it is lost. Backwards compatibility requirements means that languages cannot change slow features that are widely used, and even if they eventually get changed, existing systems can't always simply be recompiled and redeployed without extensive testing. We can't bring performance back later.

Influenced by C and C++, a was discussed in Influence on page 15, TONAL must take into account, and take advantage of, the reality of physical machines. Machines have size limits, and size limits affects performance, whether it is data size, or code size. The smallest size is zero. Any code that can be executed at compile-time only can be discarded. Any temporary data that can be computed at compile-time can be discarded. Information that is lost to the run-time cannot be recovered, such as the exact type and therefore exact sizes, mustn't be left to run-time dispatch just for the sake of polymorphism. We can't afford to leave out any generic programming.

Data that we can't discard must only use the data size that is necessary, which may not always be the smallest if there are performance gains to be had for alignment correctness. Therefore, as a base for the rest of the language, we must have a rich library of machine types. There are many different types of machines, now and into the future, but if we have compile-time programming, we can cater to these differences with normal programming. We can't afford to leave out compile-time programming with machine types.

Machines have memory, but they don't collect memory that is left as garbage that programs generate. They don't even have the concept of garbage memory. Any notion of used and unused memory is something that is tracked by a language for correctness. The only way a language can keep track of when memory can be reclaimed is for the programmer to tell it. We can't afford to leave out ways for a language to automatically track memory at compile-time; a way for a programmer to tell the language without having to do so manually. We can afford to bring back later a separate library that keeps track of memory at run-time. By necessity it must be built upon automatically tracked memory - there is no way to make the converse cheaper.

All machines have support for basic control flow for jumping; conditional and not. Experience shows us, however, structured forms of jumping, like loops, branches, and named jumps, local and non-local, gives a language more information about the intent of a program, and so more opportunities to run things at compile-time. So we can't afford to leave out the basic traditional structured control flow, but we also can't afford to leave out other control flows that, while requiring a lot of code generation in most instances, can be eliminated if executed only at compile-time.

More expensive machines have hardware threads, multiple cores, separate processing units, or working in a cluster. They are not always available and involve unpredictable flows, so there aren't many, if at all, meaningful structures we can reason about at compile-time. These are things that can be added back later as run-time libraries, but the language itself can at least define a consistent way to use these platform specific features. This is where compile-time programming facilities help because platform features can be enabled, selected, or removed at compile-time without special syntax.

Any high level features that can be implemented with the language - ie, in language libraries - in a fashion that allows use at compile-time should be implemented. In a way, we can't afford to leave them out, as they can build on each other at no cost and so a lot of advanced features come for free. If they were to be left to be implemented as an afterthought, we would again face the issue of trying to claw back performance in ways that break backwards compatibility. It is better to have these features at compile-time, even if their run-time performance is slightly slower, because compile-time means no run-time cost, and tuning for run-time will always be required anyway.

The features that require operating system support can be added back later as run-time libraries as they cannot be purely implemented with the language but require special compiler support.

While influenced by LISP and functional styles, as was discussed in Influence on page 15, nevertheless TONAL does not share the same obsession with any single concept - macros, purity, call-with-cc, monads. TONAL hedges its bets on compile-time and generic programming and a syntax that is unified for both that has all the necessary power to achieve what all those features are capable of. For example, macros are used to re-order source code without having to evaluate it at the point where the macro is invoked, in order to allow the creation of domain-specific languages. With compile-time and generic programming, something akin to new syntax can be created with compile-time values at no cost to run-time performance. We can afford to leave out macros, and we can afford not bringing them back later.

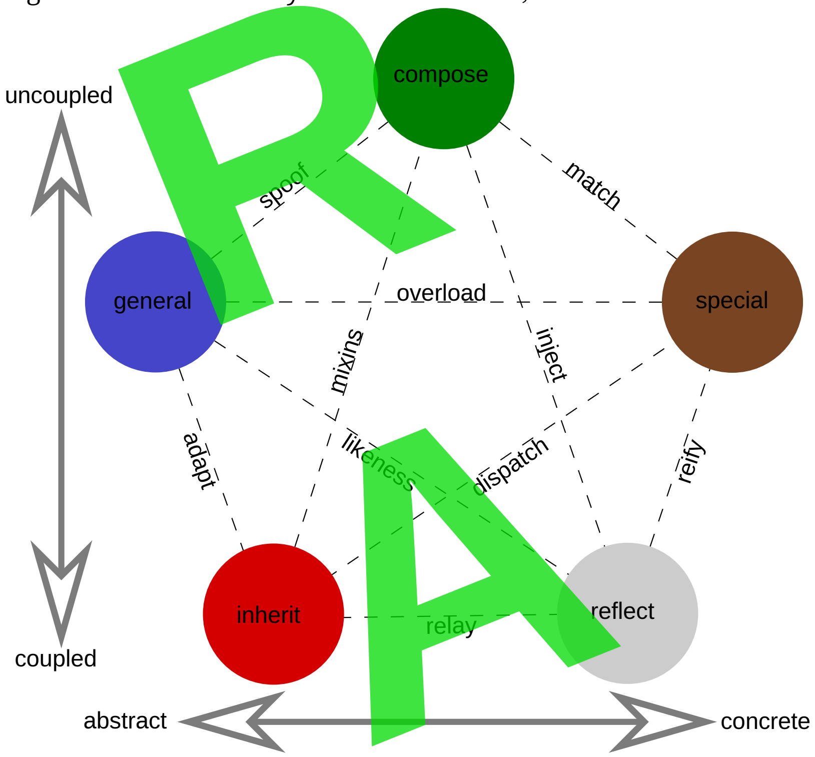
Obsession with purity of any form is the cause of many of the world's greatest atrocities, so TONAL does not partake in that obsession either. Modifications of things in-place is a fact of life. While reducing side-effects has real benefits in reducing errors and even improve performance, the hoops to achieve statefulness continues to be a major turn-off for functional styles of programming. Programmers see all the effort to use statefulness in a functional style and see imperative style languages need no effort to do the same; to do the thing that programs, by definition, need to do the most often.

We cannot afford to leave out statefulness, but at the same also cannot afford to leave out functional styles. Most code should be written in a functional style - immutability by majority (not default), and composition of functions to make higher-order functions. Stateful programs have a whole host of other issues, such as aliasing, concurrent modification, ownership transfer. TONAL solves them with explicit marking of mutability changes, which is enabled by compile-time programming. Mutability is fundamental to a language, so is something that must be implemented at the compiler level, while functional composition is better served with compile-time generic libraries, as the TONAL language itself is designed to enable such flexibility.

The experience of C and C++ compared to other languages shows that a proper module system needs to be designed from the beginning. Compile-time and generic programming introduces complications for modules since a lot of information needs to be known across a whole program to take advantage of the flexibility and performance. The separation between modules needs a way to share information between them without the high coupling of textual substitution. One such solution would be to generate intermediate files and caches. On top of that, traditional build systems operate on text files with no knowledge the semantics of the files and special support is required for any language specific intermediate files. At the very least, we can't afford to leave out modules, because they set the limitations of the language via the requirements of implementation. Build system support we can bring back later, as it would depend on how modules could be implemented.

4 TONAL Pentadynamics

Figure 4.1: The Pentadynamics of TONAL, and their interactions.



[1]Software engineering can be described as managing complexity by divide-and-conquer. Many techniques, syntaxes, and philosophies have been invented to solve the divide-and-conquer problem. Many claim to provide the ultimate paradigm to the divide-and-conquer problem.

Languages traditionally classify capabilities as belonging to a paradigm, like “object oriented”, or “functional”, or “procedural”, or indeed “multi-paradigm”. The Pentadynamics do not constitute a paradigm, but rather highlights the main organizational forces pulling on programmers when

[1] This section uses metaphors and real-world analogies in order to get readers to think about how each dynamic is materialized, rather than how they've been traditionally implemented in compilers or interpreters. This section is about the dynamics and should be applicable to experts and novices alike, so it should avoid getting too deep into those implementation details. It also prevents the language being tied down to any single interpretion of the dynamics, and leave room for them to be implemented in many ways, and of course be interpreted to include different features or different usages of features.

dividing and conquering. The Pentadynamics also interact with each other as divide-and-conquer strategies sometimes complement each other, and other times at-odds.

User requirements and the problem space pulls on the design of a solution in multiple directions. Therefore divide-and-conquer strategies do not belong to any single Pentadynamic, with each Pentadynamic naming a general direction that a design is pulled towards, rather than strictly defining a category with hard boundaries. These are highly overlapping magisteria.

Language features are usually invented ad-hoc to solve a real, but limited, coding scenario, and thus are permeated with all of the Pentadynamics to varying degrees. This is opposed to trying to invent features based on theoretical concerns; they fit neatly within categories but are useless in real code. Real code can be categorized multiple ways at the same time, and can even change nature of its categorization according to its interactions with, say, development practices.

Compose - music and biology are endlessly complex constructions. Limited rules operating on self-contained units can build up symphonies and ecosystems. The most uncoupled systems are compositions of non-overlapping, non-communicating processes, each doing their own thing in their own time, but are useless. Achieving objectives always requires some kind of coordination between parts. On its own, it is equally used in abstract, and concrete ways; much like music is based on abstract theory, and biology deals with the tangible.

General - water has many different uses, but comparatively few ways to contain it, or transport it. Water vessels either have a bowl or a hole, and made of non-porous materials; the designs of which are also applicable to other fluids because it is possible to generalize on common attributes. Rivers become pipes; lakes become bowls. Generalization blurs the sharp realism into smooth abstract forms; their surfaces can join together without too much interlocking and becoming stuck.

Special - a lump of rock can be chipped and carved, clay can be moulded; specific tools forged for specific tasks. Material concrete is the metaphor by which specialized components in software are described. Tools need to be reusable - or at least their design - must be reusable, even if a tool is destroyed in its use. Tools cannot be so intertwined in the final product itself as to become part of the product, by definition, otherwise they're just components of the product. Even so, such components need not be melded.

Inherit - fire burns with oxygen; energetic oxygen chemistry made eukaryotic life, and multicellular life, possible[2]. The combined abilities of the mitochondria ancestor and an as-yet-undiscovered prokaryote ancestor[3] gave rise to great and robust diverse physical forms. Inheritance is not merely concrete physical reproduction, but the abstract intertwining of information itself. Inherited structures can be highly irreducibly complex; very difficult to decouple[4].

Reflect - mirrors are used to look at things from different angles, including the self. The greatest power comes from being able to change oneself; changing oneself requires being able to look at the self. Changing the self without discipline or principle leads to fragile development. A strong foundation guides development of a well-defined self but with flexibility and dynamism over time.

Composing generalizations - the boundaries where compositions happen define a set of general expected behaviours. Any components that can spoof those behaviours can be used at those boundaries.

Composing specializations - where special cases exist in general behaviourial requirements, special components can exploit acute knowledge for better performance. Special components can be automatically

[2] https://doi.org/10.1073/pnas.96.23.12971

[3] And speculatively, possible separate ancestry of other organelles.

[4] https://www.talkorigins.org/faqs/comdesc/ICsilly.html

matched, implicitly or explicitly, to special cases.

Inheriting generalizations - generalizations can be formalized to allow indirect fulfillment of the requirements. Adaptor specifications can be specified by component interfaces and stable protocols, and implemented by third party plugins, just like how peripheral hardware can all be adapted to a USB interface.

Reflecting specializations - mathematical identities are static patterns for transforming from one truth to another truth, without error or loss. Upon application of identities on dynamic problem spaces, the patterns are reified into rules and strategies for finding better fitting solutions.

Generalization and specialization - unified by overloading. One name can be used to establish a relation between multiple incarnations of the same underlying notion. The name generalizes a notion, while parameterization specializes.

Inheritance and reflection - unified by structure relaying. Many facades can reuse a structure, especially if they're related. Once a facade is obsolete, the subsequent facade can be re-layed on the structure. The structure is relayed from one stage to another: the runners change, but the baton remains the same.

Composing inheritance - entities in the real world fit many categories; exhibits many disjoint and blended behaviours. Categories and behaviours be can be mixin to the one object in order to be used in many different contexts just like real objects can.

Composing reflectively - components, by and large, do not assemble themselves. Components define their internal structures, and what is needed to create those structures, but materials come from the outside. Entities build components from looking at a detailed plan, source the needed parts, and inject them into correct positions.

Generalize reflections - images of static objects can reproduce great amounts of detail, devoid of motion blur. Such images have limited information of how the object moves or its how parts work together. In many problems we are not looking for exact replicas, but something of sufficient likeness of how an object behaves.

Specialize inheritance - most knowledge comes from the future, so not all paths can be set in stone in the present. Paths can, however, be preordained for limited categories of known possible cases. When actual cases are encountered in the future and classified, they are dispatched to the path most specialized for their categories.

Part II

Syntax

TONAL can be thought of as a language to control a program called a compiler. A TONAL compiler can be thought of as an interpreter that processes TONAL commands to execute instructions and produces a program as output. TONAL commands tells the compiler to do things such as define a name as a type; define a name as a function inside that type; define the parameters and the command sequence of that function; and so on and so forth. TONAL does not make a distinction, or give nomenclature to, statements, compound statements, declarations, expressions, definitions, lvalues, rvalues, glvalues, prvalues, xvalues. TONAL commands are all there is, whether they command the compiler to make a definition, or to invoke the definition. There is no essential difference in terms of syntax.

5 Archetypes

Table 1: *All TONAL Archetypes*

Flavah	**Generation |**	**Generation ¦**
Scalah	long	real
Vectah	atom/qtom	list
Tensah	func	type

TONAL programs are structured around six Archetypes. They are called Archetypes because terms like fundamental datatypes only typically cover types that fit into registers. TONAL instead has the concept of Archetypes because every thing in TONAL patterns after them. Funcs and Types themselves can now fit into scheme without the extra requirement of being machine representable. They are, after all, high level constructs to create meaning where there isn't any - they're purely for us as programmers.

Archetypes do not imply any derivation hierarchy. They are merely archetypical of every construct in TONAL. There is no common object that everything inherits from, because it doesn't make sense and is unnecessary. It would just be an attempt to artificially shoehorn everything into one thing because it feels neat and tidy but doesn't provide any real benefit. Archetypes are unified in one aspect, in that they are all treated as things that a programmer can manipulate, instead of a subset being things and another subset being merely descriptions of things.

One way to think about the Archetypes is along two lines of generalization - flavah and generation. They are a bit reminiscent of the standard model of physics, which is a nice coincidence to help with memorization recall.

Scalah Archetypes are dimensionless values. They are single units and the most fundamental of the Archetypes.

Vectah Archetypes are one dimensional sequence of values, which are themselves values that can be put into other values. They generalize Scalah Archetypes by extending what can be represented through permuting Scalah value sequences.

Tensah Archetypes are those for which we define their meaning and build upon other Archetypes to create new value structures. They generalize Scalah and Vectah Archetypes by specifying custom behaviours and representations that are not as fixed as those Archetypes.

Generation I Archetypes have a discrete domain; either a limited set of values or functionality. Like fundamental particles, they are the lightest of their flavour.

Generation II Archetypes have, ostensibly, a continuous domain (subject to machine representation, of course); a continuous set of values or open ended possibilities of functionality and representation. They are conceptually heavier than Generation II Archetypes.

The Long Archetype represents integer numbers, as in the Z set. It holds at least base 2, 64-bit 2s complement representable numbers.

The Real Archetype represents real numbers, as in the R set. It holds at least IEEE-754 binary64 representable numbers.

The Atom Archetype are identifiers of length greater than zero used for naming things and referring to named things. The Qtom Archetype is a quoted Atom, representing text of any length, containing any character from at least Unicode 11.

The List Archetype represent syntactic lists of any length containing and/or referencing any value. There is no physical storage or ownership of values; it is merely as if the sequence of values were written by the programmer each time. Lists are themselves values. The values in a list may be future-tense.

The Func Archetype defines subprograms that can be invoked with zero or more parameters; may grab zero or more let-name views from its environment; may be suspended and resumed; may trip conditions.

The Type Archetype defines a permutation of sets of values, their invariants and valid operations; may grab zero or more let-name views from its environment; may be derived from other types. Every type and Archetype has a type.

It may seem strange that a machine-oriented language would omit the mainstays of other machine-oriented language, like pointers, enums, bitfields and unions. Perhaps they can even be placed into something like a Generation III Archetype. But that would be going too overboard. There is no argument that they are essential to writing code that squeezes more performance when required. TONAL makes the case, though, that due to their highly machine-specific nature, they are better implemented on top of TONAL to take advantage of compile-time programming, which will then make explicitly available all the machine-details so that physical design decisions can be made programmatically.

Consider pointers. Memory architectures have not be flat for a long time, not counting the unused segmented memory feature of certain processors. There are variations in cache architecture; the availability of coprocessors like GPUs; expansion peripherals; networked distributed systems. Direct memory access over all these systems is thus a much more complicated affair. Even operating-system provisions like memory-mapping makes it harder to pretend to have a flat address space. Safety issues with pointers is well known - most commonly out-of-bounds access in various guises (including pointers not derived from initialized memory), and leaks. Garbage collection may also come into play.

All of these issues argues in favour of a more rich abstraction over pointers, but without unnecessary overhead, so that programmers can actually programmatically encode and handle these differences instead of relying on documentation, compliance, and imprecise checkers. Exposing some of these details allows programmers to programmatically make design decisions taking into account correctness and flexibility without sacrificing performance.

Bitfields and enums can actually be implemented as library solutions now that we have a language in TONAL that prioritizes undifferentiated compile-time generic programming. Unions are more of a challenge, but again, with undifferentiated compile-time generic programming, some options are open that were not available with native unions, such as compile-time tracked unions in addtion to traditionally tagged unions.

Archetypes are how TONAL realizes undifferentiated compile-time generic programming. Archetypes behave just like any other object, but a TONAL compiler knows the value of Archetypes. All Archetypes must have values known at compile-time, but values of other types can also be used by the compiler for undifferentiated compile-time generic programming if they can be converted to and from Archetype values. Archetypes makes the undifferentiated possible by using value semantics instead of using syntax and type-system tricks.

6 Literals

Literals are the syntax for introducing Archetype values into TONAL programs. Literals are contiguous sequences of characters that have no syntactic substructure that can be edited without changing its value (other than padding zeros for numbers).

The Literal Archetypes are Long, Real, and A/Qtom. In TONAl, space characters are reserved for separating individual tokens of source. Therefore only Qtoms are an exception, and can contain any space character. Quotes, apostrophes and parenthesis are also considered to be separating characters.

Figure 6.1: Railroad diagrams for subtokens common to Real and Long Literals.

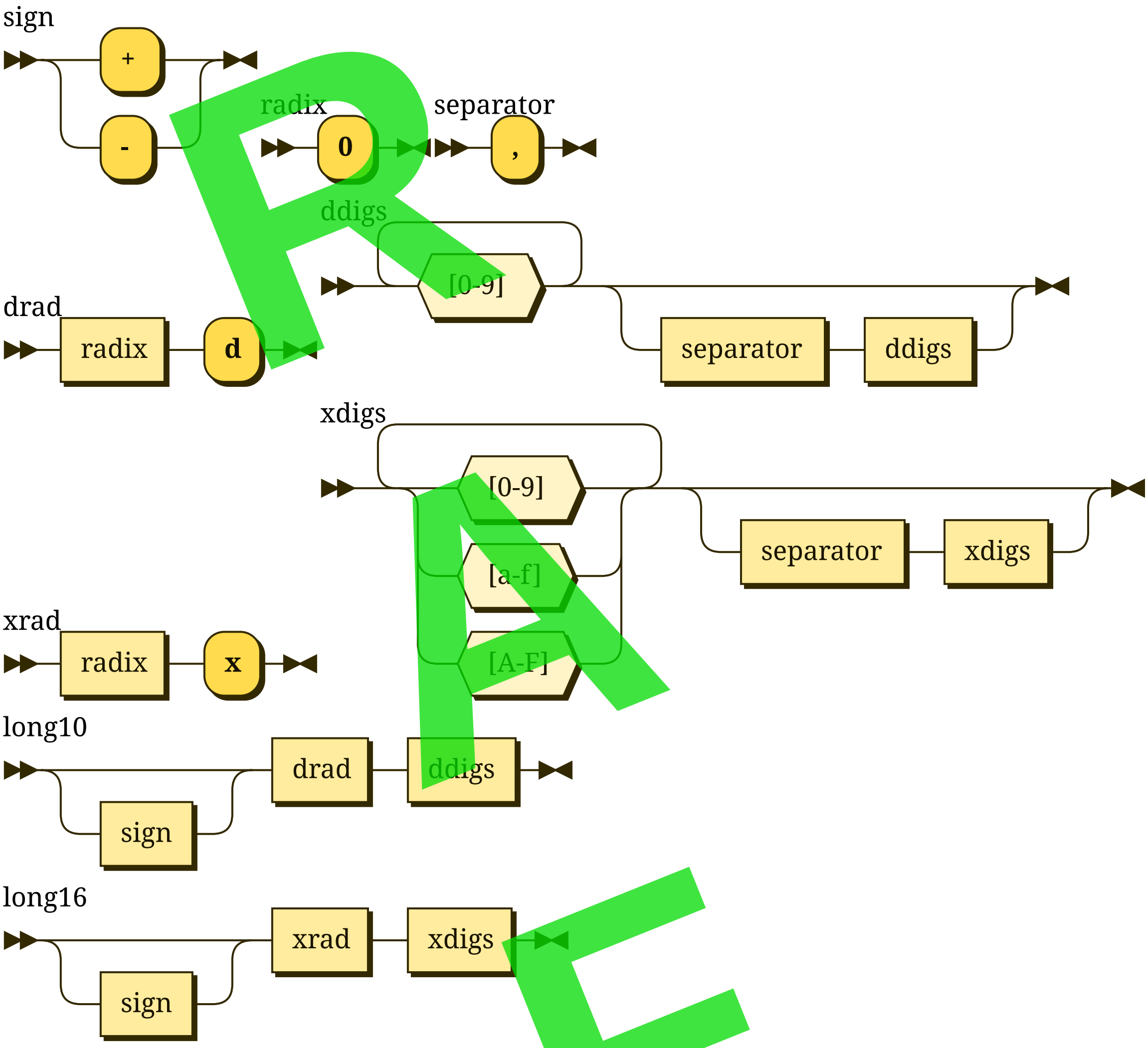


Figure 6.2: Long Literal railroad diagram.

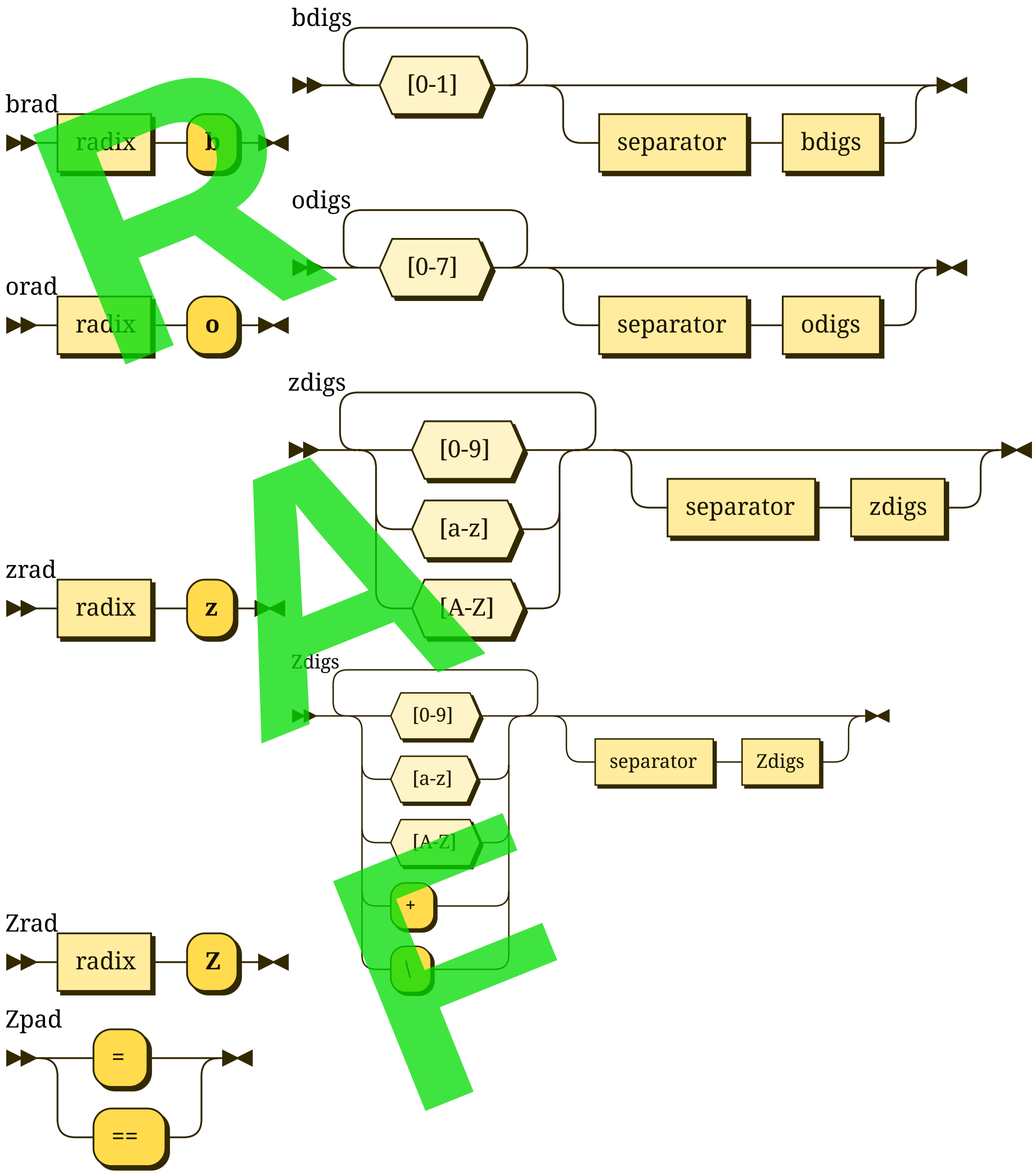
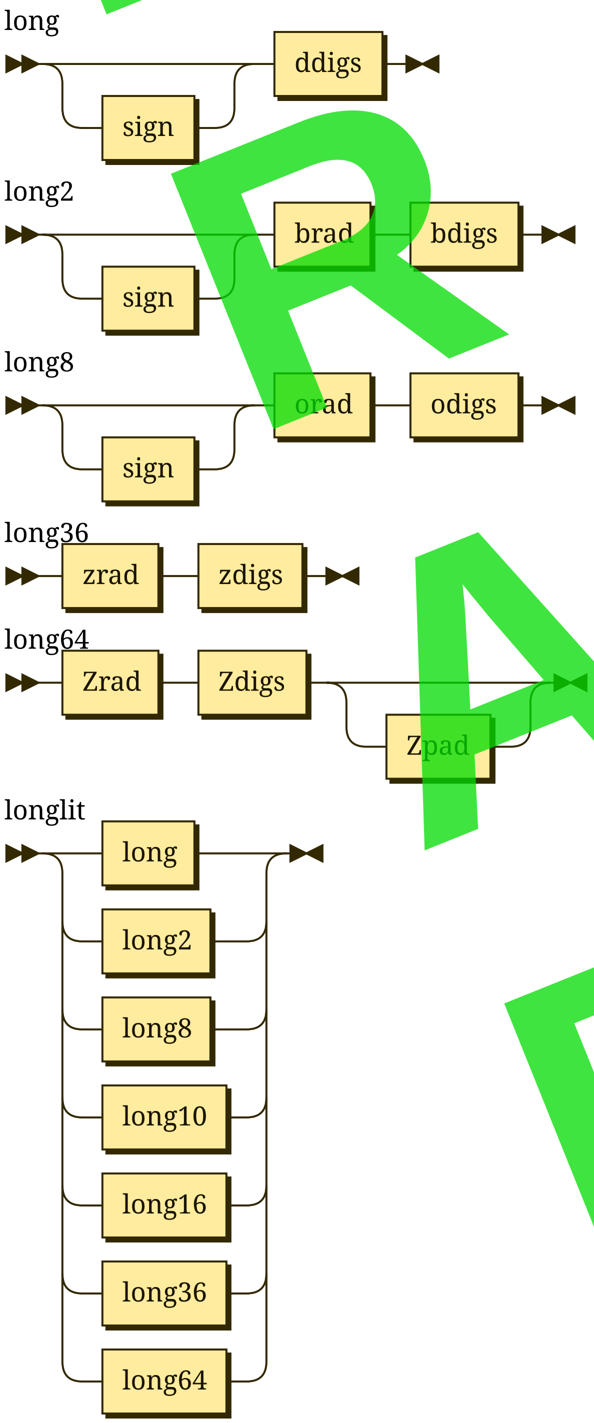


Figure 6.3: Long Literal railroad diagram (continued).



Long literals can be negative or positive. They can be binary, octal, decimal, hexadecimal, base36 and base64. Decimal is the default when no radix is specified. The default digit separator, a comma, can be used to break up large numbers for readability.

Figure 6.4: Real Literal railroad diagram.

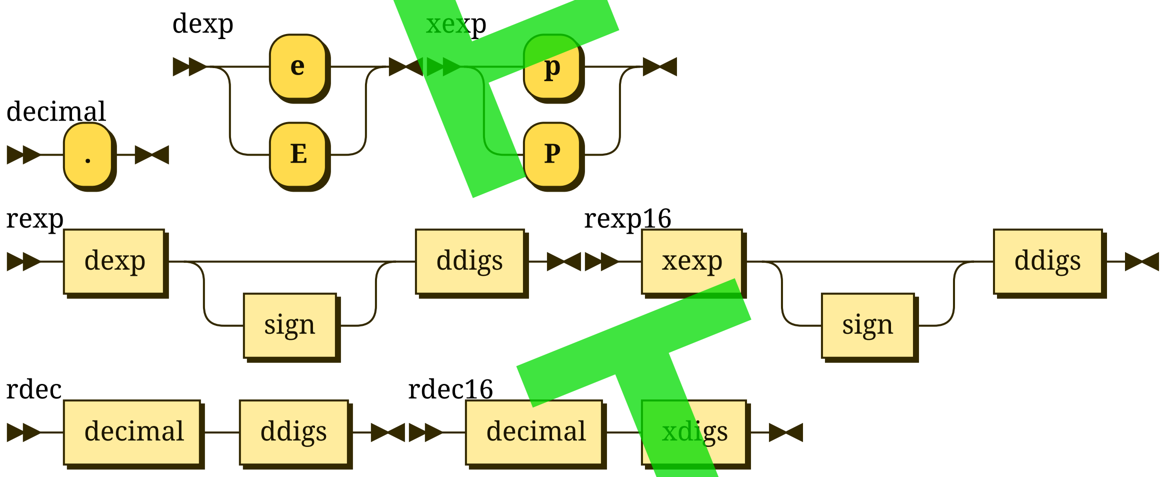
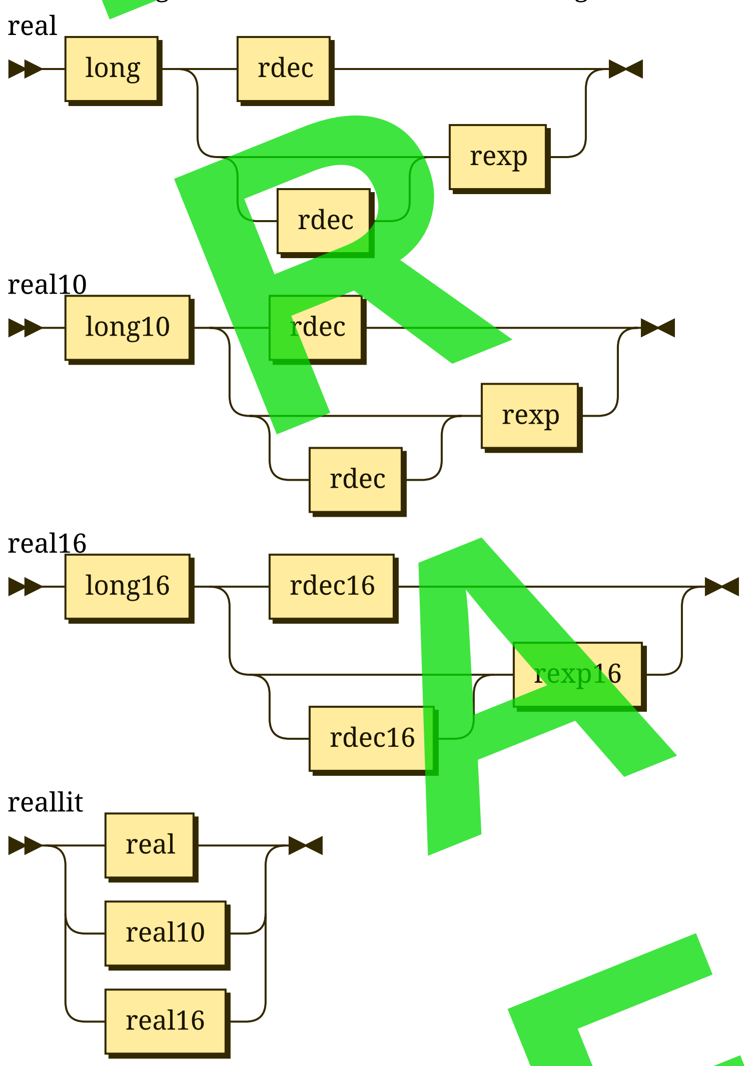
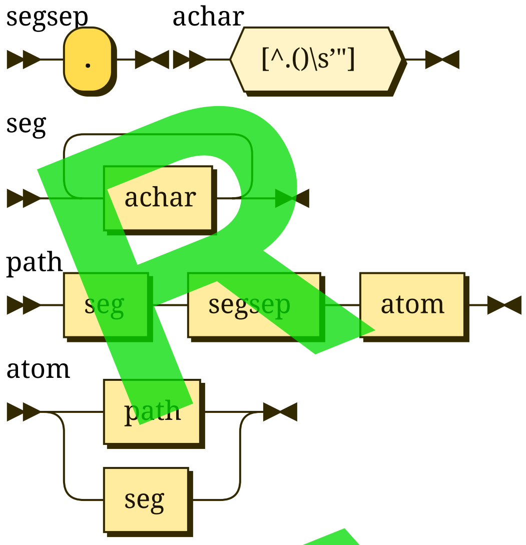


Figure 6.5: Real Literal railroad diagram (continued).



Real literals can be negative or positive. They can be decimal or hexadecimal. Decimal is the default when no radix is specified. The exponent is decimal, but in powers of 10 for decimal, and powers of 2 for hexadecimal.The default digit separator, a comma, can be used to break up large numbers for readability.

Figure 6.6: Atom Literal railroad diagram.



Atoms can contain any non-space, non-dot, non-parenthesis, non-apostrophe character. Atoms can form paths, with a single dot as a separator, in order to refer to objects within enclosures. Atoms with the @ character(s) are reserved by TONAL.

When discussing the structure of the Atom, we use the AnAtomy.

$$rooted\ branch = a.b.c.d.e \tag{6.1}$$

$$node = a,\ or\ b,\ or\ c,\ or\ d,\ or\ e \tag{6.2}$$

$$root = a \tag{6.3}$$

$$leaf = e \tag{6.4}$$

$$bud = .e \tag{6.5}$$

$$stem = a.b,\ or\ a.b.c,\ or\ a.b.c.d \tag{6.6}$$

$$limb = d,\ or\ c.d,\ or\ b.c.d \tag{6.7}$$

$$branch = d.e,\ or\ c.d.e,\ or\ b.c.d.e \tag{6.8}$$

By far the most natural way to structure information in computer science is the tree. The shape of botanical trees are a natural way to visualize hierarchical relationships between information. Cladistics and program structures naturally are organized in the same way.

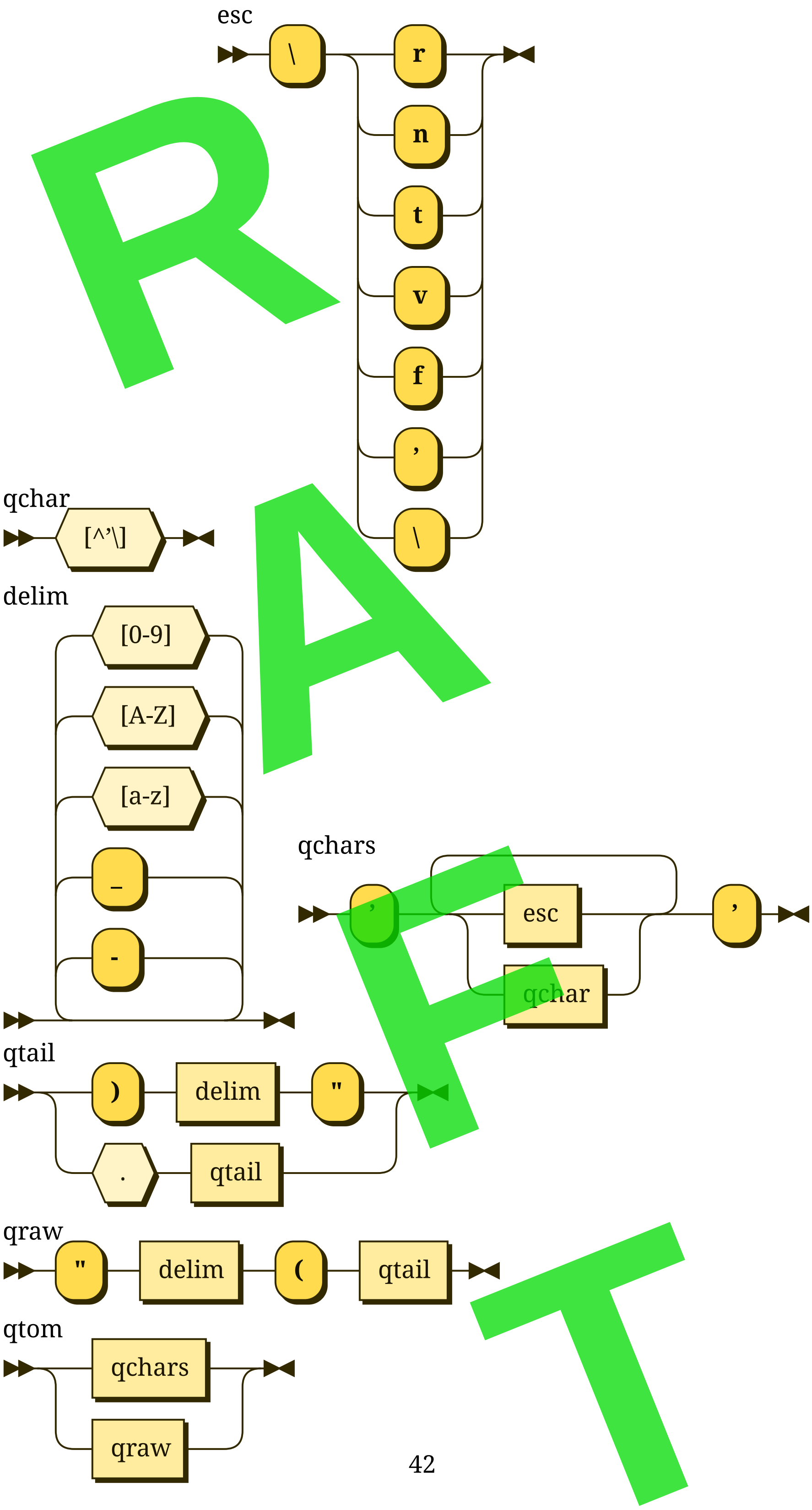
Computer science has mostly ditched the tree metaphor in favour of mathematical terms, because maths pays but botany doesn't. TONAL takes the tree metaphor and botanical terms to make the AnAtomy less abstract.

The nodes 6.2 of an atom 6.1 are like the nodes of a plant that are points along the plant where structures sprout from. The root 6.3 does not have nodes preceding it, while the leaf 6.4 is at the very end, just like they are in plants. A bud 6.5 grows out of a node.

A stem 6.6 is a root with nodes budding up any number of times before the leaf. A branch 6.8 is any node that is not a root, budding any number of nodes up to the leaf. A limb 6.7 is the common part of a stem and a branch; that is to say any non-root nodes budding up any number of times before the leaf.

Most atoms in code will technically be branches because the stem is contextually implied. The branch/limb/stem distinction allows us to talk about the parts of an atom without having to clarify whether there is a root/leaf node or not.

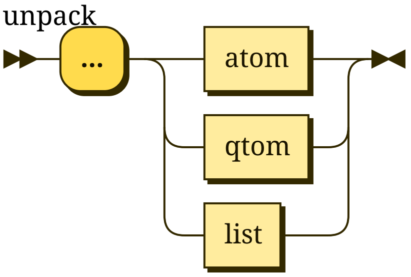
Figure 6.7: Qtom Literal railroad diagram.



Qtoms do not differentiate between characters or strings as in other languages. All unicode characters are valid in TONAL, so they can be used in Qtoms without hexadecimal codepoints. Qtoms are immutable.

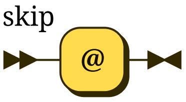
Qtoms can be raw, with a user-specified delimiter, so they can contain the Qtom quotes and the escape-character without escaping. Raw Qtoms that contain formatted/syntactic data, like regular expressions, can use the user-specified delimiter to specify the syntactic engine for 3rd party tools for validation, highlighting, autocomplete etc. The delimiter can be accessed programmatically.

Figure 6.8: Unpack Literal railroad diagram.



Unpack is the only literal allowed with multiple dots. They are used to specify pack-parameters, or to expand argument packs or lists in place.

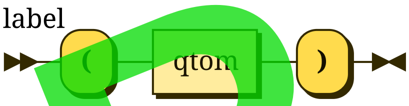
Figure 6.9: Skip Literal railroad diagram.



Skip is used to explicitly skip the rest of the current (possibly empty) pack-parameter, or to skip the current defaulted parameter; to the next parameter.

6.1 Labels and Comments

Figure 6.10: Label and Comments railroad diagram.



Labels serve as named locations for non-linear local control flow. Labels appearing before control flow can be used as early exits. Labels also should be descriptive, so to encourage that, labels are just Qtom literals. And because Labels are just Qtom literals, they are also just Comments. It comes full circle, Labels are just Comments, and so both should be descriptive.

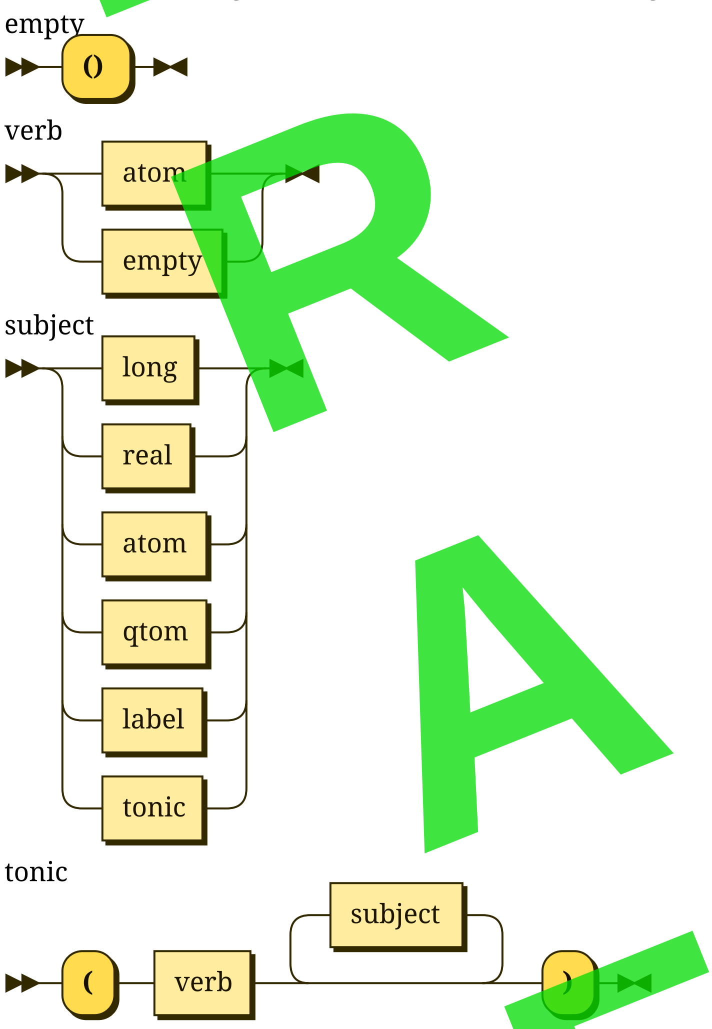
Having Labels as Qtom literals, as part of the regular syntax, also means that there is less friction between the language and special documentation syntax used for generating documentation. External tool support is not an afterthought.

Labels and Comments can be placed anywhere as they are ignored by the compiler for the most part. Labels that are jumped to are preserved, and so may only be placed in certain positions in Tonics that can actually be jumped to. For example, it makes no sense to jump to a parameter definition.

7 Tonics

Anything in TONAL that is not a literal is a command. All commands are girt by parentheses, so technically, a Label Literal is actually a command, but they have no other role than pure syntax. There are no other delimiters used for nesting, so all those characters are available to be used in Atoms.

Figure 7.1: Tonic Form railroad diagram.



The main form of a TONAL command is called a Tonic. All Tonics have two main positions - the Verb, and the Subjects. Literals in the Verb position are called Verbs. Literals in the Subjects position are called Subjects. They are usable interchangeably. Verbs are doing words, which tells a TONAL compiler what to do. The Subjects controls what the Verb does, beyond just what word is the verb.

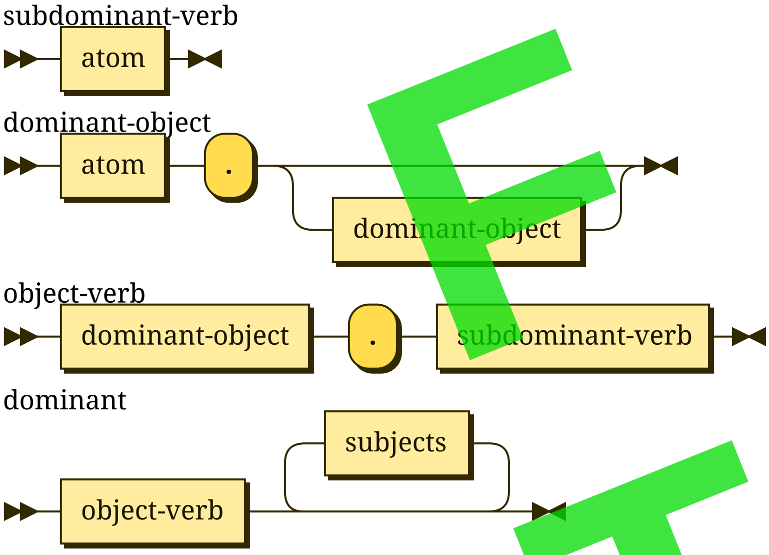
All commands follow the basic Tonic form, with special forms depending on the verb and subjects. There are no special syntaxes for things that are present in other languages, like compiler directives, macros, attributes, generics, annotations, formatted comments, manifests, service connector definitions, build files, etc. From here on, all syntax diagrams for any of the scale degrees will omit the opening and closing parentheses.

Table 2: *Syntax Scale Degrees*

Do	Tonic
Re	Supertonic
Mi	Mediant
Fa	Subdominant
So	Dominant
La	Submediant
Ti	Subtonic

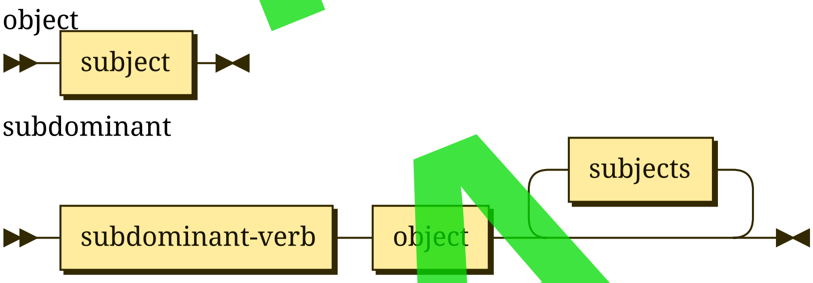
The Tonic form is so-called because it has an analogous function in TONAL in that it sets the tone for the rest of the language. Tonic is used within TONAL in a few senses. In a general sense, it refers to all forms of TONAL commands, being the Tonic scale degree; but in discussions with the other forms, it refers to commands that are not the other scale degrees. The other forms are either very apt, or very not, analogies of the seven scale degrees.

Figure 7.2: Dominant Form railroad diagram.



The most dominant form of command in TONAL is the Dominant form, which occurs when the Verb is an atom in the form of a path. The Object position is the atom path all the way up to the final dot, and the Verb position is final atom in the path. The dotted notation is familiar to all programmers by evoking the Object-Oriented paradigm, which is the most dominant paradigm in the software industry and feels completely natural with the Object position referencing some enclosing context for the Verb.

Figure 7.3: Subdominant Form railroad diagram.


object

subdominant

The Subdominant is not as dominant a form, but belongs to the same Object-Oriented paradigm as the Dominant form. The first Subject serves as the Object in which to try to find the suitable Verb, just like the Object part of the Dominant form's Object-Verb. If the Object doesn't enclose the Verb, or there is no Object (and therefore no Subjects), then this is just a Tonic and the Object is whatever encloses the Verb, when it is found, starting from where the Tonic is invoked.

The Tonic, Dominant, and Subdominant are essentially the same form with the same purpose - invoking a func. The other forms are special syntactical forms with meanings defined by the TONAL language and so will be discussed in their own sections.

The terms Dominant and Subdominant may also applies to Atoms with the same format as their Verbs, but not in the Verb position. In the context of such usage, use Dominant Tonic and Subdominant Tonic to refer to the Tonic instead of the Atom literal.

8 Scores - Keys, Bars, and Clef

A musical score, in overly general terms, is a composition that has a clef, a key signature, and bars. A clef delineates the beginning of the score. The key signature is considered in its entirety, and simultaneously across all bars. It imparts tonic structure to the notes. The music plays through the progression of the bars. There may be repeats. There may be jumps to labelled positions.

In languages like C++, compound statements are a list of statements of code. The body of a function or the block of a scoping keyword, like if, for, try, etc are compound statements. The program's execution progresses through these statements. The body of a class is compound statement and member declarations are processed before their definitions.

TONAL uses the musical metaphors of keys, bars, and clefs to avoid ambiguity and overloading of terms. The clef uses the empty list syntax to visually distinguish and demarcate the preamble from the sequence of tonics that make up the implementation.

Keys are the sequence of tonics following the clef that are the TONAL equivalent of the body of a class in languages like C++.

Bars are the sequence of tonics following the clef everywhere else that are the TONAL equivalent of the body of a function, or the compound statements of scopes. However, bar semantics are not only limited to bars. Even keys have bar-like semantics in some cases, just like how musical scores can have key-changes.

9 Major/Minor Distinction

Take the equation:

$$y = ax + b \tag{9.1}$$

In fact, you may have a few equations:

$$y = ax^2 + bx + c \tag{9.2}$$

$$\sin^2(\theta) + \cos^2(\theta) \tag{9.3}$$

Are they statements? Are they declarations? Are they expressions? Are they sub expressions? Are they definitions? Are they compound statements? There are many different terms in many programming languages, with subtle differences between them, often based on how their grammar is specified or what their compiler's frontend parser calls them. To people writing programs, it doesn't make a difference what they're called. They look like what they look like and things that look alike should behave alike.

In primary school, we learn about BODMAS/BIMDAS/PEMDAS. Programming languages extend the order-of-operations with language specific operators, but the principle is the same. Some calculations take precedence over others and their result is unofficially remembered while the rest of the calculations are performed.

In the equation 9.2 on the current page, the term ax^2 gets processed first. In that term, the x^2 term gets processed first. That produces a value - that doesn't have its own name - which gets used in a product with a to produce yet another anonymous value. The value of x^2 can now be discarded. Then the next term that gets processed is bx, producing yet another anonymous value. There are now no more exponents or multiplications to process. The next term to process is $ax^2 + bx$ using the anonymous values that were calculated, producing yet another anonymous value. The values of ax^2 and bx can now be discarded. The next term to process is $ax^2 + bx + c$, using the anonymous values that were calculated, producing yet another anonymous value. The value of $ax^2 + bx$

can now be discarded. Finally, $y = ax^2 + bx + c$ is processed, giving the anonymous calculated value the name of y.

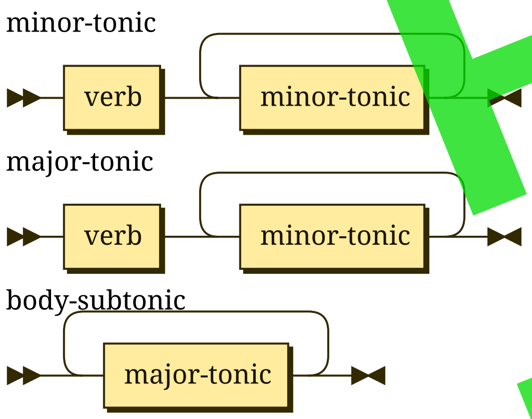
Then we move on to the next equation 9.3 on the preceding page, do the calculation, but we don't use the final anonymous value.

The value of y may be used again and again in further equations. Equations can just be added one after the other, as we do when calculating systems of equations.

Two patterns are established: one pattern is the calculation of intermediate terms, according to specific ordering rules, to be used for an overall equation; the other pattern is the calculation of full equations themselves, one after the other. In fact, in the example given, because equation 9.2 on the previous page and equation 9.3 on the preceding page do not share any references to named variables, they can be calculated out of order, which is a fact that compilers and processors can take advantage of to speed up calculations.

TONAL gives names to only these two patterns of code, which reduces confusion due to many terms that mean similar things that have no bearing on what programmers actually think about.

Figure 9.1: Major/Minor railroad diagram.



[a]

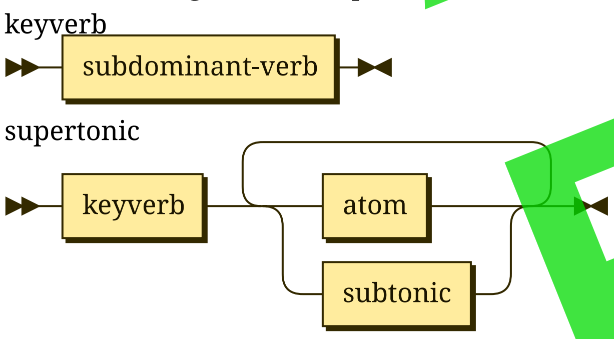
[a] Other components of the Tonics are omitted for clarity - to illustrate the relationship between major and minor only.

The order of operations for languages like TONAL is specified by this acronym - P. P stands for Parenthesis. Parenthesis (ie, Tonics) are always processed first, and in a left-to-right order. Tonic subjects that aren't Tonics are either names or literals, so they are already processed. Every Tonic subject that is a Tonic is a Minor Tonic. Major Tonic refers to an unlimited sequence of Tonics that either do some processing and the final value is ignored, or are Supertonics, including but not limited to those that bind a name to a value. They are found only in a Subtonic called a body in certain declarative or control flow Supertonics.

The Major and Minor nomenclature recalls the music theory theme, but has a hierarchical relationship compared to the musical function of major and minor keys.

10 Supertonics and Subtonics

Figure 10.1: Supertonic Form railroad diagram.



Supertonics are Tonics with special powers through the magic of Keyverbs. Not keywords, but Keyverbs. Keyverbs are Verbs with predefined spellings. TONAL does not have the concept of reserved keywords that other languages have, in order to give programs more freedom in naming things. Keyverbs are Verbs, so they are only reserved for the Verb position, unlike keywords, which are reserved regardless of where they are used.

Some Subjects of Supertonics are words in specific positions in the enclosing Supertonic. They could have been made to be Keyverbs but they don't have use outside of their respective Supertonics so it's a waste of a Keyverb. They don't need to be reserved keywords because they have extremely limited uses. Subjects that are Atoms in specific position are reserved just for that position, because there could be no disambiguity with a let-name. In that, TONAL takes after C++ (as discussed in Influence on page 15) where, eg, the location specific keywords 'override' and 'final' get around the potential explosion of reserved words that are commonly used names in programs.

All Keyverbs live within the type 'gut' that serves as a namespace. The 'gut' is preferably omitted in code, but for special circumstances. In the event a new language feature is introduced with a new Keyverb, that Keyverb may conflict with existing code that may have used the atom to name one of their let-names.[5] TONAL allows programmers to temporarily reassign a local replacement Keyverb so that it is painless to upgrade to a new version of the language without unexpectedly changing semantics. The Keyverb can then used with the local replacement, or be used with the 'gut.' prefix.

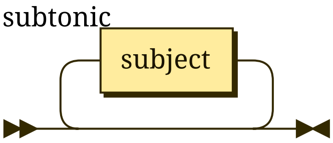
The 'gut' type is special - any Atom path (not necessarily in the Object-Verb Dominant position) that starts with 'gut' unambiguously refers to anything that is contained within the language defined 'gut' type. Other special types are 'hype' and 'std' and have the same semantics, but they do not define any Keyverbs. In the event a new language feature is introduced with a new special type, that type may conflict with existing code that may have used to atom to name one of their let-names. TONAL allows programmers to temporarily reassign a local replacement atom to use as that special type.

[5] This only affects atoms in the Verb position of a Tonic, since this is what the concept of Keyverbs is designed for. If a program has existing things with the same name, but are used in the Subjects position, then TONAL will never mistake them to be Keyverbs. But still, be sensible about it, and don't name things that reuse Keyverb names just for the hell of it.

The temporary local reassignment does not transfer over to included source files when such a reassignment is in effect. Programmers often want to view a source file separately, out of context, and it would be confusing to not see the regular Keyverbs.

Even the command for the temporary local reassignment of Keyverbs and the special types are themselves Supertonics, so it would be easy for an ad-hoc parser to do the same processing that TONAL does to handle these reassignments.

Figure 10.2: Supertonic Form railroad diagram.



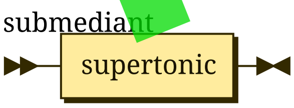
Subtonics are Tonics with bits missing, like having a Verb position, that are Subjects of Supertonics. They can miss out on having a Verb because their position within the Supertonic tells it what it is. Subtonics only has Subjects and do not have a Verb, but they still must not have a Keyverb as the first Subject even if it views a legitimate let-name that is not a Keyverb. This is not because TONAL cannot disambiguate it - it definitely can.

The issue is due to ad-hoc tools. One of TONAL's design points that emerged is that Keyverbs provide a significant opportunity to design for allowing programmers to write ad-hoc tools. Keyverbs are relatively few in number and have only one position - ie, after a parenthesis - so it would be easy to write a small tool from scratch that only looks for Keyverbs. Or the tool can look for things that are not Keyverbs in the Verb position. Authors of such programs would not have to deal with ambiguities just to get an ad-hoc tool working.

Supertonics have very specific forms, but they all follow the Tonic with a Keyverb pattern, so an ad-hoc tool can simply skip over the things they're not interested in. They can safely ignore nesting in most cases, because Supertonics largely mean the same thing. In cases they don't, their pattern of Subjects and Subtonics are different from Supertonics; or if necessary, it's easy to find the enclosing Tonic to get the context.

Some Supertonics have alternate forms. The Mediant forms are related to pattern matching, and they are limited to only a few unrelated Keyverbs without much commonality between them. In short, it is easy to remember M for Mediant, M for matching.

Figure 10.3: Submediant Form railroad diagram.



Supertonics that are scoped control flow commands have a special form, unchanged, that are the Verb of a Tonic with no Subjects. And apart from lists, Submediants are the only form that allows a Tonic as its Verb.

The purpose is straightforward. The Supertonic is a Verb, so it is invoked. TONAL does not follow other modern languages in making everything an expression, so flow control constructs don't evaluate to a value. There are uses for this, hence its popularity in other languages[6], and C++ has a way to achieve this called "Immediately Invoked Lambda". The Submediant syntax was chosen for this reason, because having a Supertonic as the Verb is like immediately invoking a lambda. The difference is that there isn't a local func that's constructed that would require grabbing the local environment, and it is not storable at any memory location.

10.1 Keyverbs

TONAL tries the reserve as few Keyverbs as possible. Even though the Keyverb design means there are no reserved keywords, it's still better not to have too many different primitives for a language. With a mix of

[6] The tertiary operator in C and C++ are similar concepts, but the only example in those languages.

Keyverbs and placeholder atoms, we can increase the variations without increasing mental load.

On the flip side, we don't want to get into the same situation as C++ where the keyword “static” has taken on so many different contextual meanings.

All in all, TONAL will try to reuse Keyverbs provided that there's a clean syntactic break from existing Supertonic forms and that the meaning of the Keyverb strongly relates to what the other forms, and the word itself, means.

Keyverbs fall in five categories.

Declaration keyverbs are those that introduces a distinct thing.

Control Flow keyverbs are those that does linear jumps; related to branching.

Control Jump keyverbs are those that does surprising jumps; they are either easy to abuse and widely considered harmful, or they are expensive to use, or they are hard to follow and mistakes create surprising side-effects.

Lifecycle keyverbs are those that control the cleanup of things that were constructed.

Access control keyverbs are those that alter the default settings of who can touch what and do what.

Table 3: *Keyverbs with major supertonic form*

Declaration	Control Flow	Control Jump	Lifecycle	Access Control
let	if/iff	wait	push-in	readers
func	loop	give	pull-in	writers
type	stop	goto	push-out	virtual
	next	trip	pull-out	include
	eval	trap		grab

Every Keyverb except the list specifier can be used in major form. Not all of them can be used in a declarative body subtonic, and not all of them can be used in an imperative body subtonic.

Table 4: *Keyverbs with minor supertonic form*

Declaration	Control Flow	Control Jump	Lifecycle	Access Control
func			push-in	readers
type			pull-out	writers
()				virtual
				include

List specifiers can only be used in a minor form, because the list must be either assigned to a let-name, or passed to a Tensah.

Table 5: *Keyverbs with mediant form*

Declaration	Control Flow	Control Jump	Lifecycle	Access Control
let			push-in	
func				
type				

Table 6: *Keyverbs with submediant form*

Declaration	Control Flow	Control Jump	Lifecycle	Access Control
let	if/iff	trap		
func	loop			
type				

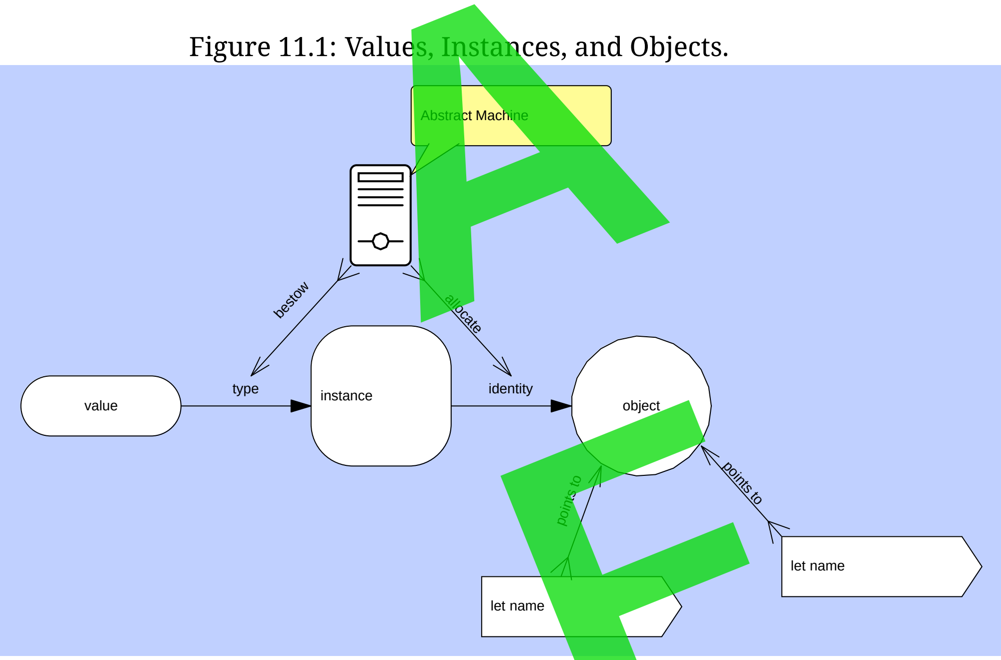
Part III

Objects

11 Taxonomies

TONAL is defined in terms of an abstract machine (AM, herein). This AM is assumed to be running on top of the C abstract machine, which in turn generalizes a basic hardware model; however, this is not a prerequisite or part of the definition. The 'bit' (not assumed to be a binary digit) is assumed to be the lowest level of information of the AM.

Figure 11.1: Values, Instances, and Objects.



A value in the AM is a finite sequence of bits. By themselves, they have no intrinsic meaning. Nothing can be done unless the AM bestows a type upon the value.

An instance is a specific value with a type; a member of the set of all valid potential values of a type. The AM allows us to discern all that can be done with (or involving) an instance, but it doesn't tell us where, when, and what order those things can be done.

An object is an instance with an identity allocated by the AM. Whereas values may have a matching sequence of bits but their equivalence is undefined; and matching instances may be equivalent but their identicality is undefined; an object is only identical to itself, but may be equivalent with other, non-identical, objects. The AM enforces the constraint of identity, whether or not that object's identity is accessible.

The term 'variable' is often used to talk about an object or its value interchangeably, so for the purposes of this document, the term 'let-name' will be used instead, because variables are often defined with a 'let' command. A let-name is a proxy to the identity of an object in the AM. A let-name allows us to discern where, when, and what order things can be done with (or involving) an object.

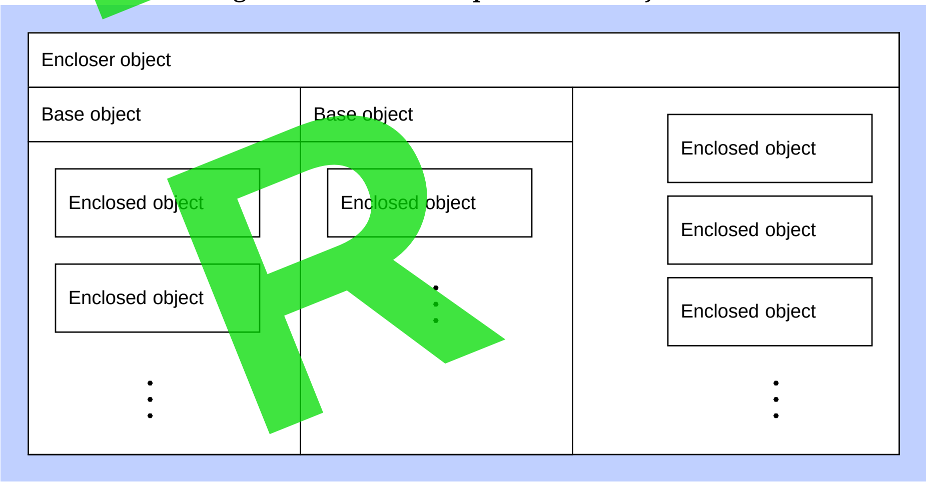
Table 7: *Abstract Machine tenses*

Present-tense	roughly corresponds to	Compile-time
Future-tense		Run-time
Past-tense		Moved or destroyed

A let-name is in either 'future', 'present', or 'past' tense. A let-name in future-tense means its object's value is only known when the compiled program is executing. A let-name in present-tense means its object's value is known when the AM is executing TONAL commands. A let-name in past-tense means that the AM has disassociated the let-name from its object's identity. Using a past-tense let-name is an error. The dissociated object may have a new let-name bestowed up on it by the AM.

A present-tense let-name might move to future-tense if it interacts with an object's value not known to the AM, or another future-tense let-name. A future-tense let-name might move to present-tense if it is wholly replaced by an object with a value known to the AM.

Figure 11.2: The composition of objects.



Objects may be made up of other objects, as defined by the object's type. Those objects are called 'enclosed', and the object that encloses those objects is the 'encloser'. The identity of the encloser is dominant over the enclosed; the identities of the enclosed are subdominant to the encloser. The identity of enclosed objects are reachable only through the identity of the encloser.

In the real world, complex objects are composed from smaller objects – like Generic-Brick-Construction-Toy. TONAL's primary design tool is composition of smaller objects. To make cost-efficient complex objects, designs have to be built around well-defined, standardized interfaces – like Generic-Brick-Construction-Toy. Adherence to interfaces and their requirements do not dictate how an object meets them, only that they declare that they meet them.

Objects can implicitly adhere to an interface as long as the generic Tensah that is called upon it accepts it in the present-tense. Objects can explicitly adhere to an interface by its type being derived from the interface. Such an interface is called the object's base type; objects can have more than one base type. A base type can be a partial or complete implementation of an interface. The base type object is enclosed by the deriving object, but allows the deriving object to be used as though it is that enclosed base type object.

Real world objects interact in limited ways, even when they're part of the same object. Many objects have some surface boundary that other objects interact with and prevent them from touching the insides. But even for objects inside that surface, not all of them should be able to interact with each other. By way of a car analogy: a car should not touch the insides of another car; a driver needs to be granted access to some parts inside of a car; a passenger could be granted access to a much smaller set of parts; a car's exhaust and its air conditioning system shouldn't exchange gases.

Objects have control, in the present-tense, over which of its enclosed objects – including base type object – that other objects can have access to. They can even control access between enclosed objects.

Read access allows an object's value to be used. Write access allows an object's value to be modified. Virtual access allows an object to customize how its base type object performs some function. Grab access allows an enclosed Tensah, one which doesn't implicitly grab its encloser, to access its co-enclosed objects. Include access allows Tensahs and data defined in external files to treated as being defined in-place.

TONAL access categories are independent of each other. eg, we can read the level of a fuel gauge, but we use a special process to add more fuel, not by manually writing the level of the fuel gauge itself. eg, you can customize your order in a restaurant, but you don't actually direct how the chef makes it.

12 Conditioning

Error handling in most languages is added as an afterthought and not treated at language level. For languages with error handling support, like the common try-catch-exception scheme, they only cover runtime errors. In languages like C++, it is even impossible to enforce at compile-time whether, and which, exceptions need to be handled. On the flip-side, languages like Java require declaring exceptions as part of the function signature, and/or for exceptions to be handled, which can be quite annoying.

The exception concept can also be confusing because it's supposed to be used only for exceptional circumstances. If naming is the most difficult task in programming (and science in general), then categorizing things is the second. "Exceptional" means something different to everyone in different environments and time-of-day. There is no universally agreed-upon understanding of what constitutes an exceptional circumstance.

Many APIs have error codes or status code that is either returned from function calls, or has some global state that must be documented and checked after every API call. Return codes may be annotated to force the compiler to issue some warning or error if it hasn't been checked, but global state errors have no language support. Status code reports on a status which may or may not be considered exceptional, but all else being equal, the designers of such an API would prefer programmers check the status before continuing.

Assertions, pre-conditions, and post-conditions enforce contracts that cannot otherwise be enforced through the use of types. For example, a ranged-integer type is used to enforce contracts about accepted integer ranges in APIs that use it, but the ranged-integer type itself can only use raw integers. It has to use contracts, because there aren't restricted types all the way down.

Implementations of contracts in most languages tend to be enforced at runtime, and they also tend to evaluate to immediate failures. External tools may be needed to handle these failures in order to do useful things like automated testing. In C++, static_asserts cannot be handled at all, as they cause compile-errors on failure, so they are impossible to test in an automated way without some macro trickery to replace static_asserts with normal asserts or exceptions in tests.

TONAL conditions unify all out-of-band status reporting use-cases - non-return code schemes, in other words. They syntax is broadly the same as exception handling in other languages. Conditions are not thrown or raised, but tripped, and trapped (or trapped not, there is no try). Programmers get TONAL present-tense condition trapping for free, and therefore also get contracts and unit-testing support when and where it matters most: before the program even runs, and library/application specific guarantees beyond just the language-level checks.

Conditions that are not trapped by a func are recorded in a set of conditions so that they are tracked at present-tense. It is not necessary for programmers to maintain lists of possible exceptions to silence the compiler at every function, like it is in Java. Nor is it necessary to check for a condition every time, as it is for return codes or global error numbers. It is even possible for the AM to track which conditions will not be trapped and deduce an exit fast-path for a task or the entire program.

API designers set up trip-wires all around the code and programmers trap conditions that they know how to handle.

12.1 Atonal states

The counterpart of tonal music is atonal music. Atonal conditions arise when code is incompatible with the TONAL language. All atonal conditions arise in the present-tense, generated either by the AM, or tripped by code.

Atonal conditions that can be trapped are atonable. Atonable conditions must be trapped, and possibly explicitly re-tripped in present-tense, or it is an immediate AM-error. Atonal conditions can be used by library designers to do things like allowing programmers to choose a different path, such as selecting different overloads, or build configurations.

Atonal conditions that cannot be trapped are unatonable. Tripping an unatonable atonal condition is an immediate AM-error. An unatonable condition can be intercepted in order to be inspected, but will escape any trap. There are certain rules of the language, such as access rules, must not be recovered from, because it increases the chances of errors when fundamental properties of the language are broken if trapping them were allowed.

The capability for regular code to trip atonal states in present-tense to drive the AM is another way to eliminate differences between regular code and the AM. Code can be treated as though it is just a generic extension of the AM's operation.

13 @jectives

Atoms beginning with the single character @ have values that are injected by the AM. For reading and parsing ease, such as for third party tools, the @ character is forbidden from any user-defined let-name in any position of the atom.

These values are provided to make present-tense reflection easy to use. All values are one of the archetypes.

Table 8: *@jectives for Source*

@line
@file
@date
@time
@epoc
@tick
@nest
@path
@libs
@root
@prod

These values are defined anywhere in a source file. They are noncontextual in that there are no enclosing objects that they can be used as a Dominant let-name. These values expose information about the compile process.

The value @line gives, as a Long, the number of newlines passed by the AM before this point; @file gives, as a Qtom, the name of the current file being processed by the AM relative to the initial working directory that the AM started in; @date gives, as a Qtom, the UTC day that the AM started on; @time gives, as a Qtom, the UTC time at microsecond resolution that the AM started on; @epoc gives, as a Long, the number of microseconds since the Unix Epoch; @tick gives, as a Long, an always increasing number every time the AM injects this value; @nest gives, as a Long, the Tonic nesting level at the time the AM injects this value; @path gives, as a Qtom, the AM's installation filesystem canonical URI; @libs gives, as a List, the Qtom filesystem canonical URIs for where, and the order, in which to search for libraries; @root gives, as a List, the Qtom filesystem canonical URI of the directory of the root source file; @prod gives, as a Qtom, the filesystem canonical URI of the directory of the output binary.

Table 9: *Tensah @jectives*

@type
@name
@object
@derived
@params
@grabs
@size

These values are defined relative to the immediately enclosing Tensah. They can be accessed by Subdominant from within a Tensah, or by Dominant with an object.

The value @type gives, as a Type, the type of the value that is the Dominant. When used from within a type, it is that type. When used within a func enclosed by a type, it is that enclosing type. If a func-enclosed func grabs @object, it is the type of @object. The value @name gives, as a Qtom, the Subdominant name of the Tensah (maybe even from the @type object). The value @object gives, as a value, the enclosing object only when there is a Dominant object. If a Tensah grabs @object, then the object is the Dominant object. If the Tensah is accessed via a type as the Dominant, then there is no @object. The value @params gives, as two Lists in a List, of Qtoms for the parameter names and their types. The value @grabs gives, as a List, the types of the objects that may have been grabbed by a Tensah. The value @size gives, as a Long, the number of bytes that an object takes up in memory, including padding.

The value of @object and @param depends on how a Tensah is accessed: for @object, via a Dominant object vs via a type; for @param generic parameter types are unknown until invoked. Therefore those @jectives are not accessible outside of the Tensah being accessed. Nevertheless, the Tensah could choose to save the information when accessed and expose the information manually.

The value of equating @type to a Type Tensah when used allows anonymous types to be referred to.

The values of the elements of @grabs can't be given, as their TOWEL may have been transferred to the Tensah. It would expose too much internal information. If the Tensah's enclosing @object is grabbed, then it is always the first element of the list, and the Tensah is also in the @object's @enclosed objects.

Memory only exists in future-tense, so the @size of all archetypes is 0. Types that derive from the Scalah archetypes do have a @size greater than 0, even though they may have a value in present-tense. Accessing the @size of a type may forbid some possible optimizations; such as eliding v-table pointers, or eliding enclosed objects, or hoisting present-tense enclosed objects outside of a type; since the AM must give a consistent answer.

Table 10: *Func enclosed @jectives*

@func
@early-destruction
@current-condition
@conds
@traps

These values are defined for a func in addition to the objects defined in *Tensah @jectives* on page 64. The value @func gives, as a Func, the enclosing func. This allows anonymous funcs to be called recursively without having to design the language to get around to problem of how to access something while it's being defined. It also avoids the problem of accidentally calling an overload in the enclosing type if it exists. The value @early-destruction gives, as Long, 1, if an object is being destroyed and a push-out has triggered an early destruction; 0, otherwise. The value @current-condition gives the object that was tripped as a condition, as though it were pulled-out, such that it is primed for TOWEL roundtrip.

The value @conds gives, as a List, the conditions that can escape the func. This value can not be accessed inside the func itself, as the the list cannot be finalized until the end of the func.

Funcs do not define an @encloser value because either a func is enclosed by a type, in which case it already implicitly grabs the enclosing @object or @type; or if it is a func-enclosed func, it should explicitly grab what it needs (including the enclosing func's @object, if it has grabbed it) instead of blanket grabbing everything via the @encloser.

Just like @object and @params, @func can only be accessed from within the func it refers to, for the same reason that there might be some generic parameters that are only on invocation, or some grabbed values.

Table 11: *Type @jectives*

@bases
@encloser
@enclosed
@~type

These values are defined for a type in addition to the objects defined in *Tensah @jectives* on page 64. The value @bases gives, as a List, the types and/or the present-tense value that a type derives from. The value @encloser gives, as a Type, the type inside which the type is defined inside. The value @~type gives, as a func, the destructor of the type, which is useful for anonymous types. The value @enclosed gives, as a List, the names and the types of the Type's enclosed objects.

As with @object, @params, and @func, these @jectives are only accessible from inside the type, for the same reasons, but they could be exposed as enclosed let-names or through accessor funcs.

14 TONAL hooks

A musical hook is a piece of music that is recalled often. The AM has hooks for common customization requirements. They are not explicitly invoked by the programmer, but are instead implicitly invoked by the AM at specific syntactic junctions. The hooks allow the AM to guarantee safety and performance by reliably doing all the things that programmers eventually forget to do at some point; or the program logic is too complex and would get missed by the programmer. Where the AM can determine safety, the AM can elide some invocations, bring forward invocations, or delay invocation until necessary.

The following subsections explain the purpose and action of these hooks. The syntax will be explained in Supertonic Func on page 129.

14.1 Conversion

Creating an object of one type from an object of another type is called conversion. In some languages, it is called coercion. Relaying is sort of like conversion, except it's the same underlying object. Some languages implicitly convert an object when constructing a variable, or assigning to an existing variable, or when passed to a function that takes a different, but convertible type.

Implicit conversion is the cause of a lot of errors, especially in regards to readability - code meaning what it looks like it means, and between different number formats. Finding the correct function that is overloaded can result in surprises, and therefore bugs, if implicit conversion is allowed.

In TONAL, all but one case of conversions are just a construction of the type that is desired - hence there is no implicit conversion. When pruning an overload-set of Tensahs, exact types must match, so the programmer must explicitly use the construction of the desired type for that Tensah parameter. Even if the parameter is a base type of the object being passed in, conversion must be explicit.

The only case in TONAL where conversions are implicit, including upcasting to the base type, is when there is no overload pruning. If a Tensah doesn't have overloads - just the one definition - then implicit conversion can take place. There is no risk of calling the wrong Tensah. If the programmer then adds an overload in the future, the AM will then complain about requiring the exact type to be passed.

Conversion may not necessarily result in the creation of a new object. In the above example of a base type, no object of the base type needs to be created. It is just has the effect of selecting the overload candidate, and the actual object will be passed to the overload if it is the selected one. It is as though the object were relayed as the base type, but without actually relaying the object being refered to.

A similar case is conversions of a machine number type to a wider type, such as from char to int, or float to double. No underflow or overflow errors are possible with widening conversions, so no copy is required when converting for the purposes of selecting an overload from the set.

TONAL's present-tense programming is enabled by the use of the archetypes.

An object, maybe of a user-defined type, is considered present-tense if it is convertable to one of the archetypes. The unpack literal can be used on a user-defined type if the type can be converted to an archetype list. The list itself is present-tense, but the elements do not need to be.

14.2 Redirection

Generic programs might require the programmer to make an object look like a different type by way of wrappers/adapters/bridges/decorators. In some extreme cases, for example, the wrapper type may map close to the destination type, but due to some design constraints, cannot merely inherit from the destination type, so the programmer would have to manually connect the wrapper type's interface to the destination type's interface.

For a concrete example, consider higher-order types, such as an optional type. If an optional object contains the object of the destination type, then we would want to allow generic programs to access that optional object as the destination object. However, the optional object has its own interface - eg a func named valid - that might clash with the destination type.

Redirection would allow programmers to alter how TONAL understands dominant atoms. If a redirection exists, TONAL will always defer to it, regardless of whether other enclosed objects and overloads exist. The redirection is then responsible for interpreting the atom - as a qtom - and forwarding to the desired destination, or to an enclosed object of the wrapper type according to some naming scheme defined by the programmer.

Another use for redirection is to make an type behave like a func when used in the verb position of a command.

Redirection goes to the heart of how TONAL handles dominant letnames, so there are some restrictions that prevent redirections from behaving like regular funcs.

Recursion is disabled for redirections, but more accurately, redirections and their overloads are invisible from within a type, which means any enclosed object fallback value command or enclosed func does not see any redirections that are defined. Respectively, accesses from outside the type only sees the redirections. This simplifies how programmers can think about redirections, as they don't have to worry about infinite recursion from within a type in present-tense. They also don't have to worry about what actually gets refered to.

15 TOWEL

TONAL extends the Resource Acquisition Is Initialization paradigm that C++, D, Ada and Rust are built on with Total Ownership With Elastic Lifecycle.

An object, under TOWEL and RAII, is automatically destroyed when it goes out of lexical scope. In both TOWEL and RAII, the lifetime of an object begins when the let-name/variable is first introduced. In both TOWEL and RAII, the lexical scope is bounded by imperative bodies. In C++, that means the braces. In TONAL, that means func bodies and control-flow/control-jump bodies. In both TOWEL and RAII, moves and copies are elided if an object can be constructed in its final destination.

However, in TOWEL, moves are destructive, meaning that the let-name can no longer be refered to after a move. It doesn't mean that destruction isn't performed. On the contrary, the responsibility for destruction is merely moved to the final lexical scope that the object resides. This eliminates the need for a C++-style move construction and a direct memory copy suffices (in cases where moving is necessary).

In languages like C++, RAII intersects with its type-system through type-qualifiers - qualified types are distinct from the unqualified type in many contexts. Constant references can extend the life of a temporary object.

r-value references for temporary objects support move-semantics, which alters what can be done with object lifetimes. Because qualified types are distinct from each other, they can be used for function overloading in those languages.

TONAL breaks away from the tradition of convolving lifetime management with the type-system, due to the discovery that TOWEL makes type qualifiers, not only unnecessary for lifetime management, but also not necessary to be part of the type-system. In fact, TOWEL supertonics lets the programmer capture the intent to read/write/discard, and that tells the AM whether to use views, allow mutability, or no longer used in a certain lexical scope.

Programmers can also see the intent of other programmers. Programmers can tell others when they no longer need to write to an object. Programmers can tell others when a let-name is only temporarily required before being shunted somewhere else. Most importantly, programmers can tell others who should own the object and when it can be destroyed.

One side effect of not requiring reference or const/mut qualifiers means that funcs don't/can't have different overloads that takes different qualifiers of the same type. This simplifies API design when funcs don't need to explicitly handle each case. The API is not polluted with information about type qualifiers. One common annoyance in generic C++ is having to account for all qualifiers, necessitating the notion of forwarding references with a syntax that is often confused for r-value references. Programmers just tell the AM and other programmers how they intend to use an object, and the AM will take care of the correctness and performance.

Another side effect is that there is no need to talk about value categories. In C++, there is talk of l-value, r-value, x-value, gr-value, pr-value. The differences between them can be subtle, while also not telling you what is going on. One common confusion in C++ is the difference between the type of a variable vs the type of the object, eg function arguments. A parameter's type might be an r-value, but the type of the variable is an l-value. The object that the variable name points to might be temporary. With TOWEL, there is a direct physical metaphor that any person living in a physical world can understand.

In TONAL, you'll always know where your TOWEL is.

15.1 Func local

Programming languages and programmers can convey information by doing nothing. The default option itself has intent. In TONAL, all objects begin life inside a func. Whether a TONAL program is executed, or a TONAL shared library is loaded, a func is invoked that creates objects and/or has objects created in it, that invokes other funcs which creates more objects.

Objects are created to be modified. More precisely, objects are created to be modified in the lexical scope they were created. Objects are commonly created and then tweaked a bit before being passed off to other funcs to do work with. Programs do work, and to do work, programs have state: objects with changing values or enclosed object values. So when objects are first created, they are non-const values: they may be written-to and read-from. This is the same as most languages where non-const is default, like C++, and the opposite of newer languages, like Rust, where things are non-mut by default.

After a func local object is created and tweaked, several things can be done with it:

1. Go out of lexical scope, to allow the destruction of the object.
2. Viewed as another let-name.
3. Be passed into a func that does work using the object.
4. Passed out of the local lexical scope as the evaluation of the func.

5. Passed out of the local lexical scope as an output parameter.

6. Replaced with a new value.

7. Replace an enclosed object of the func's enclosing object.

Points 5, 6, and 7 require explicit TOWEL transfers. Point 2, 3, and 4 may have explicit TOWEL transfers. The other points do not transfer TOWEL.

With respect to point 1, func local objects remain writable until the end of their lexical scope. At the end of lexical scope, they become past-tense and are destroyed. No TOWEL transfer occurs - it is the default thing that happens.

With respect to point 2, a let-name that refers to another let-name without explicit TOWEL transfer - eg, it is not the construction of an object - is a read-only view. No copy is made, in contrast to C++, which requires a reference-qualified variable in order to prevent a copy. While the secondary let-name is in lexical scope, the primary let-name is also read-only. The primary let-name still has the TOWEL of the object.

With respect to point 3, this is the same as point 2. TOWEL remains with the primary let-name when there is no explicit TOWEL transfer, so the func only has a read-only view of the let-name. In contrast to func-local objects, which are expected to be modifiable in the general case; objects passed to a func are expected, in the general case, to merely inform the func as to how it should operate. In this case, TONAL takes after const-by-default languages like Rust. This strikes a balance between C++ and Rust, between usability and safety. Func-local objects knows where their TOWEL is, whereas invoked funcs do not, so it is usable and safe to make func-local objects modifiable by default, while being safe for objects passed to funcs read-only by default. In some ways, it is also very usable to prevent unnecessary modification by invoked funcs, as it can make following the logic of a program easier.

With respect to point 4, TOWEL is implicitly transferred. The func finishes and no longer needs the func-local object. Due to the object beginning its lifecycle locally, the AM has complete freedom to elide any copying or moving of the object to its evaluation destination. In actual fact, when evaluating a func-local object, there is no TOWEL transfer, because the AM knows to construct the object at its evaluation destination from the beginning.

With regards to enclosed objects of func-local objects, points 1 to 3 are the same. With point 4, enclosed objects of a func-local object cannot be constructed at the evaluation destination.

With regards to the intersection of point 2 and 4 - evaluating a func-local let-name view to another let-name - this requires a TOWEL transfer, because it is not correct to evaluate to a view to a local let-name in any language. Once the func-local let-name is destroyed when the func finishes, the view would otherwise be dangling, so a TOWEL transfer must happen to prevent that.

Func-local objects with enclosed objects that are views to other objects have the same problem on evaluation, but such objects cannot be constructed at the return destination at all and therefore a AM-error.

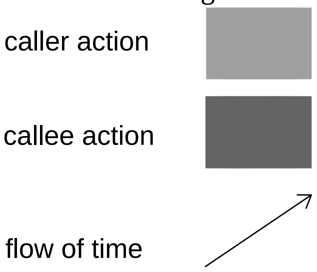
15.2 Transfer

One way to think of funcs is having an inside and outside. In most languages, there is talk of a call-stack. Depending on architecture, compiler, and operating system; call-stacks may grow up, they may grow down, they may be discontiguous in the face of concurrency, like threads, fibres, signals and resumable functions. This variability could mean confusion, since stacks, in computer-science, generally have a top; but choosing this terminology for call-stacks isn't very descriptive if the stack is down-growing. In languages like C and C++, the concept of a call stack is not even a part of its abstract machine definition, in order to allow freedom of implementation - especially for features like resumable functions. Call-stacks and alternatives are just implementation details.

In computer-science, there is talk of black-boxes. Programs, subprograms, and types should be black-boxes, with no coupling between the inside of a box and the outside of a box. In RAII languages, and therefore TOWEL, there is talk of objects being in-scope, and out-of-scope. TONAL takes inspiration from that model, hence considers the inside and outside of a func; of a scope.

From the outside (where the func is evaluated), the inside of a func (the func that is being evaluated) should be completely concealed. From the inside of a func, the outside should be completely occluded from view. The inside and outside are only connected via the parameters, the arguments, and the evaluation.

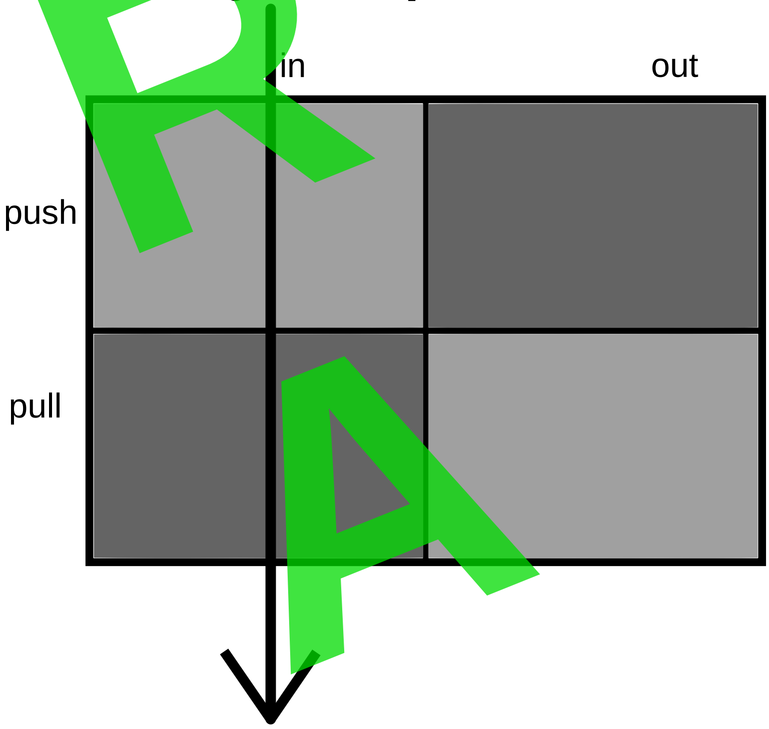
Figure 15.1: TOWEL transfer diagram legend.



Cross-boundary TOWEL transfers are cooperative because of this black-box principle. Whatever TOWEL transfers of arguments happen inside the function is completely invisible to the outside, and vice-versa. The programmer just has to tell the AM their intent; their constraints; their knowledge. In the common, best, case, the AM is allowed to do nothing. The outside can transfer TOWEL as they want, but if the inside doesn't do anything to accept the TOWEL, then nothing happens. The inside can transfer TOWEL and write to objects, but it can't affect the outside if the outside does not cooperate.

From the outside, the two TOWEL supertonics are push-in, and pull-out. From the inside, the two TOWEL supertonics are pull-in, and push-out. If a func is a box, then you can only push things into it, or pull things out of it. If you are inside a box, you can only pull things in, or push things out.

Figure 15.2: Cooperative inward transfer.



When you push something into a black-box, it's gone; out of reach. You no longer have any control of that thing. Programmers often construct an object just to pass it to another function, to never be used again. Pushing-in an object into a func in TONAL tells everyone that intent. The object is constructed and is no longer required there.

But to be able to push something into anything, you must have a hold of it. To push a let-name into a func, it must have TOWEL. Once pushed-in, the receiving func has the object's TOWEL - ie, responsible for destruction or transferring TOWEL - so the let-name is considered unreachable. An object that is constructed as a func argument as the result of a minor tonic is automatically pushed-in. A let-name that is a view of another object or an object with an enclosed object that is a view to another object cannot be pushed-in, and so, is a AM-error.

For a let-name to have an object's TOWEL, the object must either be func-local, or it must be pulled-in. From inside a func, all argument let-names are considered read-only views of an object until pulled-in. This includes the enclosing @object as in implicit argument, if the func was invoked as a dominant or subdominant form of an object. One reason to pull-in an argument, rather than creating a func-local copy, is to make a func more readable without creating so many func-local let-name, when a func argument is already well-named for the purpose.

The inside of the func has no idea whether or not an argument was pushed-in. The outside of the func cannot know whether or not a func pulled-in an argument. The AM does know, when in present-tense.

If the AM sees that an object was not pushed-in as an argument, and the func pulls-in an argument, the AM must maintain the black-box property of funcs by making a copy of the object. The outside of the func has the original object's TOWEL, while the inside of the func has a new object that it can write to, as if it was created as a func-local. If the object's type is not copyable, then it is a AM-error.

If the AM sees that an object was pushed-in as an argument, and the receiving func does not pull-in the argument, the inside of the receiving func still sees the argument let-name as a read-only view. Therefore no copy needs to be created. However, the receiving func itself has TOWEL over the object, so it is destroyed when the receiving func has finished, at the latest.

In languages like C++, the former is known as pass-by-value; the latter is known as pass-by-const-reference. In TONAL, this determination is made possible because the programmer tells the AM which lexical scope has TOWEL, and which lexical scope requested TOWEL. The programmer doesn't need to qualify a func parameter's type with const or reference qualifiers. The programmer knows whether they no longer need an object locally, so pushes in to communicate that knowledge. The programmer doesn't have to worry about whether or not there is an unnecessary copy on the other side.

If the AM sees that an object was pushed-in as an argument, and the func pulls-in the argument, then the AM can elide the copy, altogether. This avoids the AM-error for non-copyable objects. The AM might even determine that moving the object isn't necessary either and constructs an object in-place inside the receiving func.

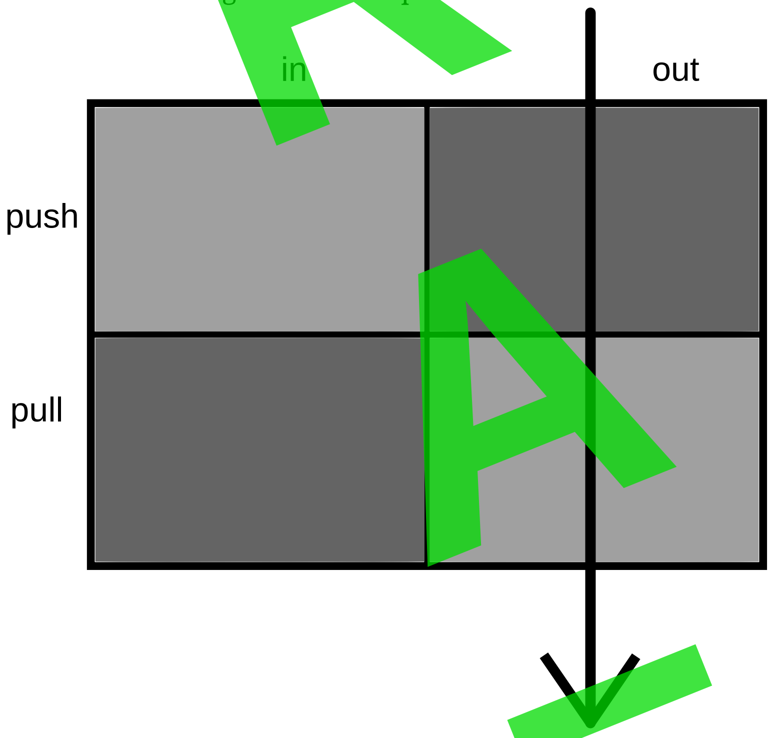
In C++, this is achieved either by constructing a temporary as an argument, or by moving an object. The function parameter would either be a value, or r-value reference-qualified. In TONAL, the AM makes this determination by what the programmer coded.

Func-local let-names that are views of other objects can also be pulled-in, and has the same TOWEL semantics as an argument.

The programmer only needs to tell TONAL whether or not an object is needed outside a func, and whether a func needs to write to an argument object, and the AM figures out the minimal and optimal code to generate in each case. Neither the outside or the inside of the func can make a mistake, like forgetting to destroy an object, or writing to the outside object, or accidentally making an unwanted copy. Both sides cooperate, rather than dictate what the other must do, or assume that requirements are met.

Resumable functions are a traditionally tricky case that is eliminated by cooperative inward transfer. When a resumable func is unsuspended, the handle to the func's control block should not be touched, because as soon as the func is unsuspended, it may have already started executing - perhaps on another thread - before the control is returned to the scheduler. In code, the scheduler still has the control block's let-name. But with cooperative inward transfer, the resumable func pulls in the control block. The control block's type is not copyable. If the scheduler code does not push-in the control block to cede TOWEL to the func, then the AM fails in the present-tense when trying to copy. The scheduler must push-in the control block. This causes the let-name to lose TOWEL, and any attempt to even refer to the let-name will cause the AM to fail in the present-tense.

Figure 15.3: Cooperative outward transfer.



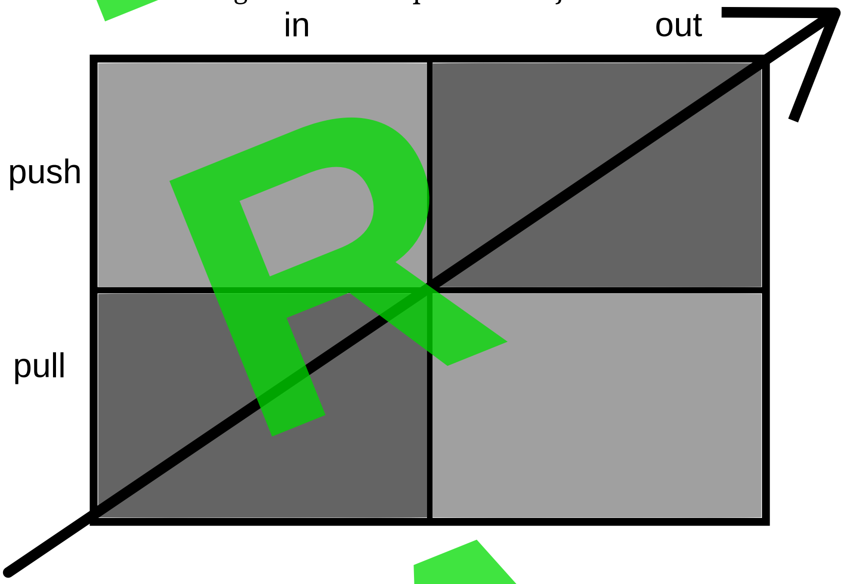
All let-names in a lexical scope with TOWEL are destroyed at the end of the lexical scope. However, all let-names with TOWEL are modifiable. Sometimes a programmer no longer needs to write to a let-name anymore - they are finished with the object. They can make it aware to the AM by pushing out a let-name. Unlike pushing something into a func, pushing something out of a func isn't an immediate transfer of TOWEL. In the interests of safety and usability, a pushed-out let-name is read-only. As long as the let-name is still in lexical scope, it can be referred to by a programmer. The AM could have somehow marked the let-name as unusable and give a AM-error when used, but that would defeat the purpose of allowing a let-name to be read-only.

A pushed-out let-name can be pulled-in again. This promotes a disciplined approach to mutability by encouraging the programmer to make the usage explicit; to clearly plan and demarcate the regions of mutability. It can show problem areas: if there's constant switching between pulling in and pushing out; or if something remains pulled-in for an entire lexical scope without being written to. Contrast this to traditional languages like C++ and Rust, where either something is non-const or mut forever in a lexical scope.

The AM could decide that a pushed-out let-name can be destroyed early. When the let-name is no longer referred to in a lexical scope, it can destroy the object, as long as it is able to maintain the destruction order guarantee. Every reference to a pushed-out let-name implicitly prolongs its life. Some types of objects should never be pushed-out in this manner. For example, mutex lock objects use the lexical scope to protect access. An accidental early destruction would make it extremely difficult for programmers to see such an error that can only occur sporadically in future-tense.

Evaluating a let-name is an implicit push-out. If the let-name is a view of an object, then it must be pulled-in when evaluating, subject to the copying rules.

Figure 15.4: Cooperative object transfer.



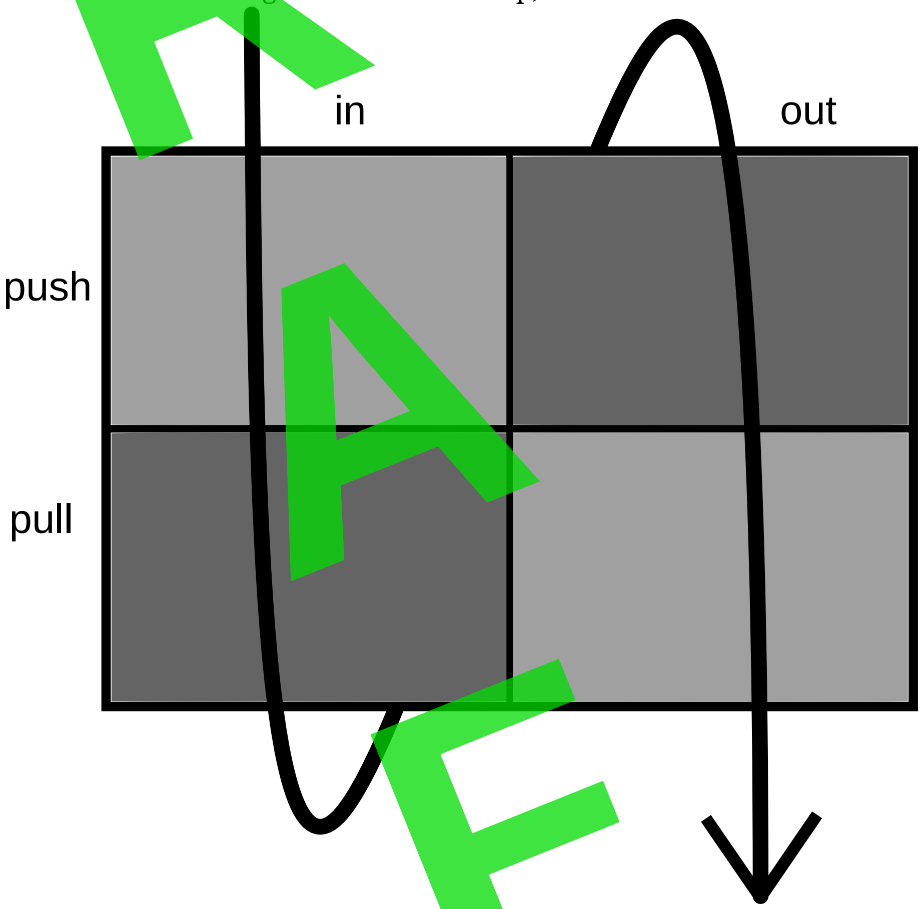
Enclosed objects have scopes and lifetimes that are dominated by the enclosing object. The enclosing object is subject to TOWEL, just like any other func parameter. In order to modify any of enclosed objects, the enclosing object must have been pulled-in at its own lexical scope, as well as pulled-in by the enclosed func.

It is an error to pull-in individual enclosed objects, because that introduces a notion of an object composed of different lifetimes, which is not something that's well studied. The enclosing object must therefore be completely pulled-in. Once pulled-in, all of the individual enclosed objects must NOT be destroyed. If the enclosing object is not pushed-out again before the end of the func, then the enclosing object is considered destroyed.

Enclosed objects, or the enclosing object, which are evaluated from a func, are readonly views, in the general case. If the enclosing object is pulled-in, then the enclosed objects are considered pulled-in, and therefore is evaluated in the same way as func-local objects. Similarly, if the enclosing object was pushed-in to some other func, then the enclosed object that was evaluated is implicitly pulled-in at the receiving scope, in order to avoid referencing an object that is destroyed.

15.3 Round-trip

Figure 15.5: Round-trip, no transfer.



The TOWEL round-trip is the serendipitous culmination of abstraction and intent-preservation resulting in performance with safety.

Some funcs play the role of a factory - they set up an object to some state that is beyond the scope of its constructor. This is usually achieved in programming languages through in-out parameters, which are typically implemented either as non-const reference or pointer parameters.

Per-usual, the problems with pointers apply. When it come to references, the main problem comes from C where a programmer, expecting to use a variable as an in-out argument, neglects to provide an initial value with a view to save a bit of time when the value is going to be overridden anyway. Over time, as code rot sets in (code gets modified, added to, deleted from), errors could be introduced where the value ends up being used before being initialized properly.

An object is passed to a func for modification via the TOWEL command “pull-out”. It evokes the imagery of pulling something out of a box. The object is implicitly pushed-in, so that if the receiving func pulls-n the respective parameter, no copy is made. When the receiving func pushes-out, the parameter object itself is modified. The pull-out completes the round-trip and object that was passed to the func is modified.

The AM detects these round-trips in present-tense and completely elide any copying in or out of the receiving func.

Round-trip semantics also apply to func evaluated values under similar conditions. An object is pushed-in to the receiving func. The receiving func pulls-in the respective argument, does something to it, then evaluates it. Outside the func, if the return-value is pushed-in to the let-name that was pushed-in to the func, then that constitutes a round-trip.

Evaluation round-trips are useful for implementing func chaining. Recall that an object evaluated from a minor-tonic is implicitly pushed-in to the major-tonic func call. As the dominant object is pushed-in and evaluated in a chained fashion, the TOWEL round-trip kicks in and elides any copying or moving of the dominant object, while also eliminating the need for explicit push-ins in subsequent func invocations.

Chaining is a common way to implement pipelines and builders but can be tricky to implement safely with conventional type-qualifiers. Composing a pipeline/builder that has to be passed along an if-else condition means using a let-name, which means with convention type-qualifiers, the programmer has to handle both the l-value reference case as well as the r-value reference case, but with a lot of care and safeguards to prevent letting a reference dangle easily. TOWEL makes it easy to switch between different modes pipelining with the round-trip detection.

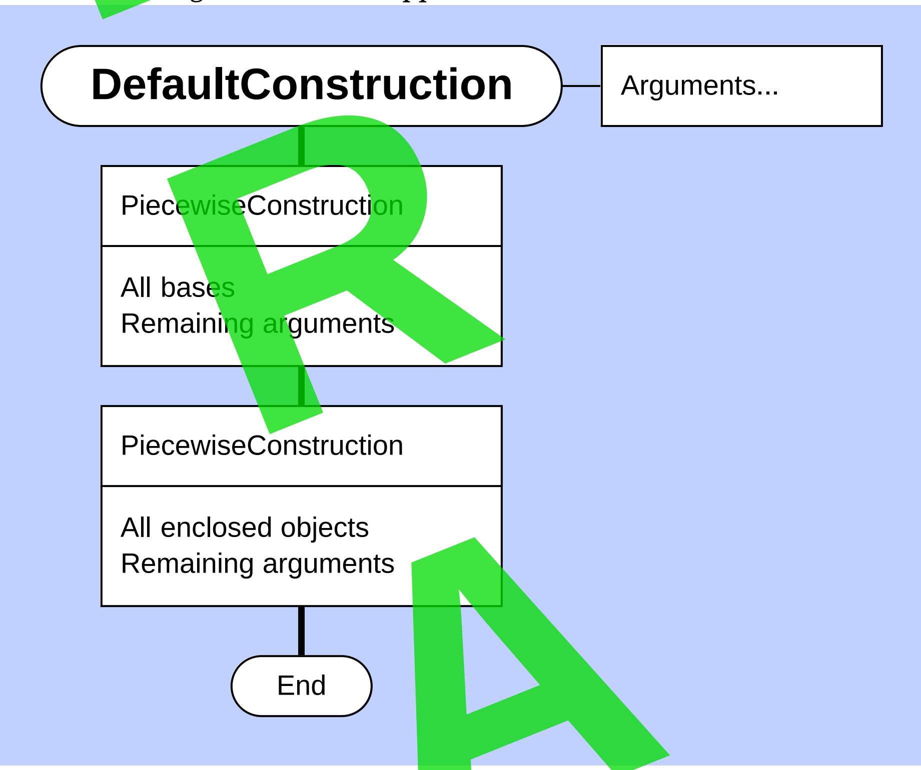
15.4 TOWEL hooks

15.4.1 Construction

Construction is the process by which the AM creates an object, giving value(s) a type and identity. Allocated memory cannot be read until an object has been constructed in it. An object is not fully constructed unless its base and enclosed objects are constructed. Base objects are constructed in a left-to-right order. Enclosed objects are constructed in a top-to-bottom order, after the base objects have been constructed. The AM maintains this invariant order of construction, even if a type specifies their own special contruction process.

The AM also supplies construction processes that makes the language work.

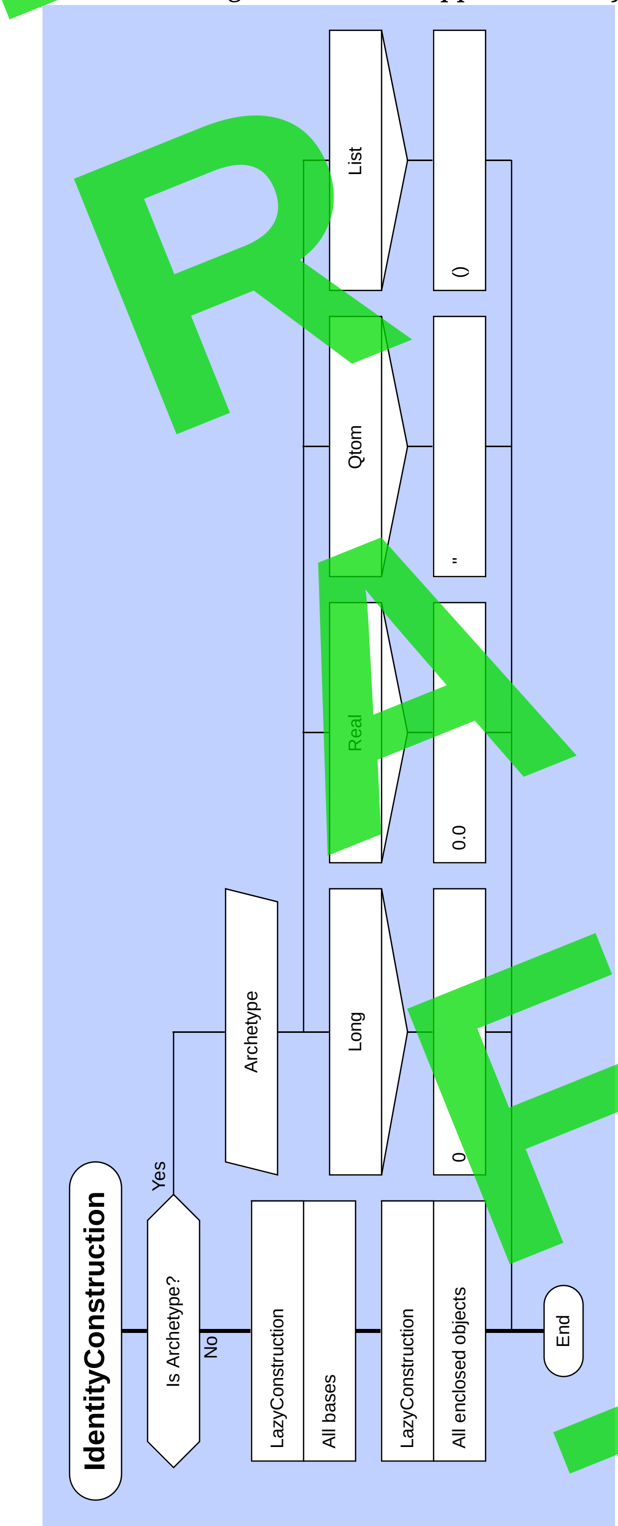
Figure 15.6: AM-supplied default construction.



The default way to construct an object is to provide argument values for each of the type's enclosed base objects, in left-to-right order, and values for each enclosed object, in top-to-bottom order. Not only is it the default supplied by the AM, it is the default in the sense that every type is constructed in this prescribed order. No matter how a programmer might define a custom construction process, the AM ensures that the order of construction is the same as the default construction.

Other languages call the constructor that takes no arguments the default constructor, but in TONAL, "default" is more suitable to denote the order of events in all cases, regardless of construction argument counts or which construction process is being used.

Figure 15.7: AM-supplied identity construction.



TONAL gives the no-argument construction the concept of identity. It borrows the sense from algebra, where a special element in a set under a given operation is the identity. Though not always the case, it is helpful to think of a value constructed by identity construction as similar to additive or multiplicative identities. For many types there are no suitable identities, which can itself be a useful property to consider.

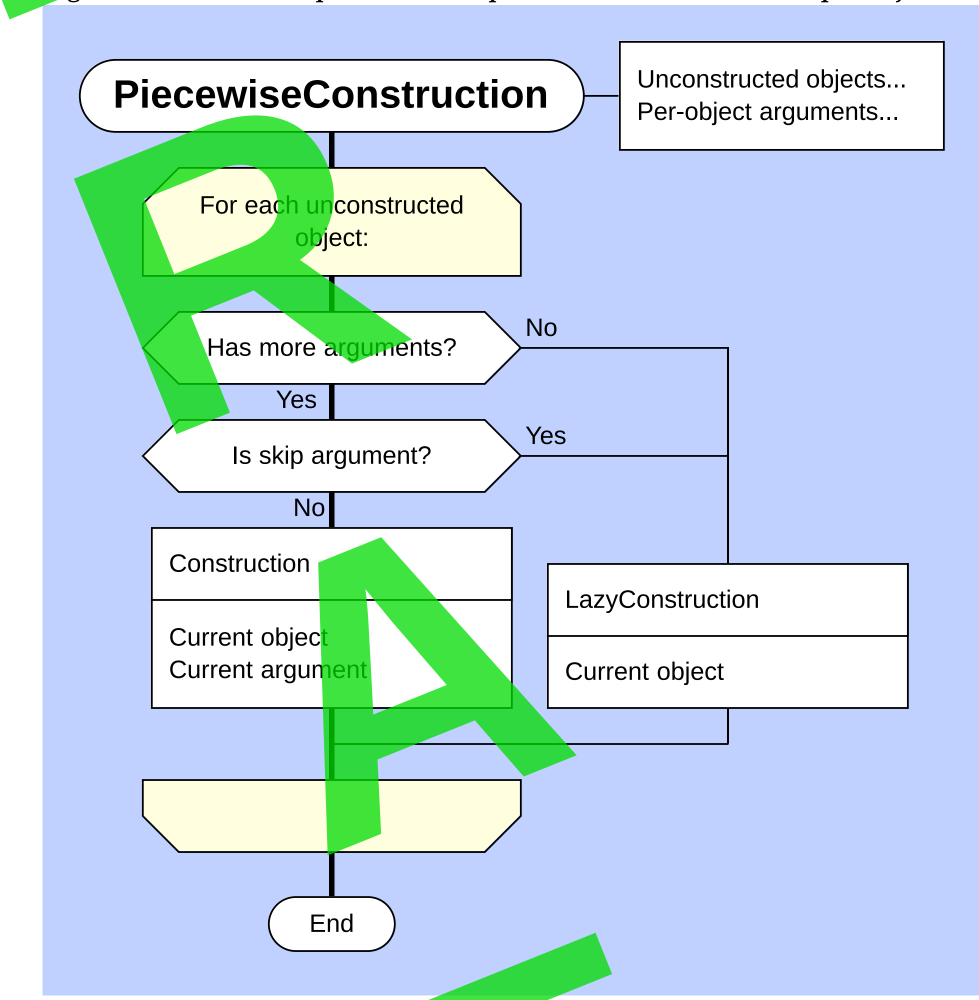
For non-Tensah archetypes, their identity is the equivalent of the integer additive identity 0. The concept of 0 and empty sequences as the identity construction carries through to all types that take after Scalah and Vectah rchetypes, like machine number types and sequence types. Empty Vecta archetypes values are particularly useful as the primary way of denoting nothingness and avoids all the costly mistakes of having some null value that may or may not be equivalent to 0. 0 is left to its rightful place as an extant value for a scalah type, and not as a error-prone double-meaning for a non-existant value.

Tensah archetypes cannot meaningfully have an identity value. Programmers must specify the value for any type or func object, otherwise the whole program is in error. No program can be valid if a type is unknown, or a func has no value. If there is some meaningful fallback value in some operations, then that has to be explicitly supplied. The AM must not guess at the programmer's intentions.

For non-archetypes, enclosed objects of a type under identity construction does not necessarily have to have an identity value. A type may have a zero-argument construction that itself provides the values for its enclosed objects that make up its identity value. The value could be specified as a fallback alue as part of the type's definition, or given during construction.

If any of the enclosed object types cannot be identity constructed or provided a value during identity construction, then the type is said not be identity constructible.

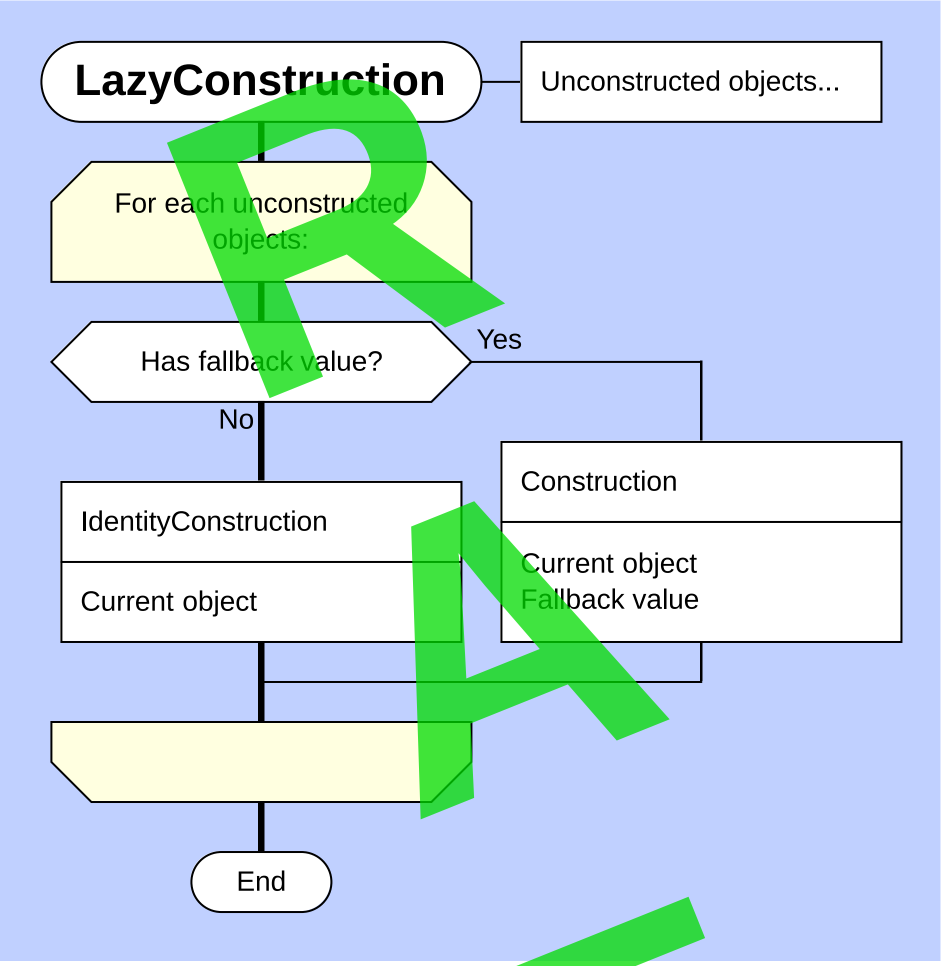
Figure 15.8: Common process for explicit construction of multiple objects.



All available arguments are used to construct each enclosed object for a default construction. In the main case, a programmer provides all arguments needed for each enclosed base and non-base objects. However, they may omit the tail of the arguments if there are suitable values, such as if the objects are identity constructible, or the type definition has fallback values for the remaining enclosed objects.

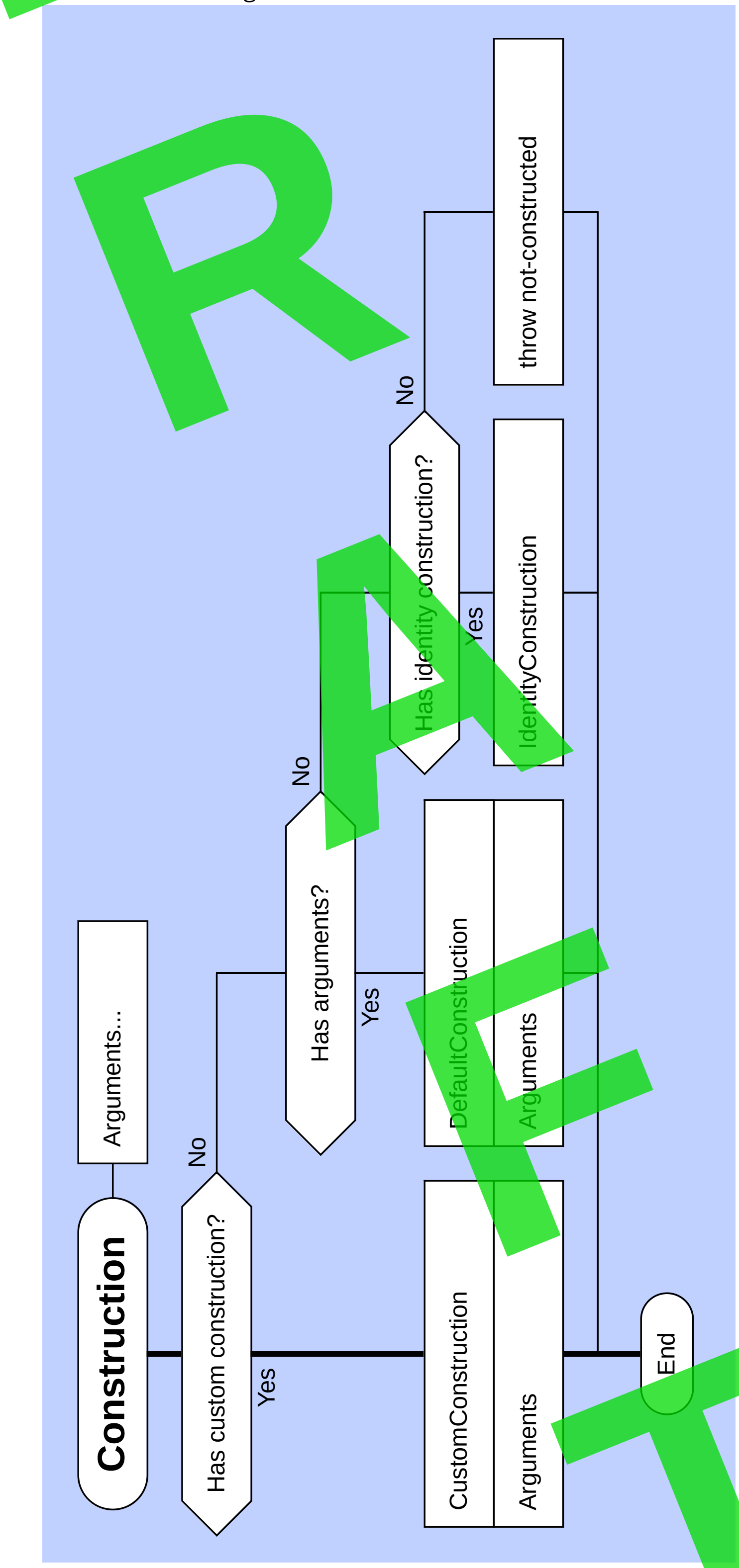
Alternatively, a programmer can provide a skip argument which will also allow the corresponding enclosed object in the default construction order to be constructed with a fallback or identity value.

Figure 15.9: Common process for incidental construction of multiple objects.



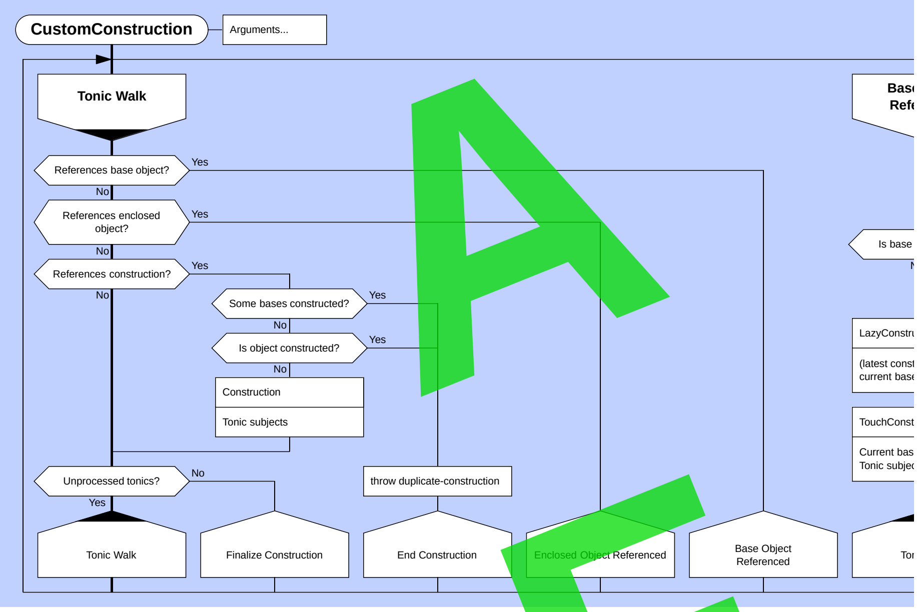
Objects are constructed at the latest possible stage - the first read of the object or its enclosed objects - but always in the default construction order. The construction might not be explicit; it might be done to keep the invariant of the default construction order.

Figure 15.10: General construction selection.



Any custom-defined construction process takes precedence over any AM-provided one. A custom construction process may defer to other custom constructions, the default or the identity construction. This allows constructions to be built-up from other constructions. This is what is termed correct-by-construction. Proper initialization of values happen at construction, even if the construction is delegated, instead of being left in a partially-initialized state that is then left over for the programmer to forget to invoke a needed non-construction func to complete.

Figure 15.11: Library-given custom construction (tonic walk).

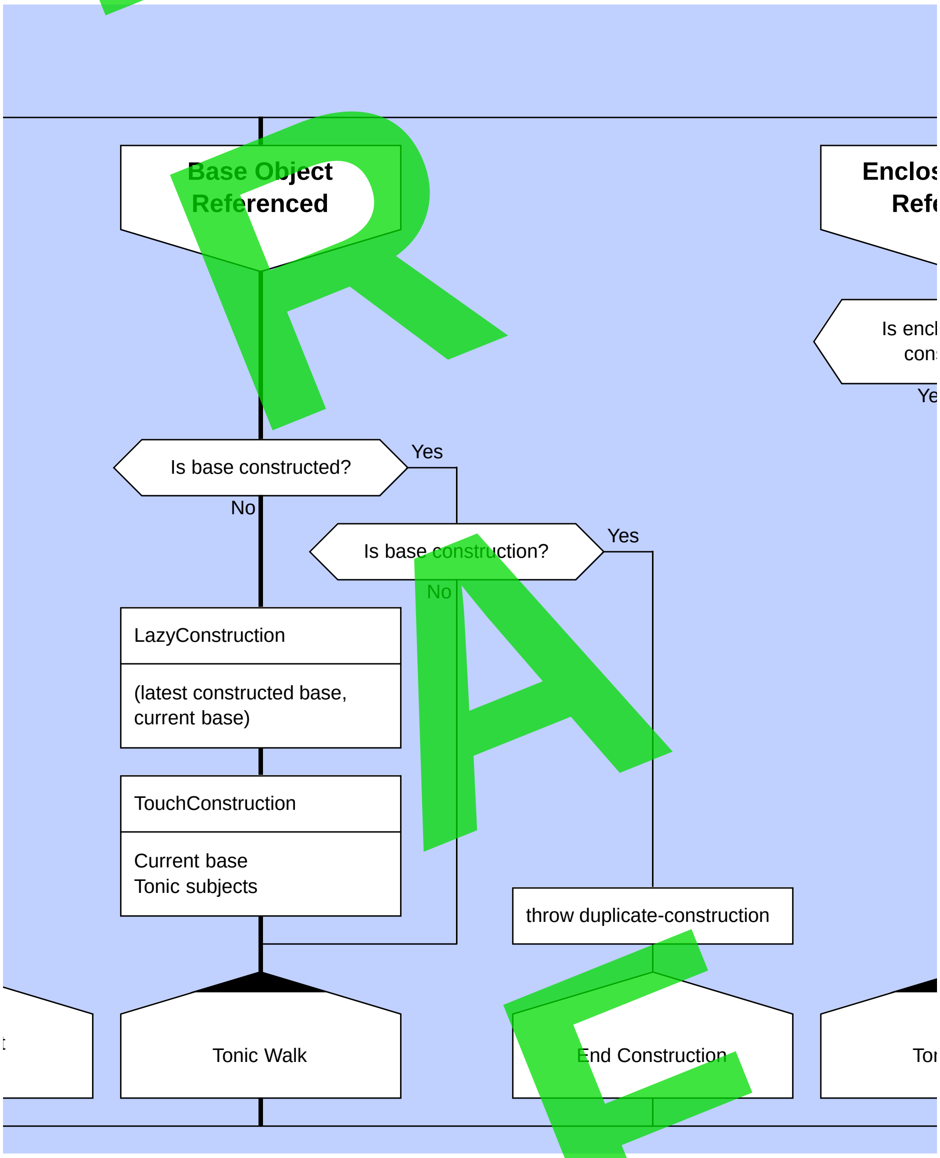


Constructors in other languages maintain their construction order by limit where constructors can be called, where members are initialized, and the precise order of those actions.

The AM ensures that the default construction order is adhered to, while not limiting how a programmer defines a custom construction. The general method of ensuring the default construction order is that any base object or enclosing object that is refered to for its first time is initialized. The value that it is initialized with is, in order of priority: 1) the value that it is being constructed with, or 2) the fallback value, or 3) the object type's identity construction. If all three are not defined, it is an AM-error.

The AM keeps track of which enclosed objects have been initialized. This ensures objects only get initialized once. Say for example there are enclosed objects A, B, and C. If A has already been initialized, and then the next object to be mentioned is C, then B is initialized by the AM before initializing C. It is an AM-error if a construction for a base, or another custom construction for the type is invoked more than once. This allows custom constructions to be written in a more natural form. Commands can thus be interspersed with constructions and initializations without breaking the initialization order guarantee.

Figure 15.12: Library-given custom construction (base object).

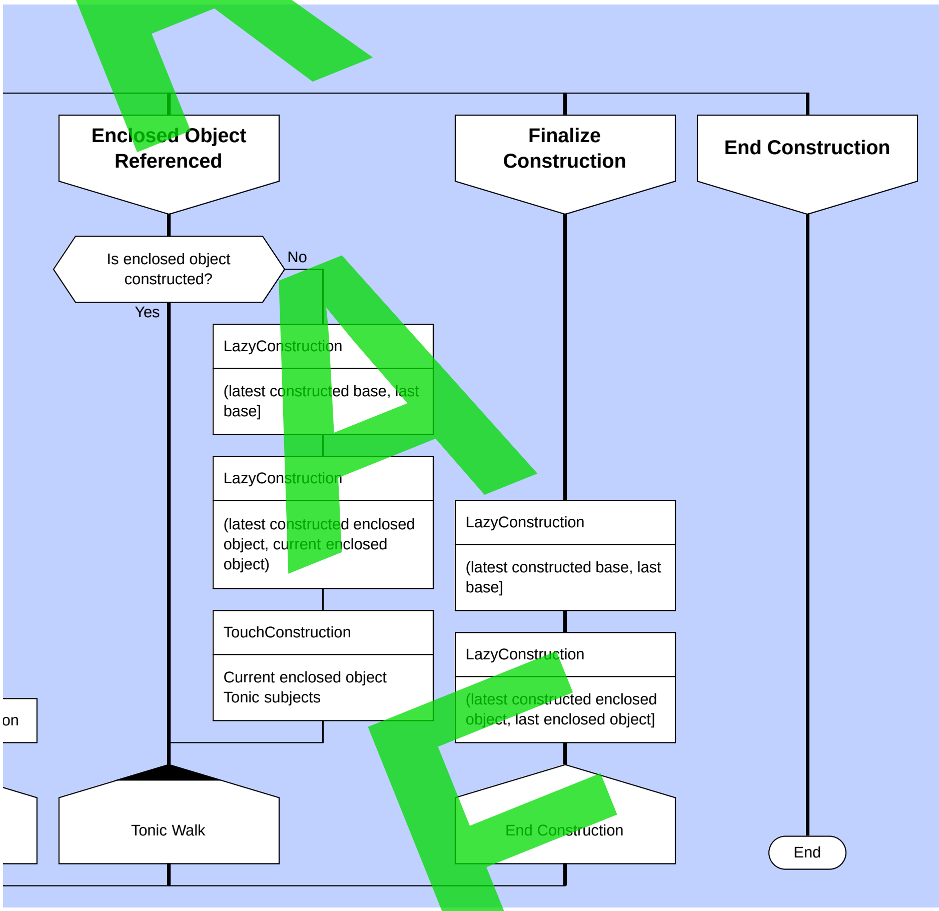


Base objects are constructed left to right. Due to the construction order guarantee, or more generally, the requirement that an object's bases are constructed before the rest of the object is, bases are implicitly constructed depth-first recursive ascent.

If a base's enclosed object is refered to, the construction order guarantee must be applied recursively.

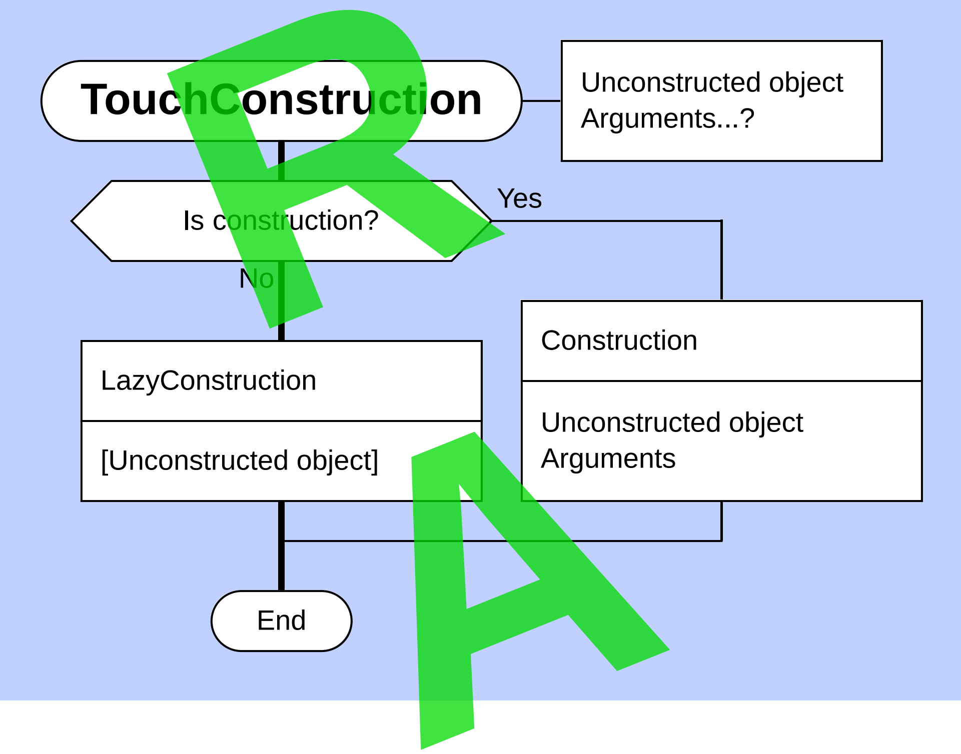
Any virtual funcs that are invoked during construction do not dispatch. The effective type of the object at the moment of invoke a virtual func is the type of which the current construction process is defined for. It is not any of its base types or a derived type. No dispatch is necessary because the type is set/known.

Figure 15.13: Library-given custom construction (enclosed object till end).


Enclosed Object Referenced
Finalize Construction
End Construction
Is enclosed object constructed?
No
Yes
LazyConstruction
(latest constructed base, last base]
LazyConstruction
(latest constructed enclosed object, current enclosed object)
TouchConstruction
Current enclosed object Tonic subjects
LazyConstruction
(latest constructed base, last base]
LazyConstruction
(latest constructed enclosed object, last enclosed object]
on
Tonic Walk
End Construction
End

Any base or enclosed object that is not constructed, either explicitly or via the construction order guarantee when referencing an enclosed object or base, by the end of a construction is constructed. This adheres to the principle of laziness. An object must be fully constructed at the end of a construction, but construction of constituent objects and bases happen as late as possible during the construction - when they are first read.

Figure 15.14: Common process for explicit or incidental construction of latest unconstructed object.



An object may not be constructed until it, or its constituent objects, are read from. Construction is delayed until first read, leaving open the possibility for collapsing the first construction and the multiple writes up to the first read. In languages like C (and C++ by extension), a programmer can declare a variable without a value, and create security issues from using the variable without initializing it. The idea was that to specify a value when one could not be known yet was wasteful. One may also argue it would be less readable to assign two values to a variable without doing anything to it in between the two assignments.

There's nothing inherently wrong about waiting until the latest moment possible before finally setting a value to a variable, but it is easy to forget. TONAL's touch construction ensures that there can never be accesses of memory without an object being constructed in it. The first read of a variable is when it matters most that there is always a known, and valid, value, so construction followed by zero-or-more consecutive writes can be elided and substituted with the last written value before it is used.

It is so named to be reminiscent of the common touch utility that modifies a file's modification time, or creates one if it doesn't exist. Touch construction, likewise, means something is ensured to exist in a timely fashion.

15.4.2 Copy Construction

Copying takes one object of the same type and creates a new object of that type. The default implementation of copy construction proceeds just as Default Construction does, with each argument being the respective enclosed object from the source encloser object, to make an exact copy of the source. Operations on the source object does not affect the copy, as one would expect.

Some types are defined to be uncopyable. If a base object or an enclosed object of a type is uncopyable, then that type is also uncopyable.

Programmers can define a custom copy construction for special semantics. Most commonly it is to define a type that is uncopyable, for instance, to model something that should always be unique. Another common reason is to share resources between objects.

Copy construction is not called by the programmer, but decided by the AM. The AM minimizes copying objects and prefers to construct an object in its final location, enabled by Lazy Construction. Copying is only performed if the AM is unable construct an object in its final location.

A programmer can create a let-name variable from an existing object, but no copies are made until either the source or the copy are modified, if allowed.

15.4.3 Relaying

A closed file cannot be read-from or written-to. It may be opened or deleted. Once deleted, it can no longer be opened. Once opened for reading, it may be read from. Once opened for writing, it may be read-from or written-to. At each stage, you have some handle to a file, but the things you can do with it depends on the state it's in. The states transitions can be said to result from certain operations. eg, from the closed state, after opening a file, the file is in an open state. Closing the file would put it back in a closed state. Each state, with restricted operations, are good candidates for a type.

A type may be used to represent a stage of a process pipeline. Data that is computed during previous stages may need to be kept around for future stages, but maybe shouldn't be exposed in some of those stages. Each stage of that process may have only a subset of valid operations on a subset of that data, and that is documented/access-controlled by the type.

It would be wasteful, both in programmer/testing time, and in present/future-tense time and memory, to create a bunch of types and objects for each stage, moving data between them, and having old stages lying around, taking up space, and being incorrectly used. It would be unproductive and error-prone to have one massive type with every possible operation for every possible stage, with only documentation (if any) to tell programmers what operations are valid at which stage, and the valid order of the stages, or indeed multiple paths through the stages.

Relaying allows a let-name to be imbued with a new type, without necessarily having to transfer data to a temporary location. The let-name from that point on cannot be accessed as the old type. It is as though a new object was created in its place, and the old object was destroyed. The AM does not provide any relaying construction for any type.

A programmer may define a relaying that does nothing, in which case the representation of the object stays as it is, but is considered to have a new type, of course providing that the new type has the same underlying representation. Complex pipelines are unlikely to keep all data around at all times, and their respective types shouldn't have so much visual noise of data that is not being used, so sometimes it is necessary to move data to a temporary place while the new object is being constructed.

An easy way to implement a design that is amenable to relaying is to have a suite of types that derive from a common base type. That way, they can relay to each other without massive changes to the underlying representation.

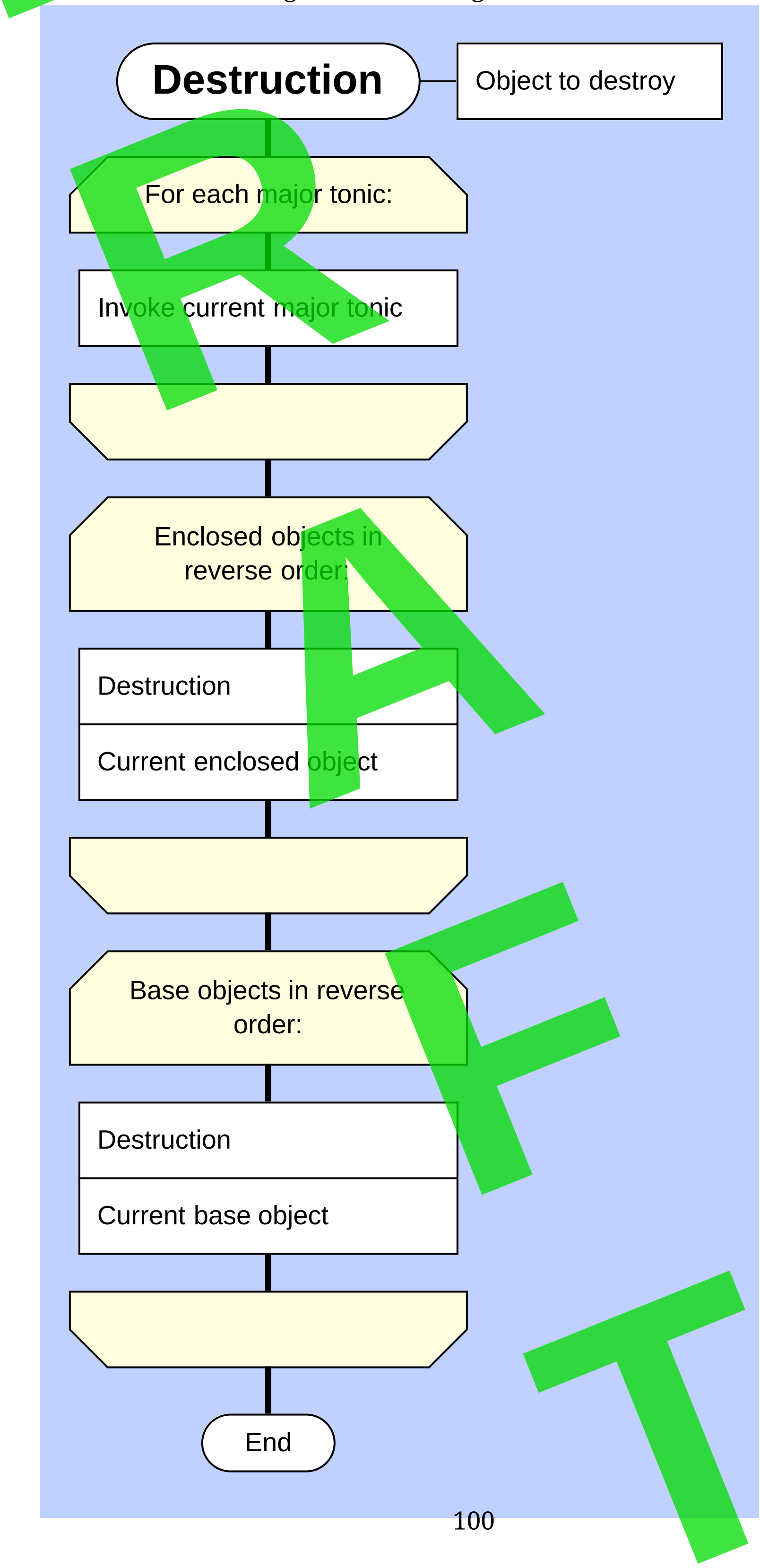
In an imperative body, types involved in relaying can be any size in present-tense. In future-tense, the object's size is the maximum of all the types involved. As enclosed objects, its size must be the same as the initial type's size. Objects declared in declarative bodies cannot be relayed, as the enclosing type must assume all of its enclosed object's types are the same.

15.4.4 Destruction

Something that is acquired (eg, extra memory, files, sockets, mutexes, database connections, etc) must be released when no longer in use; and failure to do so is a leak. Using something that has been released - including releasing something that has already been released - can lead to security errors. Human programmers tend to either forget to release things, or forget they've already released something, or are bamboozle by confusing or fast-changing code into believing they don't need to release, or that they do need to release.

TONAL calls the releasing "destruction", like other languages. Putting the language (AM) in charge of releasing things instead of leaving it up to the programmer dramatically improves quality and productivity. It is called RAII in C++. In TONAL, 15. TONAL goes further and prevents the let-name from being accidentally used after being destroyed.

Figure 15.15: AM-given destruction.



Just as construction has the top-to-bottom, left-to-right, order guarantee, destruction is guaranteed to be in exact reverse. So for example, if an older object is being destroyed, it will cause newer objects to be destroyed first. Derived types are destroyed before the base types, so if a virtual func is called during destruction, no dispatching to overrides takes place. The func of the current type of the object being destroyed is called, because the derived type has already been destroyed.

Destruction can happen when a let-name goes out of scope. Destruction can happen when a condition is tripped but not yet trapped. If a condition is tripped during construction, only the constructed sub-objects and bases are destroyed. If a condition escapes the destruction of an object - ie, not the current condition that is inducing the destruction of an object, then the entire execution context is aborted. There is no sensible state to recover to when an object cannot be destroyed.

In TONAL, all destruction behaves like a virtual func. Destruction always starts at the most derived type and from there follows the destruction order guarantee.

16 Let-Name Lookup and Pruning

Naming things is the hardest thing to do in all of computer science, and science in general. The next hardest is being able to find all the names. Either names became extremely long just to differentiate from each other, or they're short and enclosed within other names in order to not. Names can mean different things under different circumstances.

Like most modern languages, TOWEL names are lexically scoped. They depend only on its location in the text of the code, not on the state of the program. Names can be enclosed by Tensah archetypes, creating a ladder of enclosing scopes. In TOWEL, this term is shortened to "enclosure". Tensah names can be overloaded in present-tense and future-tense.

The general scheme for let-name lookup is to start from where a name is being mentioned. The name is searched for enclosure by enclosure on the ladder of enclosures. The search stops when the name is found in an enclosure. A [Tensah-]let-name is a candidate to be looked up as soon is it has stopped being spelt. If the name cannot be found, the AM fails.

When a name is found in an enclosure, all objects with that name forms an overload-set, from which the AM tries to prune down to one suitable candidate. If it cannot be done, the AM fails.

The redirection operator fundamentally redefines the semantics of name lookup. The redirection operator is essentially a program that runs in present-tense, and so is a black-box to the AM itself. If a redirection operator is defined, then if name lookup reaches the enclosure of the redirection operator, the redirection operator is always chosen as the first and only candidate. The redirection operator then runs in present-tense to either produce a single object, or trips a condition to either cause the AM to fail, or to resume the normal name lookup rules at a higher enclosure.

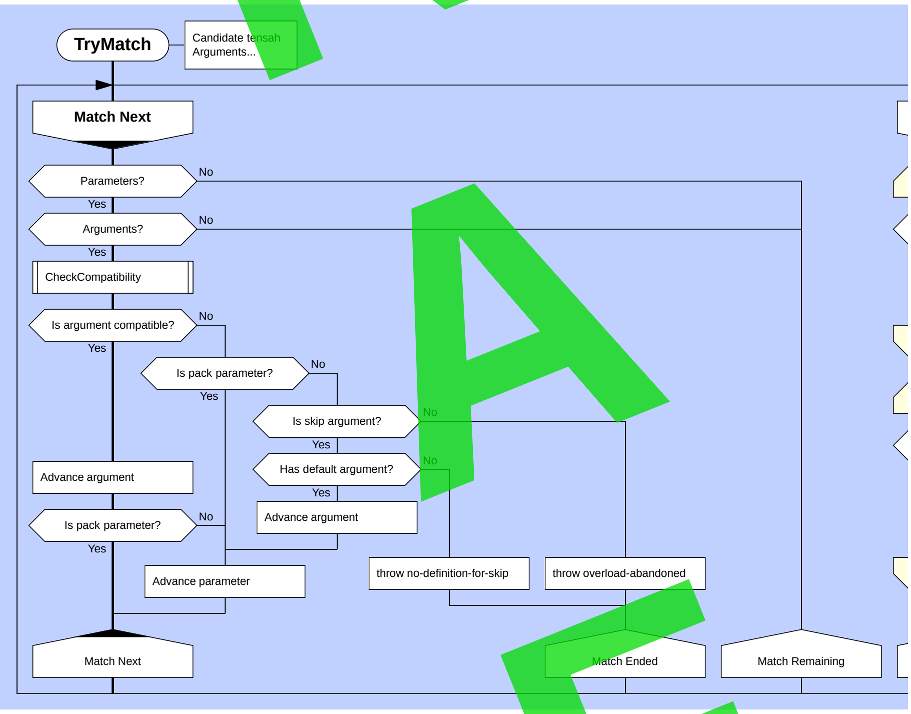
During the operation of redirection, the redirection operator is removed from the candidates of name lookup while inside the redirection operator. This prevents infinite recursion. Name lookup from within the redirection operator is upward limited to the redirection operator's enclosing type's enclosed objects.

When the enclosure being searched is a type, all matching names are gathered into an overload-set in any order. Everywhere else, the names are gathered in reverse order beginning from where the name is mentioned - essentially reverse-construction order. Matching names gathered via reverse-construction order are gathered into an overload-set.

An overload-set can only contain Tensahs enclosed by a type or an object with a type archetype (ie, not a func). An overload-set can be passed around between funcs. It is just a list of Tensahs that had been gathered by name lookup, and can be treated like any other present-tense list. An overload-set can also be used in the verb position of a command.

An overload-set cannot be empty initially, as one would not be created in the first place with name lookup cannot find a name. An overload-set of cardinality 1 will invoke the sole element as the verb, provided that the subjects are valid arguments to the Tensah. An overload-set of cardinality > 1 will need to be pruned before invoke to avoid ambiguity.

Figure 16.1: Matching parameters.



One innovation in TONAL is the generalization of parameter specification with the concepts of pack-parameters and skip-arguments. Variadic functions in other languages tend to only allow the extra arguments at the end of the other parameters. In languages that also support default-arguments, those parameters must also be specified at the end of the other parameters, and therefore do not play well with variadic parameters.

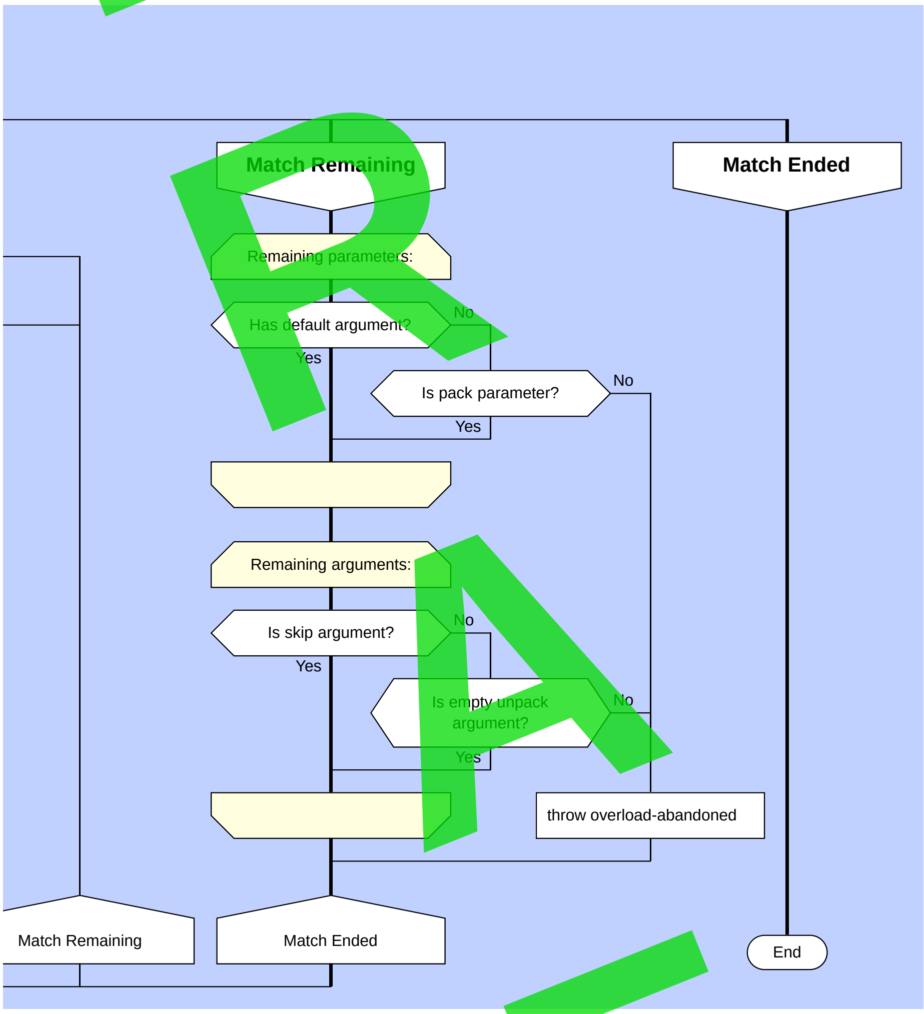
TONAL allows normal parameters, pack-parameters, and default parameters in any order without ambiguity. The mechanism that helps achieve this is the skip-argument.

If an argument is compatible with a pack-parameter, then the pack-parameter stores the argument. If an argument is incompatible with the pack-parameter, then the pack-parameter is completed.

Consider the case of two pack-parameters, of the same type, in succession. How does TONAL decide when the first pack-parameter is complete, when all the arguments are the same type also? In general, the first pack-parameter stores all the arguments, and the second pack-parameter is empty. Additionally, TONAL gives programmers the option to explicitly complete a pack-parameter with a skip-argument.

TONAL allows default-arguments in positions other than the end. It would defeat the purpose of default-arguments if the programmer has to provide an argument in order to provide the subsequent argument. The skip-argument is used in this case to allow the programmer elide the argument. The default-argument is created by the AM for the parameter, as though the parameter was at the end of the parameter list and no argument was provided.

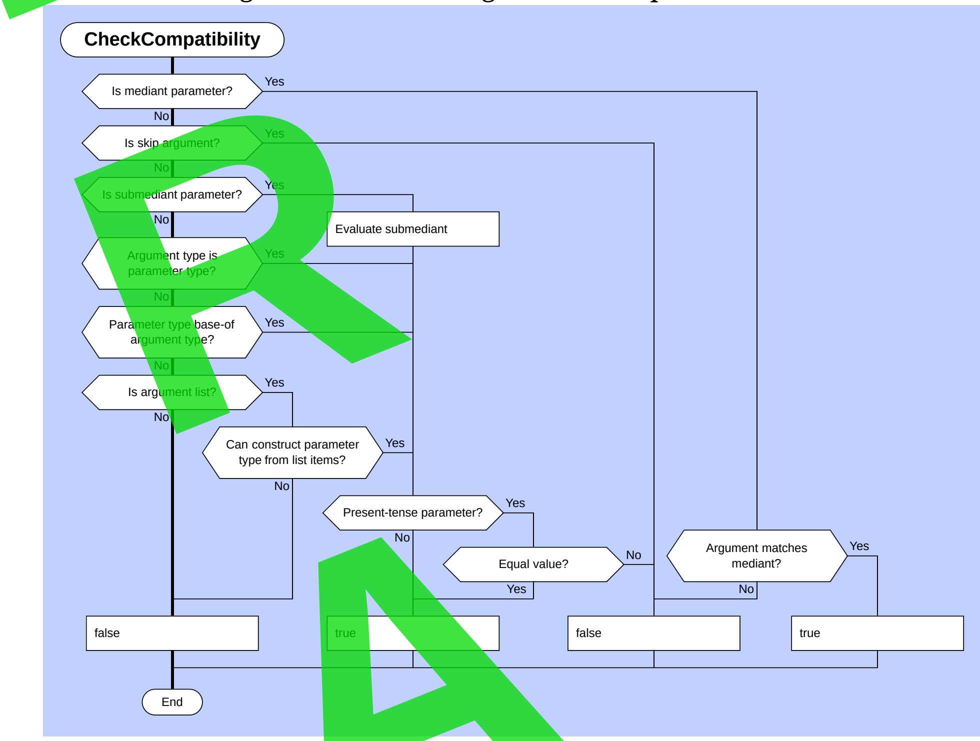
Figure 16.2: Matching parameters (continued).



A TONAL programmer is free to provide as many skip-arguments even if there are no parameters left to fulfill. This simplifies generic programming so that the programmer doesn't have to know the exact number of parameters of every possible context.

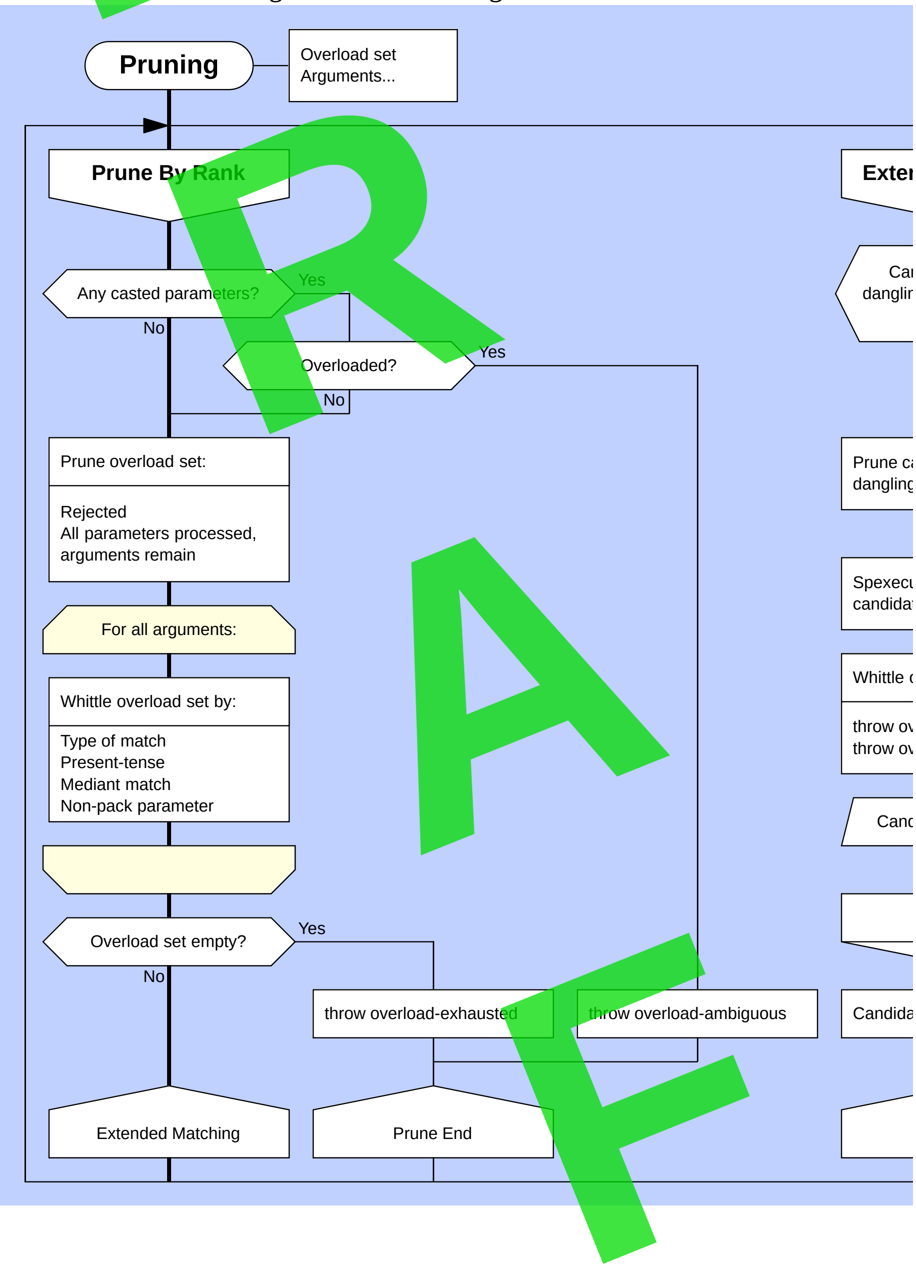
Unpack-arguments that happen to be empty are also treated the same way.

Figure 16.3: Check argument with parameter.



Exact types - and exact values in present-tense - matches are mandatory if an overload-set has more than one candidate, but at this point of the process, we don't know how many candidates there are in the overload-set, so inexact matches are allowed. Derived type and implicit convertibility/constructibility (imcluding type casts in some languages like C++) inexact matches keep a candidate in the running the overload-set.

Figure 16.4: Pruning overload-set.

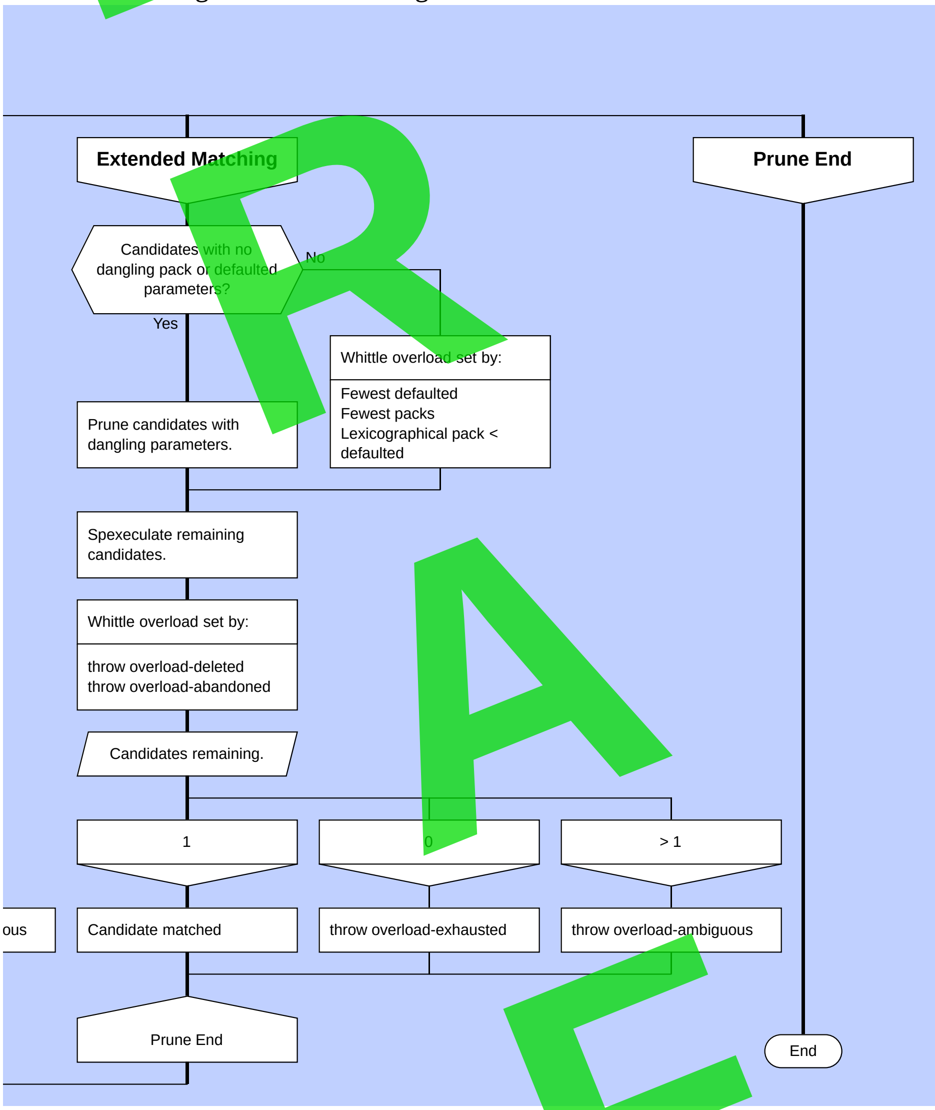


Overload pruning considers fulfilled-arguments to parameters from left to right. Assume that the AM takes the *i*-th parameter from each overload and prunes all the candidates whose current parameter being considered does not match the current argument being considered.

In an overload-set with more than one member - both pruned and unpruned, any inexact matches causes the pruning to fail due to ambiguity. The AM must not try to guess at what the programmer means by trying to find the best match, as there is no good classification preferred by every one. It is less error prone for both the AM and other programmers to force the programmer to explicitly construct objects of the correct exact type in order to document which overload they desire.

Implicit conversions are the source of a lot of errors in other programming languages because programmers can never have complete understanding of a language's details. It makes overload resolution seem almost random some times.

Figure 16.5: Pruning overload-set (continued).



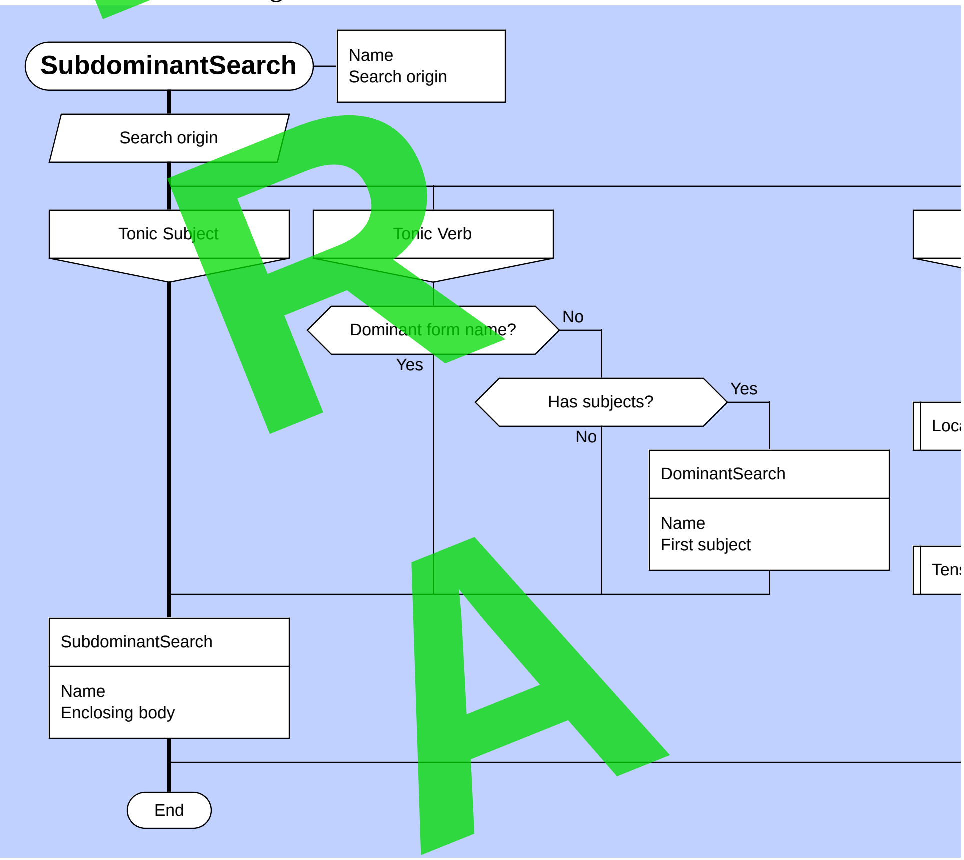
Consider the case where two Tensah candidates in the overload-set have used up all available arguments, but they both have zero-or-more pack-parameters or parameters with default-arguments that are left unfulfilled. TONAL, just like other languages with overloads, operate on the principle of tightest match wins, so in the event of dangling parameters, cardinality is the deciding factor.

In the contest between pack-parameters and default-arguments, the cardinality of a parameter with a default-argument is considered to be 1, while the cardinality of the empty pack-parameter is considered 0. The reason this is the case is because, even though the programmer does not provide an argument, the AM has to create one. It is physically more expensive than an empty pack-parameter, and so can be considered a looser match, in much the same way a function with *n-1* arguments is a tighter match than a function with *n* arguments.

Having established that, the principle can be specified as: the fewest number of default-arguments is the tightest match. Then, in the case of equal number of default-arguments, the fewest number of pack-parameters is the tightest match. Then, in the case of equal number of default-arguments and pack-parameters, only then are their left-to-right order taken into account. Earliest pack-parameter wins, for the same reason elaborated previously.

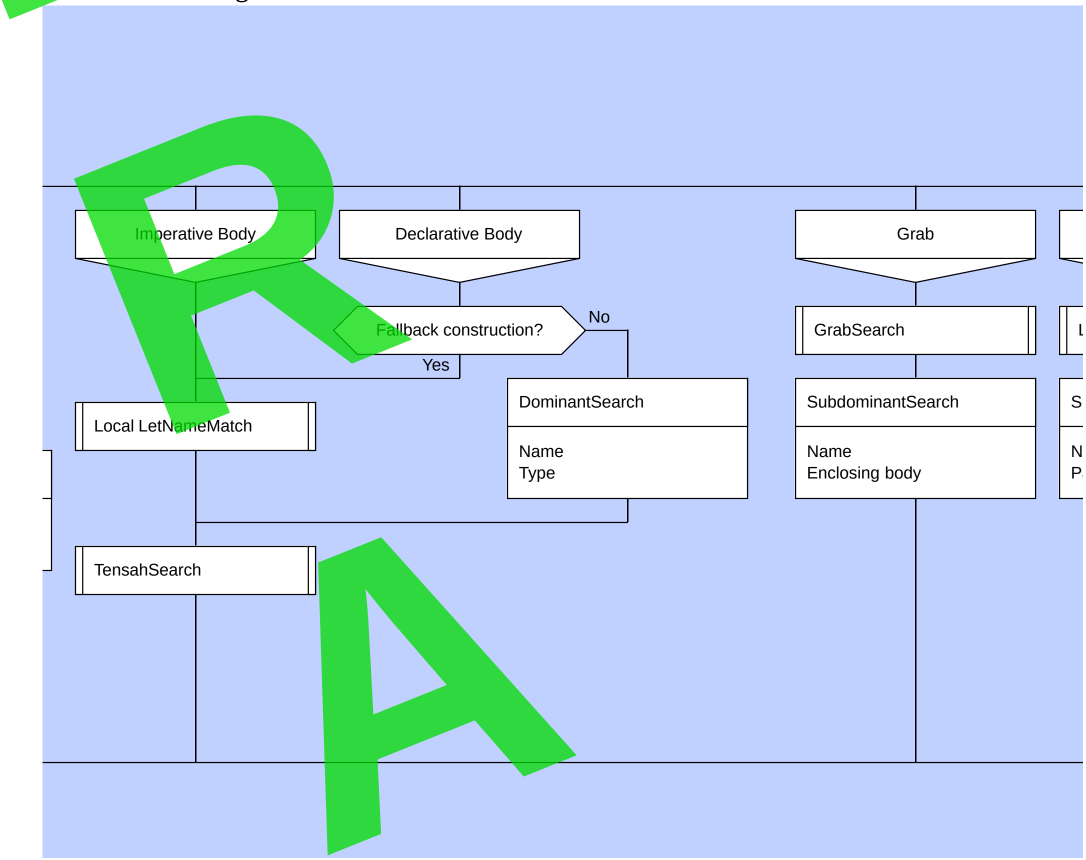
Programmers can further control the process during the speculative-execution step: spexeculation. Programmers can specify, in a more natural, imperative, manner, in the present-tense, constraints on the parameters, beyond types and present-tense values, by tripping conditions in order to remove an overload candidate from the overload-set. Spexeculation happens regardless of the size of the remaining overload-set, since further processing can still disqualify an overload candidate and inform the programmer that the program is incorrect.

Figure 16.6: Subdominant Search.



All name searches begin as subdominant searches, because all names begin with a subdominant component. Recall that a dominant name is dot separated, and by definition the first segment is not dot separated, so the first segment is found via subdominant search.

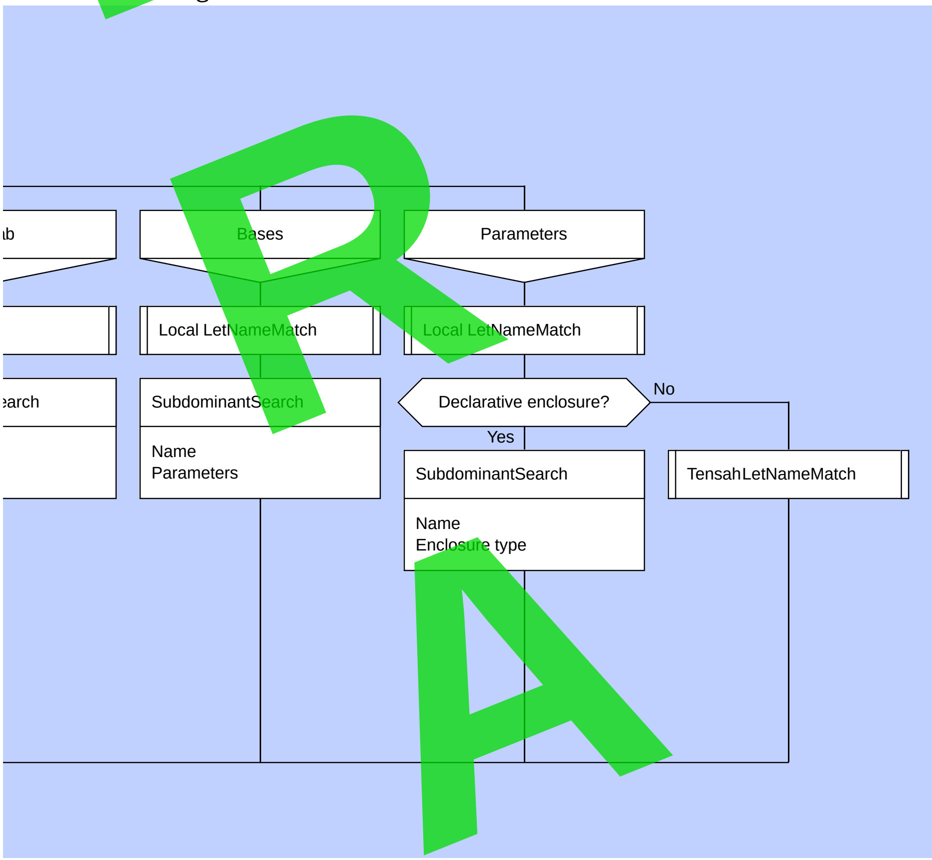
Figure 16.7: Subdominant Search (continued).



For the most part, name search looks like a simple current-row-to-top, current-column-to-left (ie, reverse construction order) of code in an editor. The body of a type is searched in this same order only during fallback construction, and only for non-Tensah let-names. Once inside a constructor, or in search for Tensah let-names, then all names are considered.

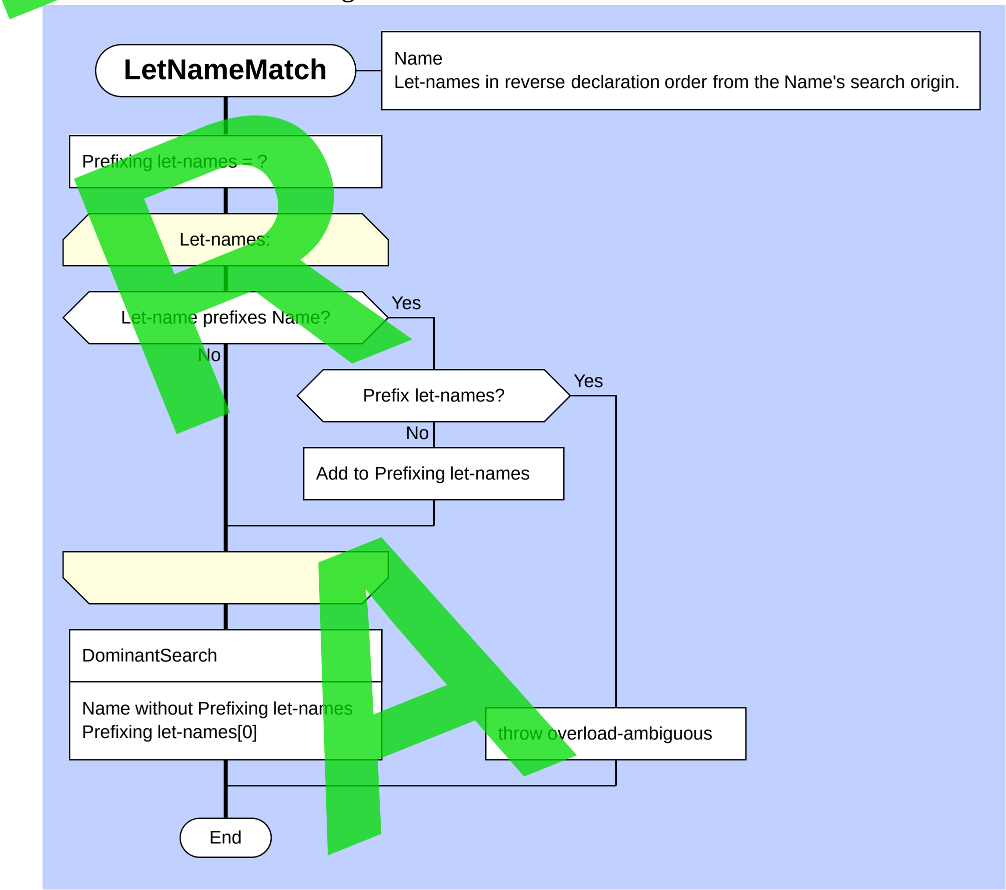
If a name search starts within a grab supertonic, it can only search for names that have already been grabbed, and if not found there, does not search parameters, but instead searches the enclosing Tensah body and so on and so forth, since the purpose is to grab let-names from the encloser. Parameters are already accessible from within a Tensah already.

Figure 16.8: Subdominant Search (continued).



In order to allow recursion for local Tensahs (those not enclosed directly by a type), the name of the Tensah is available as a candidate for name search as soon as the name is finished being spelled. Type enclosed Tensahs do not need this ability, because the name will be found when the enclosing type's let-names are searched. Tensah names enclosed by a type can also be overloaded, so direct recursion may not be the desired reason to refer to another Tensah of the same name.

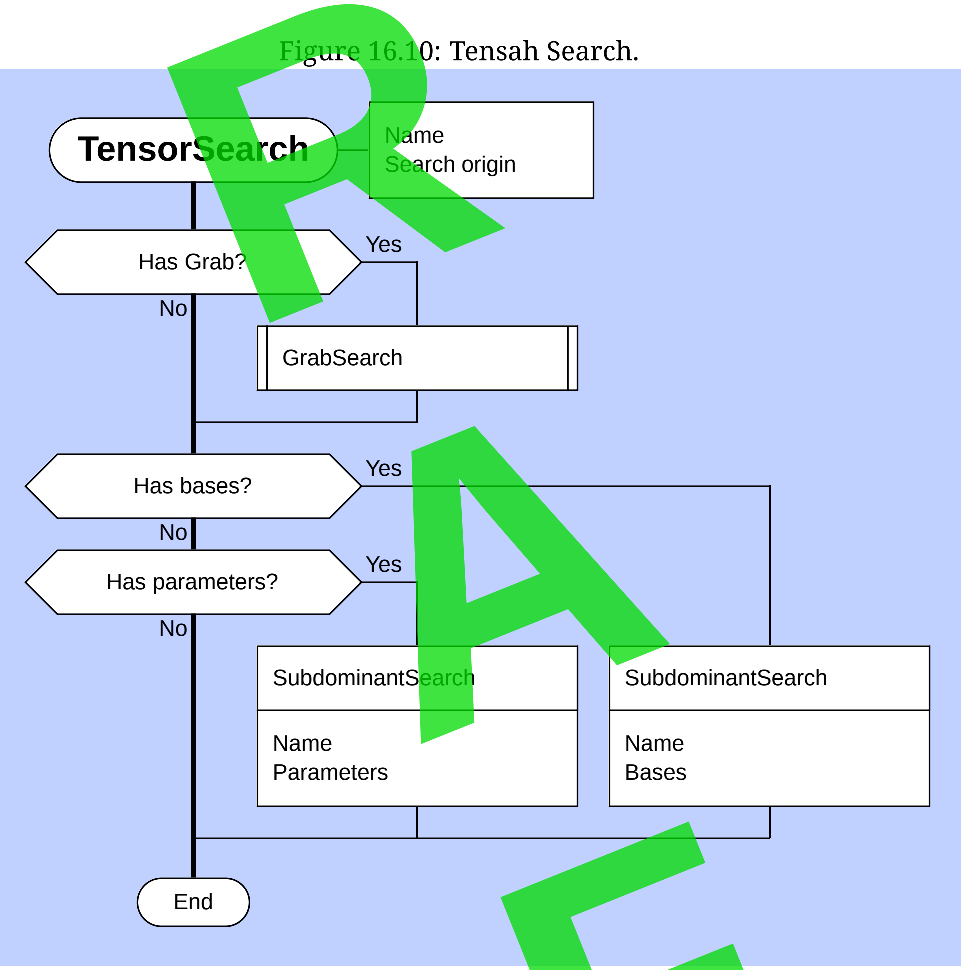
Figure 16.9: Let-name Match.



Let-names immediately become search candidates as they're introduced. Implicit let-names may be introduced in parameter lists, base type lists, and grabs. This facilitates more succint and safe code (debatably at the cost of less clear code) by allowing let-names to be directly created in the scope they will be used in.

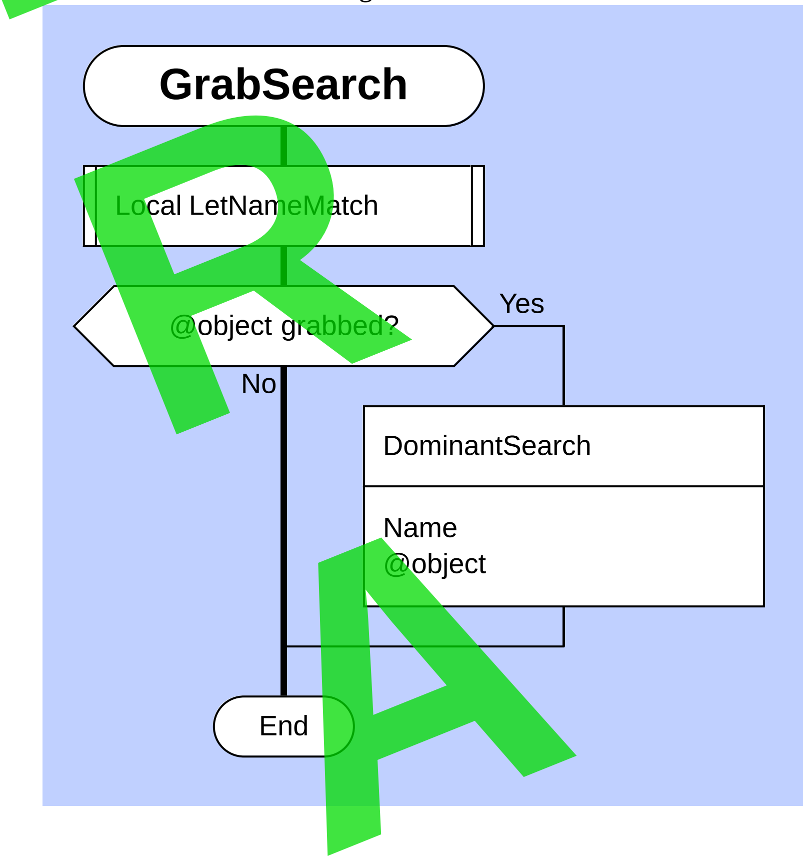
When the search name for is a dominant name, and candidate names, such as those that are grabbed, could also be in dominant form. Those candidate names are treated as potential prefixes of the search name. There is simply too much ambiguity if there are multiple candidate names that are all prefixes of the search name. If such search was allowed, there could be multiple overloads from multiple prefixes. Any rule trying to prioritize or prune candidates will be too complicated for any programmer to remember, let alone understand.

Figure 16.10: Tensah Search.



Tensah search is co-recursive-descent with subdominant search. If a Tensah has bases, then it searches the bases with subdominant search starting with at the bases. While in that branch, if nothing is found, it continues with subdominant search starting at the parameters. While in that branch, if nothing is found, it continues with subdominant search starting at the enclosure. If the enclosure is (eventually) an imperative or declarative body, then it could end up in Tensah search.

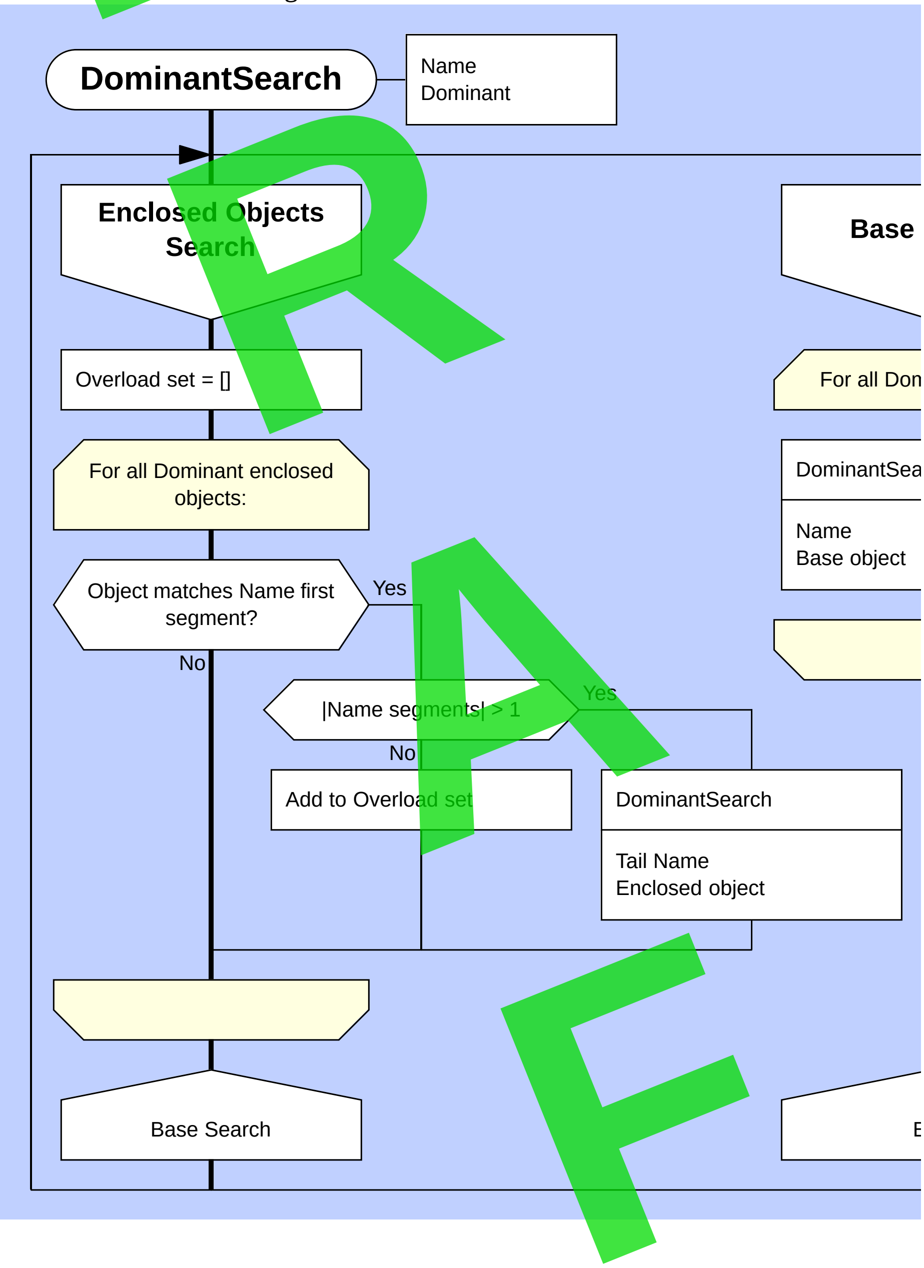
Figure 16.11: Grab Search.



Tensahs that grab the @object are somewhat like funcs enclosed by a type because grabbing @object means being able to search the enclosing object for a name. Even though @object appears in a grab supertonic directly after the parameters, the parameters are searched before the enclosing object, just like a regular type-enclosed func.

If a func grabs @object, and it is enclosed by a func that also grabs @object, then the @object that was grabbed are the same for both funcs. A func cannot grab @object when it is enclosed by a func that does not grab its @object implicity or explicitly.

Figure 16.12: Dominant Search.



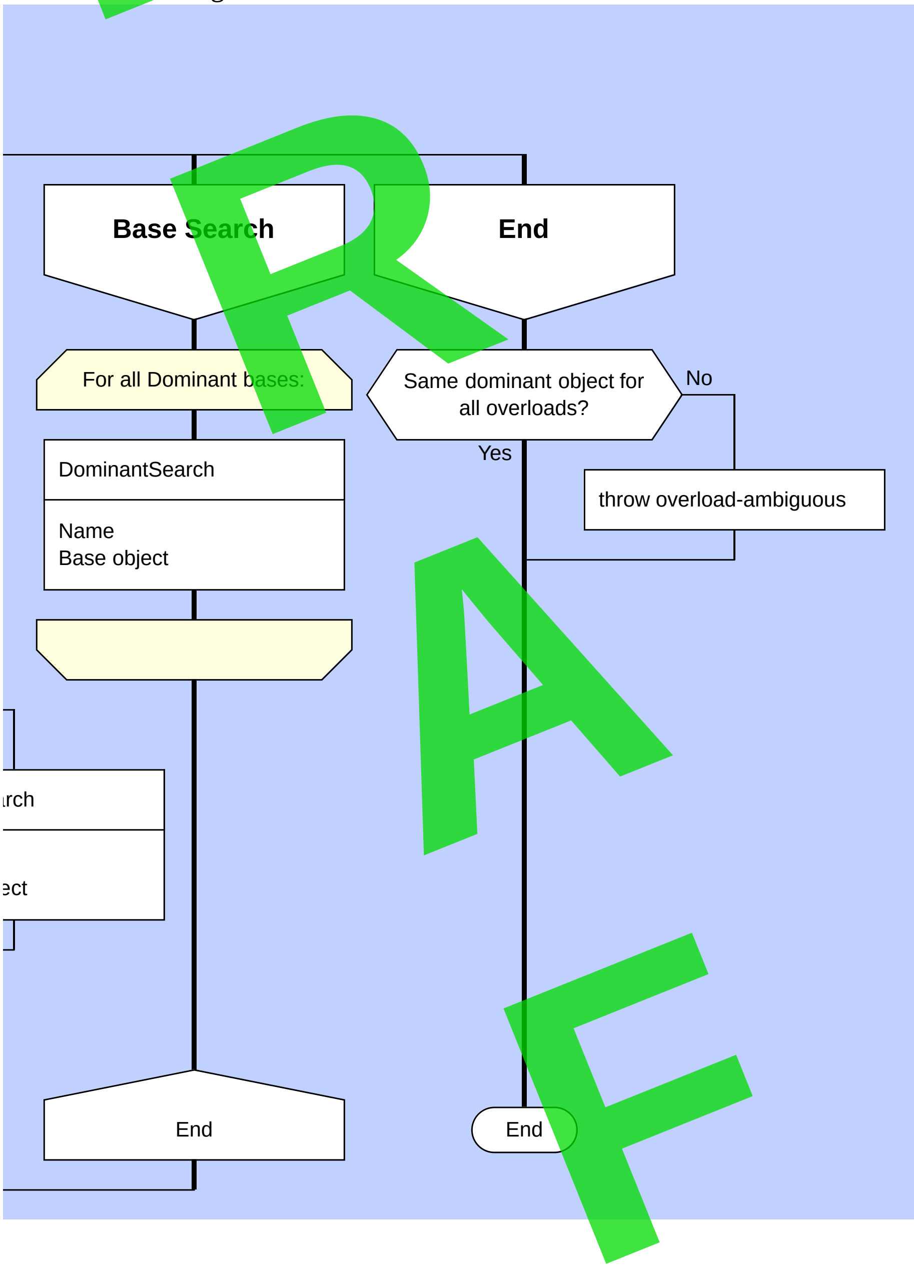
When searching for a dominant name with more than one segment, only the final segment is allowed to be overloaded.

Unlike C++, objects enclosed by base objects are searched, even if candidates have been found in the dominant object. At first, this sounds like a recipe for disaster, because, say, if a base object func and the dominant object func has the same arguments, then that would trip the overload-ambiguous condition. Recall that implicit conversions are also not allowed with overloads, and this would also trip the overload-ambiguous condition.

This is actually a desirable property, because shadowed funcs become an AM-error, with the exception of virtual funcs. Shadowed funcs are in the same category of error as implicit conversions: the @object. The ease-of-use is not worth the unergonomically hidden errors of implicit conversions.

To overcome shadowing conditions, explicitly convert the dominant object to the dominant type, or one of the base types, as documented in Conversion on page 67.

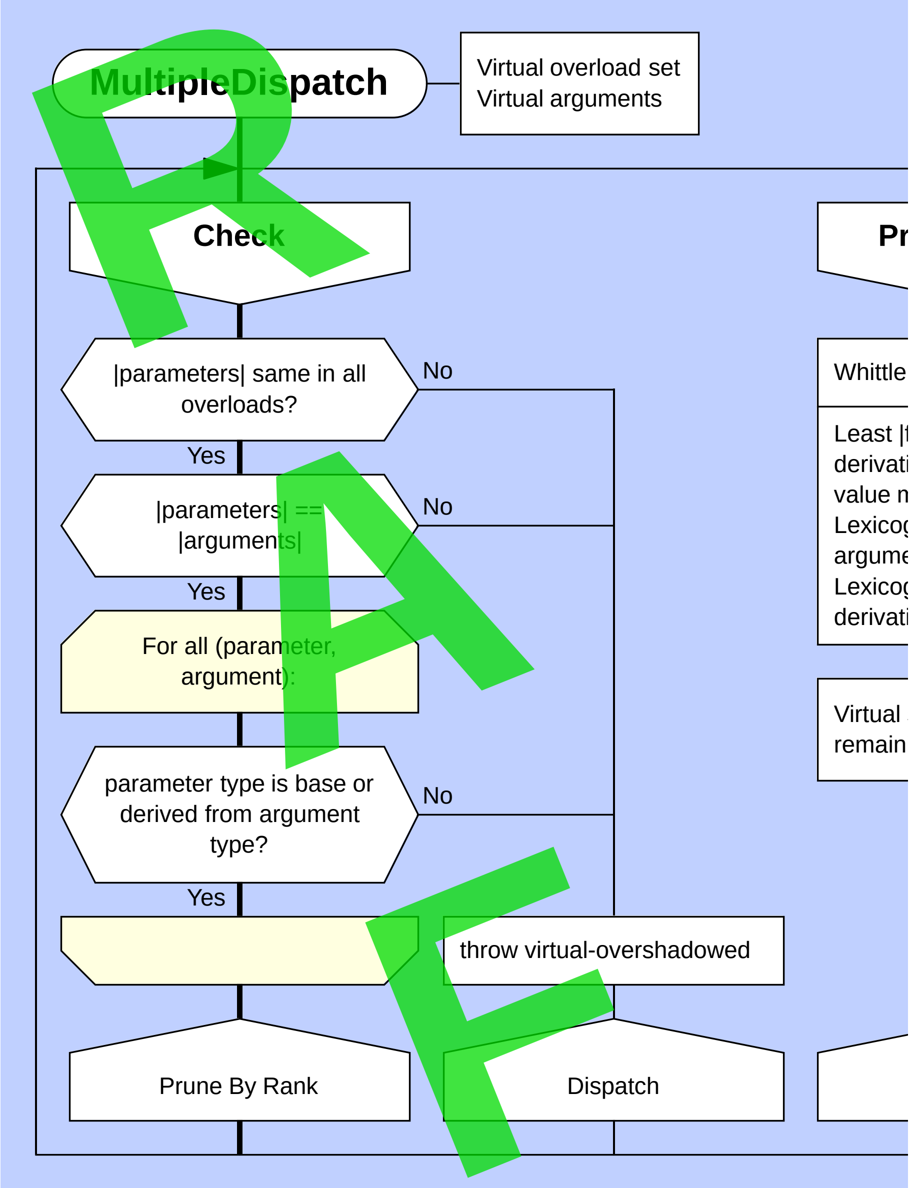
Figure 16.13: Dominant Search (continued).



Dominant search of base classes need not follow the construction-order or reverse-construction-order. The same principle is followed that finds all matching names so that shadowing funcs can be reported as errors.

The constraint that the enclosing objects of the overloaded found names be the same ensures that only the final segment of a dominant name is overloaded.

Figure 16.14: Multiple Dispatch.

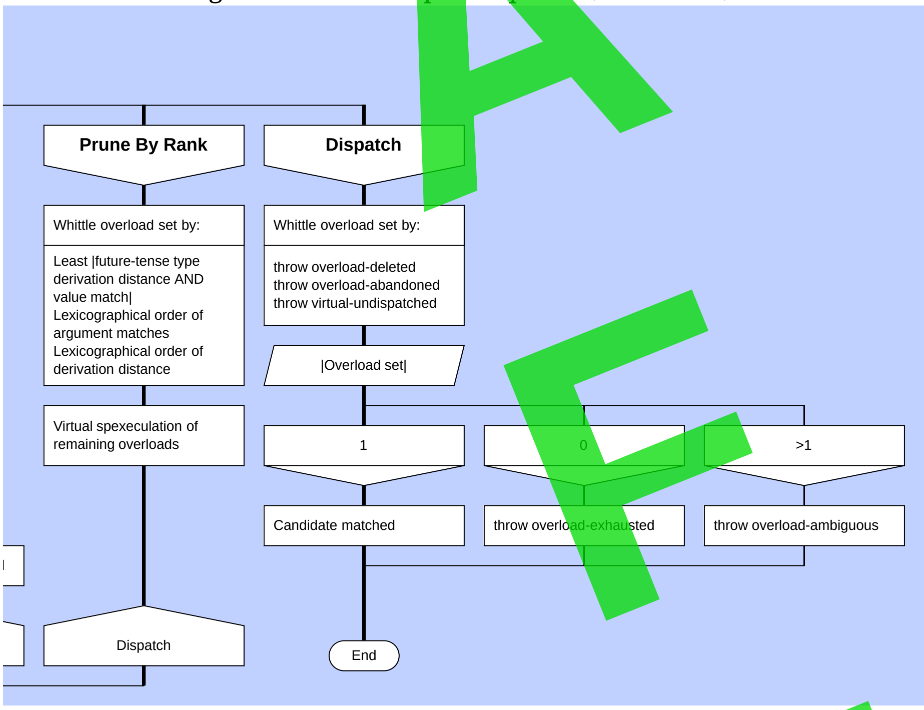

MultipleDispatch
Virtual overload set
Virtual arguments
Check
|parameters| same in all overloads?
No
Yes
|parameters| == |arguments|
No
Yes
For all (parameter, argument):
parameter type is base or derived from argument type?
No
Yes
throw virtual-overshadowed
Prune By Rank
Dispatch
Pr
Whittle
Least |
derivati
value n
Lexicog
argume
Lexicog
derivati
Virtual
remain

When an overload-set contains funcs that evaluates to different numbers of arguments, the normal pruning process will have already pruned the candidates with the best matching arguments to parameters. If the overload-set still contains multiple candidates, then normally this trips the overload-ambiguous condition. This changes when any of the parameters are virtual, and are the cause of the ambiguity.

When any parameter of any func in the remaining overload-set is virtual, then multiple dispatch pruning kicks in - in present-tense if types are known in advance. This covers the most common case of first/object-argument single-dispatch of most object-oriented languages, and induces the same, desirable, Tensah overload-shadowing AM-error behaviour. It is ergonomic, correct, and simple, to extend this behaviour to all funcs with at least one virtual parameter in any position.

The verification of equal number of parameters - of all overloads - and arguments takes into accounts pack-parameters and defaulted arguments.

Figure 16.15: Multiple Dispatch (continued).


Prune By Rank
Whittle overload set by:
Least |future-tense type derivation distance AND value match|
Lexicographical order of argument matches
Lexicographical order of derivation distance
Virtual spexeculation of remaining overloads
Dispatch
Dispatch
Whittle overload set by:
throw overload-deleted
throw overload-abandoned
throw virtual-undispatched
|Overload set|
1
0
>1
Candidate matched
throw overload-exhausted
throw overload-ambiguous
End

The normal name search and pruning rules forbidding implicit-casts implies that virtual parameters of overloaded funcs will always be in the same position(s).

Consider just the derivation-distance of the types of virtual parameters of the following:

$$f(3,1) \tag{16.1}$$

$$f(2,4) \tag{16.2}$$

$$f(1,2,1) \tag{16.3}$$

$$f(2,1) \tag{16.4}$$

Between 16.1 and 16.2, it might seem intuitive to prioritize lexicographical ranking of virtual overloads. After all, it's easy: distance-2 is obviously a better match than distance-3; the distance-2 argument comes before the distance-4 argument; pruning for normal funcs does it that way. But does that mean 16.1 is a more closer match than 16.2?

In normal funcs, all parameter types are known exactly in present-tense, due to the elimination of implicit-casting. So it makes sense for a lexicographical scheme to be used for ranking default-arguments and pack-parameters. Virtual-ness changes the semantics of matching because the type is no longer known exactly in present-tense. Inexact matches (ie, derivation-distance > 1) are acceptable as long as there exists an overload with a parameter type that the argument can be downcasted, which means that, as 16.1 illustrates, there could exist better matches further down the parameter list.

Consider a game physics engine collision simulation. Imagine that 16.2 is a func that handles collisions for some general case; that 16.1 is a func that handles collisions between some object, and an object that is destructible. Using lexicographical pruning, the general case would win. If the least derivation-distance argument wins, then we would choose the more specific func to handle collision with destructible object.

In a real world scenario - at least on classical scales - interactions between objects are commutative. A ball colliding with a wall of bricks (a common demonstration of physics simulations) should behave the same way as a wall of bricks colliding with a ball. Hence, TONAL uses the least derivation-distance to determine the best multiple-dispatch. This can help avoid requiring different permutations of arguments just to force some kind of commutative behaviour.

The least derivation-distance criteria, in a multiple-dispatch context, implies that virtual parameters are not privileged by any order. A programmer has the option to write a dispatching-func as a subdominant, so it wouldn't make sense prioritize any virtual argument. The implicit @object parameter does not subsume the least derivation-distance criteria so that object-enclosed funcs can be used in subdominant syntax and behave exactly as other subdominant dispatching-funcs.

Common uses of multiple-dispatch will not need the least derivation-distance because most problems involving virtual objects are defined in terms of the most-derived types anyway. The least derivation-distance matching is there to provide a memorable default behaviour for the in-between cases.

Between 16.3 on the preceding page and 16.4 on the facing page, both funcs have the least derivation-distance of 1. In case of a tie, lexicographical ranking finally comes into play. 16.4 on the preceding page wins the match because it has one fewer argument with a least derivation-distance of 1.

TONAL does not let a match failure in future-tense to fail silently. Multiple-dispatch is a form of pattern-matching; TONAL follows other languages in making pattern-matching complete, so that no options are forgotten.

Part IV

Structure

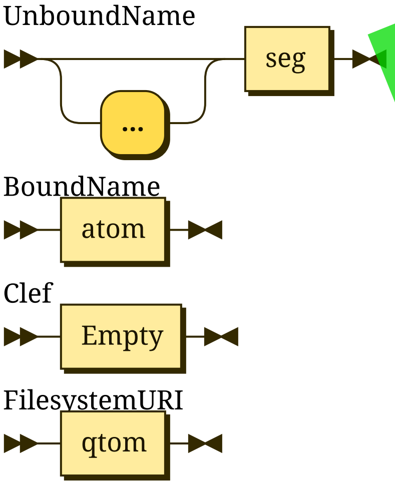
Parentheses are omitted for clarity where the grammar rule's name is prefixed by one of the scale degrees. For submediant scale degree tonics, assume an extra pair of parenthesis.

In some syntactic contexts, not all tonics of a scale degree are allowed. In order to simplify the grammar diagrams, general scale degree names are used, but with a special syntax to denote subsets and set differences.

The organization of ideas creates the structure of a program. The structure of the program defines how names are searched.

In all subtonics, the first subject must not be an atom that is any of the keyverbs. Even though the notion of the subtonic scale degree prevents syntactic ambiguity - the AM will never treat the first subject of a subtonic as a keyverb - nevertheless it will be onerous for tool writers if it were allowed, who would much prefer to be able to do a quick, context-ignorable, scan for keyverbs.

Figure 16.16: Common elements railroad diagrams.



a

a

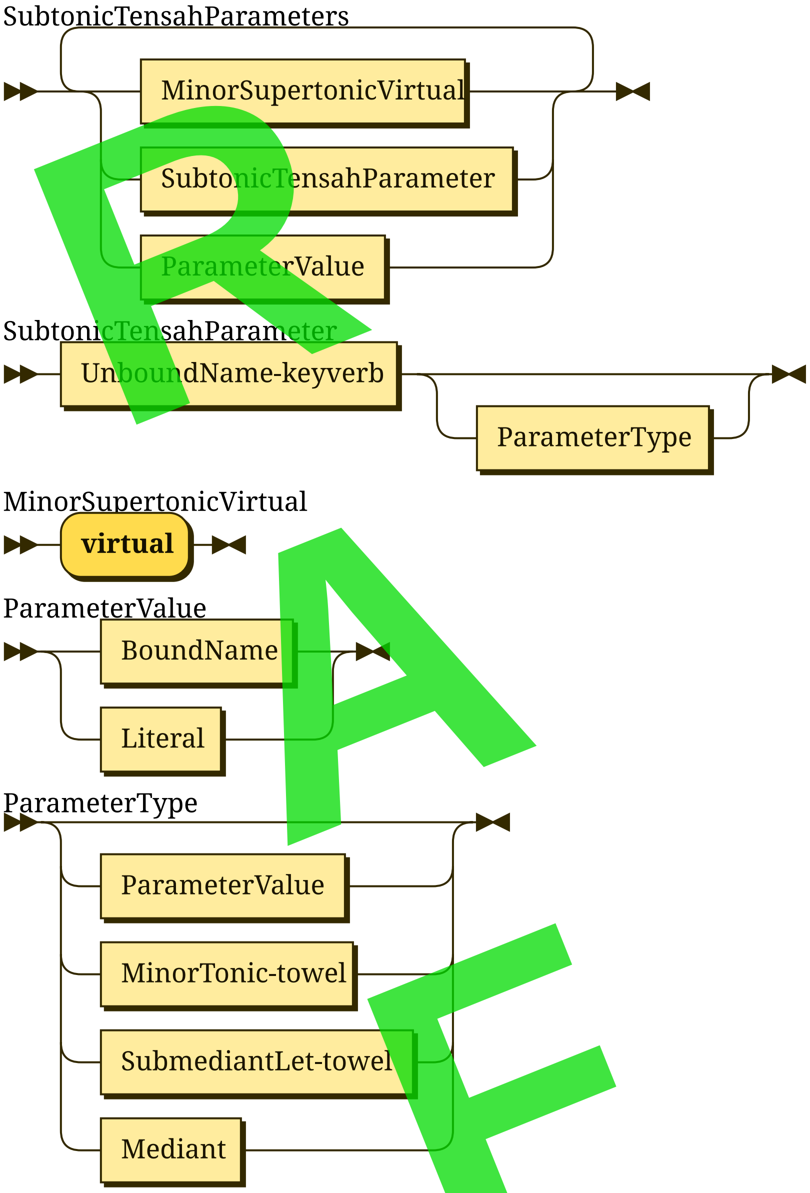
A new name is always introduced in some implied context, providing the stem, so only the leaf of an atom is permitted. The name is not already bound to an object, until a value or tensah is constructed for it, or a subject to a tonic is an argument to a tensah parameter. Parameter names may be preceded by an ellipsis to signify variadic arguments.

A name that's already bound to an object can be specified with as long a branch as necessary to find it.

A clef marks the beginning of a section in a supertonic where an unlimited number of tonics is supplied as a subject.

Included files are referenced using the standard URI for ultimate portability. It also signals, from the start, of the intent of the language to be able to be stored across networks, instead of added as an afterthought. Such networked storage may include package managers.

Figure 16.17: Common Tensah elements railroad diagrams.



a

[a]

Named parameters must not be one of the keyverbs because of the subtonic keyverb rule.

Named parameters without a type are fully generic parameters, accepting of values of any type and pre-past-tense.

Unnamed parameters, like parameter type values, serve as present-tense overload pruning, so must be provided as present-tense values. Such values cannot be provided programmatically - ie tonic - because there would otherwise be no way of differentiating between the syntax of a named parameter, and a tonic providing an unnamed parameter value.

Parameter types provided programmatically does not allow TOWEL transfers to avoid confusing, non-obvious, state changes in the meaning of a program.

Defaulted parameters, as specified by a let-submediant, also serve to specify the type of the parameter.

Type parameters must be fulfilled by arguments that evaluate in the present-tense.

Virtual parameters for types can simplify programs by their relaxed overload-pruning rules. In regular overload-pruning, multiple matches for an argument trips the ambiguity condition; requiring explicit conversions to disambiguate. For types that have dispatch semantics, virtual parameters can be used to trigger the dispatch pruning rules instead which elide explicit conversions.

16.1 Submediant Forms

The verb position is the command to be evaluated. Certain supertonics in the verb position are like anonymous “verbs” to be evaluated.

Submediants are semantically equivalent to a local func that is immediately evaluated. The local, immediate, evaluation - and func-ness - places simplifications and restrictions on name-search and control flow for ergonomic and correctness reasons.

Submediants don’t live beyond their immediate evaluation, so their TOWEL cannot be transferred. They cannot be given a let-name and reused later. Let-names from the encloser can therefore be used without needing to be grabbed since there is no danger of dangling accesses that is usually associated with closures.

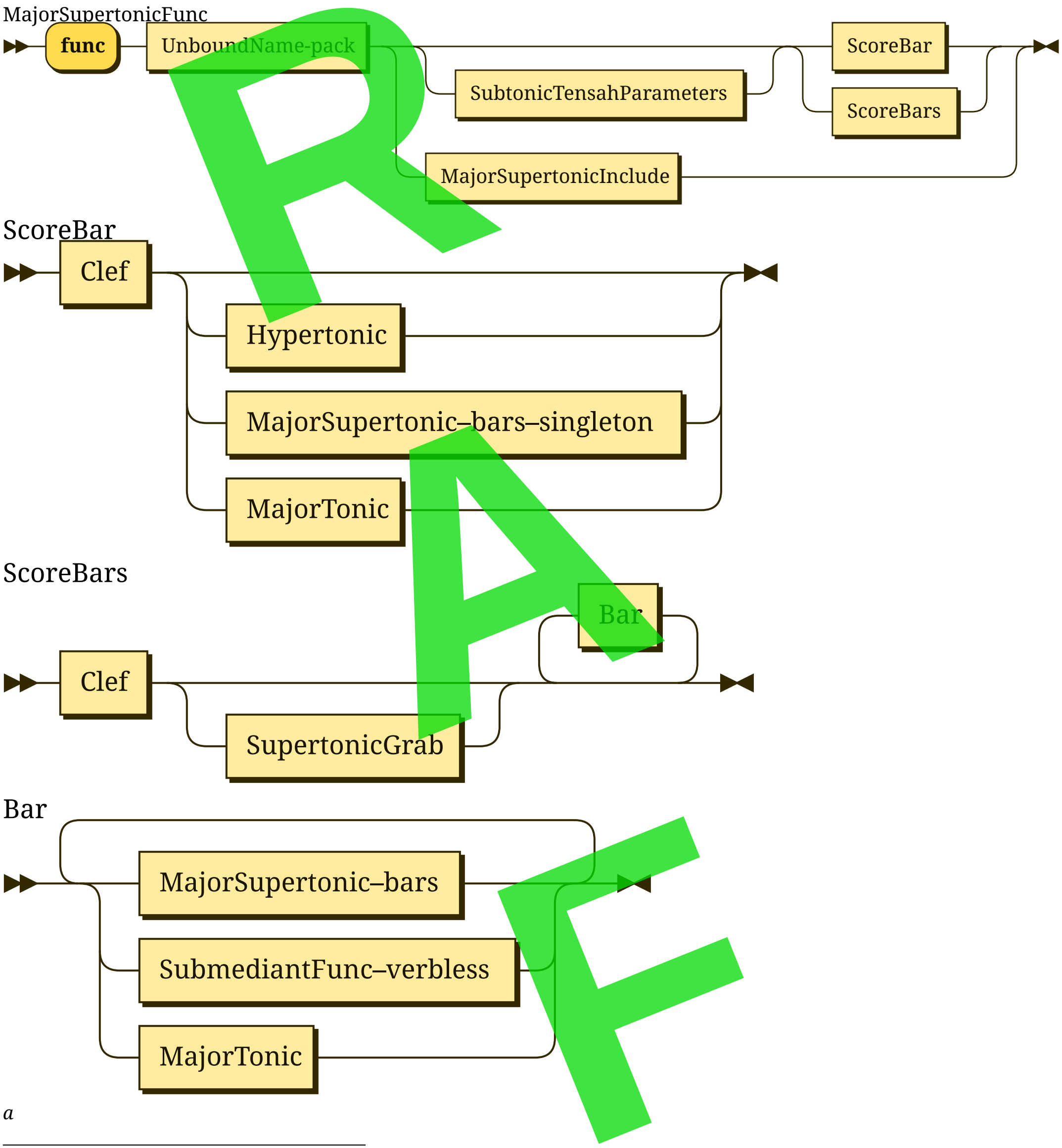
Resuming supertonics are not allowed in submediants because they don’t live beyond their immediate evaluation, and so don’t have any state than can be resumed. It also reduces misreading the encloser as being resumable.

Submediants can, and must, explicitly evaluate to some value. Like ternary operators, or expression-oriented languages in general, submediants are used to produce a value without the distraction of control-flow structures. Jumps must stay within the submediant, unless it is a tripped condition that is not trapped by the submediant.

Verbless submediants are a further simplification of the $func$-submediant. The func verb and the clef can be omitted. $eval$-supertonic belongs the encloser. The verbless submediant itself does **not** evaluate to any value, just like any regular control-flow supertonic. The verbless submediant is used to introduce a new TOWEL environment, like braces are used in C++ to create a new scope in which objects created within are cleaned up at the end of the scopes.

17 Supertonic Func

Figure 17.1: Major Supertonic Func railroad diagrams.



[a]

[a]

TONAL treats all funcs as essentially the same. In languages like C++, free/static member functions, non-static member functions, and capturing lambdas are different types. It makes higher-order functions - those that take functions as parameters; sometimes called operators in mathematics - more complex to write generically and efficiently.

By treating funcs the same, higher-order functions automatically become generic without having to account for differences between alternative machine representations for subroutines. Funcs become first-class entities via this semantic, instead of relegating to some type-erasing function wrapper type. It is still necessary to write a wrapper type for funcs for the purposes of dynamic storage.

With machine representation accounted-for generically, only the regular TOWEL concerns for objects separate object-dominant funcs, grabbing funcs, and resumable funcs, from other funcs.

Resumable funcs cannot transfer TOWELs by default. Only when parameters and grabs have TOWEL can a resumable func's TOWEL be transferred; such as to a task scheduler. Resumable funcs that do not have TOWEL transferred may have their state-memory allocation completely elided in present-tense.

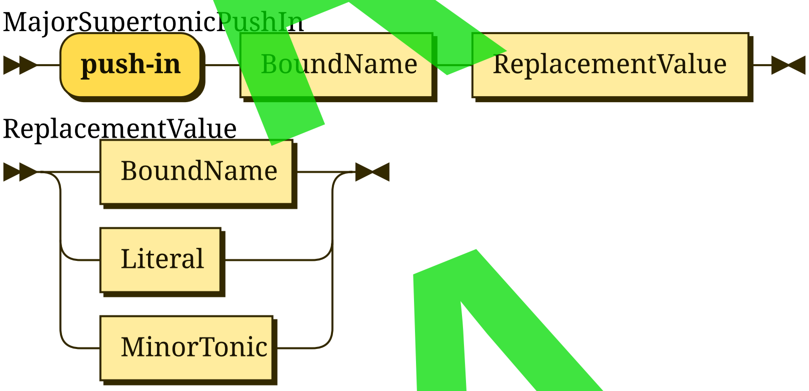
A subset of major-supertonics can appear as the only bar in a func. For exampe, funcs cannot contain just a grab-supertonic because there is no reason to grab let-names and then do nothing. A single give-supertonic may as well just be an eval-supertonic instead. A goto-supertonic wouldn't do anything on its own. Any of the TOWEL supertonics wouldn't do anything on their own either. On the other hand, hypertonics must be the only tonics in a func if they appear in a func to avoid confusing with mixing hypertonics and other tonics.

Type-enclosed funcs cannot grab, other than the implicitly grabbed dominant-object. If a func must grab, it must be the first supertonic. Unlike in languages like C++ that has the lambda capture specification at the very beginning, TONAL takes the view that the parameters of a func are visually more important than grabs. Grabs are implementation details of a specific func. TOWEL rules prevent grabs from dangling binds, so it's not as important as in C++ to be able to see reference captures.

The verbless func-submediant can only appear as a bar.

17.1 Major Supertonic Push-In

Figure 17.2: Major Supertonic Push-In railroad diagrams.


MajorSupertonicPushIn

ReplacementValue

[a]

[a]

The let-name that the replacement value is being pushed-into must have its object's TOWEL, otherwise it trips an unatonable condition.

This supertonic triggers copy-construction if the replacement value is not pulled-in. The copy-construction is evaluated with a valid @object.

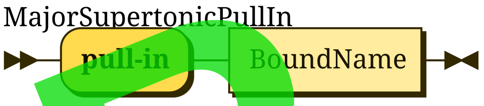
If the replacement value is pulled-in, then copying may be elided, and it may be constructed in-place.

If the let-name's object has not been read-from between its construction and the push-in, the original construction and intervening writes may be elided. Special note must be taken of writes that actually reads the object first, and thus cannot be elided.

If the replacement value is of a different type to the let-name, but the underlying representation of the replacement value is the same, or the replacement value's type has relaying, then it triggers the relay.

17.2 Major Supertonic Pull-In

Figure 17.3: Major Supertonic Pull-In railroad diagram.



a

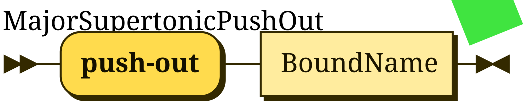
[a]

If the let-name has the object's TOWEL, then the pull-in has no effect. If the let-name doesn't have the object's TOWEL, then copy-construction is triggered.

One possible optimization that is enabled by this is lazy construction. If the pulled-in object hasn't been read from, then the object may be constructed at the pull-in site, rather than where the func is being evaluated. The AM can track the TOWEL cascading through the chain of func evaluations, and elide all writes until the first read. This is helpful in situations like storing an object in some data-structure. The data-structure may have some intermediate actions to perform - such as allocating memory - before storing the object in its ultimate destination. The AM can procrastinate constructing the object until it is stored in its ultimate destination, instead of constructing and then transferring TOWEL.

17.3 Major Supertonic Push-Out

Figure 17.4: Major Supertonic Push-Out railroad diagram.



a

[a]

Pushing-out an object that is not pulled-in, or already pushed-out, has no effect.

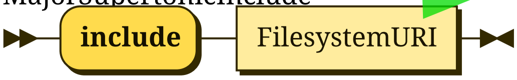
The object being pushed out can still be read after. Each read prolongs the life of the object by postponing eager destruction.

If the TOWEL roundtrip was activated as a result of the push-out, it is also deactivated after the completion of the roundtrip. The TOWEL roundtrip may be reactivated with another pull-in/push-out transfer.

17.4 Supertonic Include

Figure 17.5: Major Supertonic Include railroad diagram.

MajorSupertonicInclude



[a]

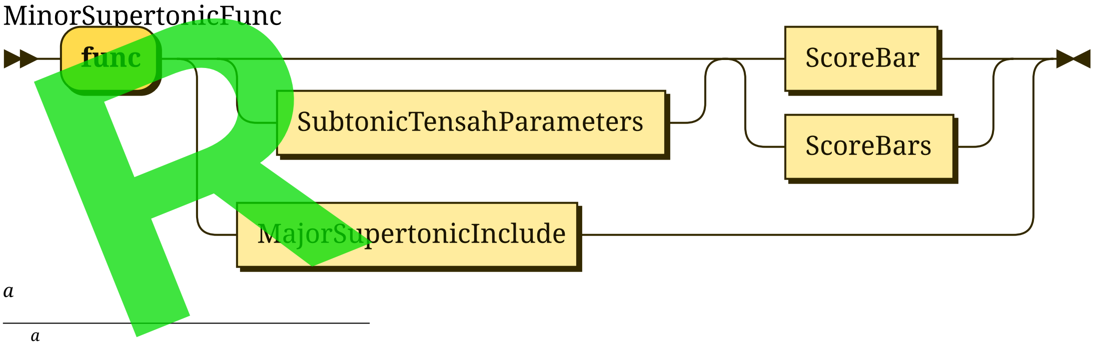
[a]

The major tonic is only used for importing symbols from other source files. No evaluation to a value occurs.

The symbols that are imported are idempotent. All includes of the same file anywhere in a program is only imported once. Included files are tracked and compared by their canonical absolute paths, but without following links, for portability, ease-of-implementation, and performance reasons.

17.5 Minor Form

Figure 17.6: Minor Supertonic Func railroad diagram.



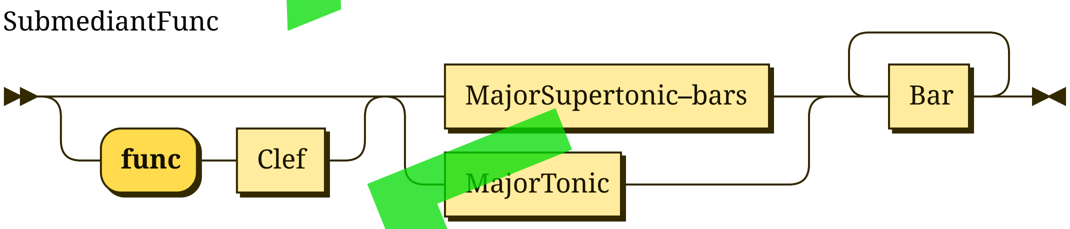
[a]

[a]

Creates a func on the fly, instead of as an enclosed object of a type. It is the equivalent to lambda functions in languages like C++.

17.6 Submediant Form

Figure 17.7: Submediant Func railroad diagram.



[a]

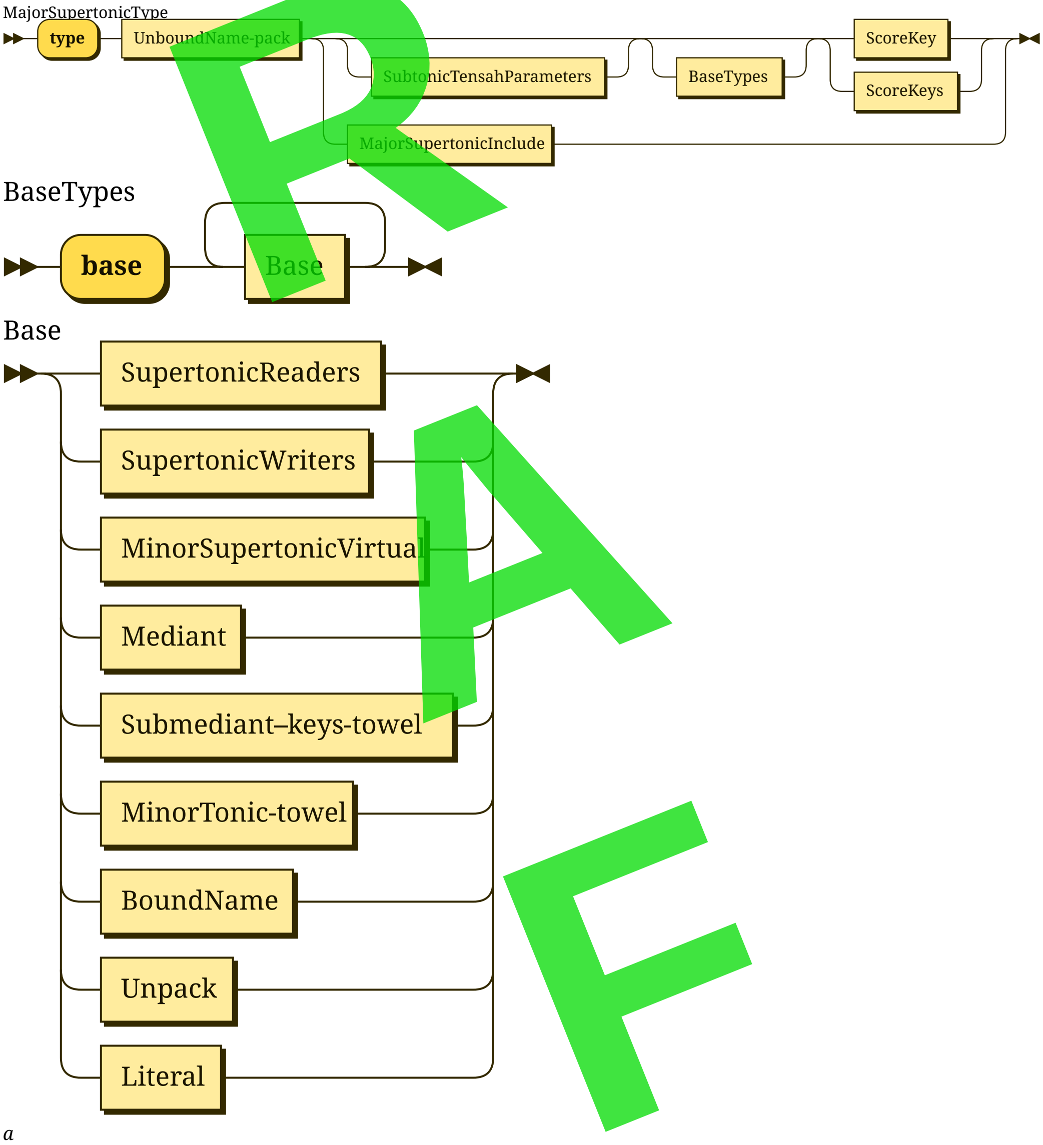
[a]

Verbless submediant funcs do not evaluate to a value and so can only be used like a major tonic.

All names are grabbed by default, since it is intended to be the same as a compound-statement in languages like C++. This is safe, since the func is not transferrable, and not even bound as a let-name. Submediant funcs are really just syntatic sugar, but classified as a func for the purposes of categorization in the scale-degree scheme.

18 Supertonic Type

Figure 18.1: Major Supertonic Type railroad diagrams.



a

[a]

The parameters and base types subtonics and the Keys can come from an include file instead of having to be written inline. This kind of project organization can help with shorter processing times and IDEs.

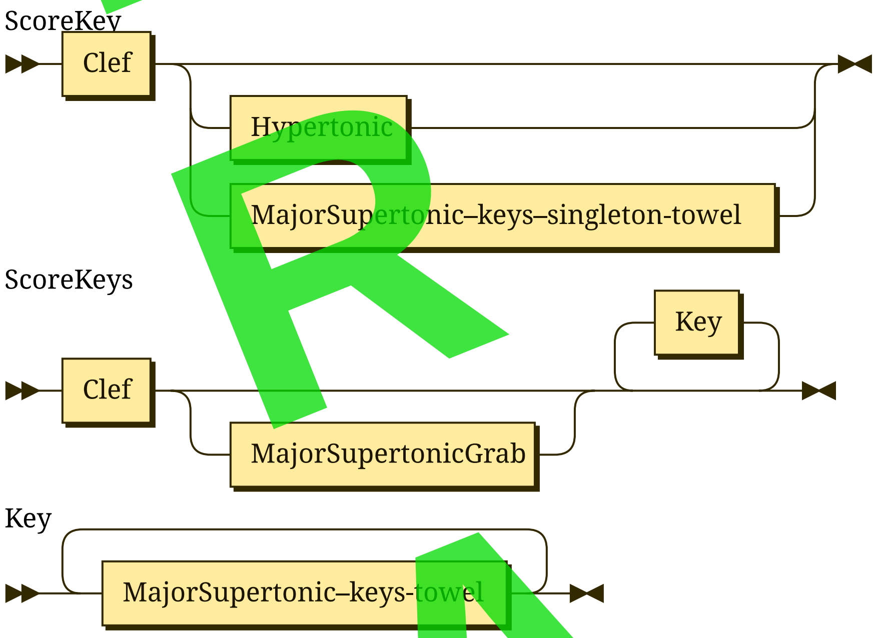
The virtual access of a base has a different meaning in the base subtonic compared to its use in the Keys. In this context, it refers to virtual inheritance, and is the way to solve the diamond inheritance problem.

TONAL even permits derivation from literals. It is the logical extension of inheritance of types where the value of the base object is known in present-tense. Such base objects are not modifiable, even in present-tense, to avoid over-complicating the language with changing type bases in present-tense.

Types that derive from an archetype must have a size, and may be preserved into future-tense, unlike archetypes, but the archetype base object is never preserved, so its size remains 0. Archetype values must always be in present-tense, but types derived from archetypes can straddle between future-tense and present-tense, therefore archetype base objects does not require a present-tense value. The AM keeps track of this.

Types derived from mediants are themselves mediants, and subject to the same limitations as mediants.

Figure 18.2: Major Supertonic Type railroad diagrams (continued).


ScoreKey
Clef
Hypertonic
MajorSupertonic–keys–singleton-towel
ScoreKeys
Clef
Key
MajorSupertonicGrab
Key
MajorSupertonic–keys-towel

[a]

Only *let*-, *func*-, *type*-, and access control major supertonics can be a key.

A base-object may have a func that is a good default interface that the derived-object would want to advertize as its own, such as a hook from the base-object. As a shorter alternative to writing a wrapper in the derived-type that simply calls the base-object, the *let*-supertonic can be used to hoist the func. This does not result in increased size of the type.

18.0.1 Hooks

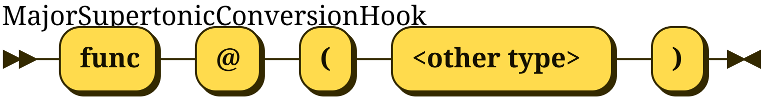
Hooks are not resumable, so must not attempt to evaluate the *give*- and *wait*-supertonics. Hooks cannot grab let-names.

Only construction hooks can be treated as a readonly let-name for an object of func type. Such objects behave as a factory-func for that type. This is to aid the simplicity of generic code that takes factory-funcs.

[a]

Figure 18.3: Conversion hook.

MajorSupertonicConversionHook



[a]

Conversion hooks are triggered when an object of one type is constructed from an existing object of another type, and there are no appropriate construction hooks that does the same.

Conversion hooks must accept just one argument, which is the type of the desired type. Any other number or type of argument trips an unatonable condition.

No other let-name is allowed the @ character, but the conversion hook can be overloaded for multiple desired types. There can be no generic catch-all conversion hook, because conversions can cause issues, and so must be carefully considered.

Conversion hooks are typically not necessary for converting an object to one of its base objects. For overload-pruning, the explicit construction of a base-type from an object is merely for disambiguation and does not involve the creation of an actual object, so no conversion needs to be done.

Conversion to a base object may prevent some optimizations in the memory layout of the underlying bits. Some AM implementations may be able to flatten out an object's hierarchical memory to eliminate padding, but if conversion to base type is forced, then that prevents it being flattened, as it must maintain its own separate internal layout to allow for extraction as its own object.

Conversions to archetypes is the mechanism by which the AM differentiates between present-tense and future-tense values. All archetypes must have present-tense values, but types derived from archetypes can

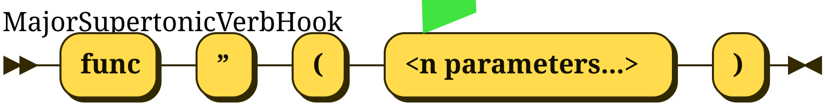
[a]

be future-tense. To use an object of a archetype-derived type in present-tense, it must be capable of being converted into one of the archetypes. A conversion hook should trip a certain atonable condition to signify to the AM that an object is future-tense. Types with multiple enclosed objects could convert to a List of present-tense values.

Unpacking an object is done by converting an object to a List, and unpacking that List into let-names.

Types that can be constructed from an archetype that it was converted to may be used by the AM to memo-ize computations in order to speed up a present-tense processing session, perhaps even across time-separated sessions such as incremental builds.

Figure 18.4: Verb hook.



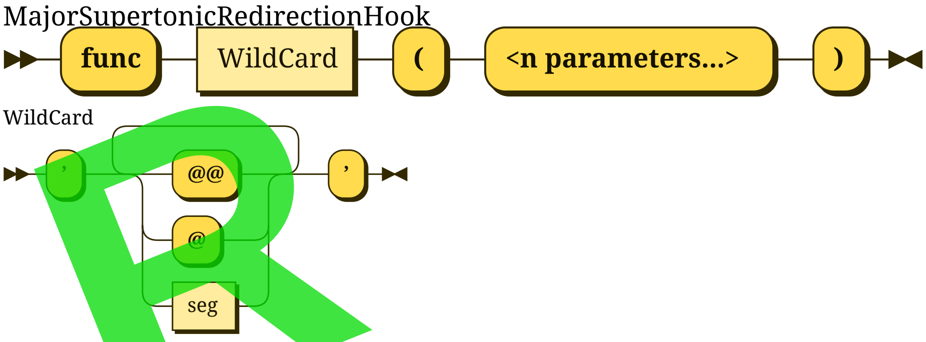
[a]

[a]

An object of a type with a verb hook can be used in the verb position of a tonic. This is equivalent to function-objects in languages like C++.

Lambda funcs, ie funcs with grabs, are great for quickly encapsulating localized functionality, but if they get too complex, they should become their own type. This enables simplifications, such as splitting a func into overloads, rather than crammed into one func.

Figure 18.5: Redirection hook.



a

a

Branch redirection would open up a can of worms, as it would allow the possibility of bypassing TONAL’s fundamental atom semantic, where each node in the atom refers to an enclosed object in the prior node. Programmers would work under such assumptions, and breaking such assumptions would be easily overlooked. The wildcard can thus only match a name that is a single node.

The characters (,), “, and ′, are not allowed in atoms, so it would be impossible for the wildcard to contain those characters. Specifying those characters in the wildcard trips an unatonable condition. The strings @@ and @ can be used in the wildcard for zero-or-more and single-character matches, respectively. They are equivalent to the ∗ and ? wildcard characters in languages like Bash shell. TONAL cannot use those characters for wildcards as they are valid in names.

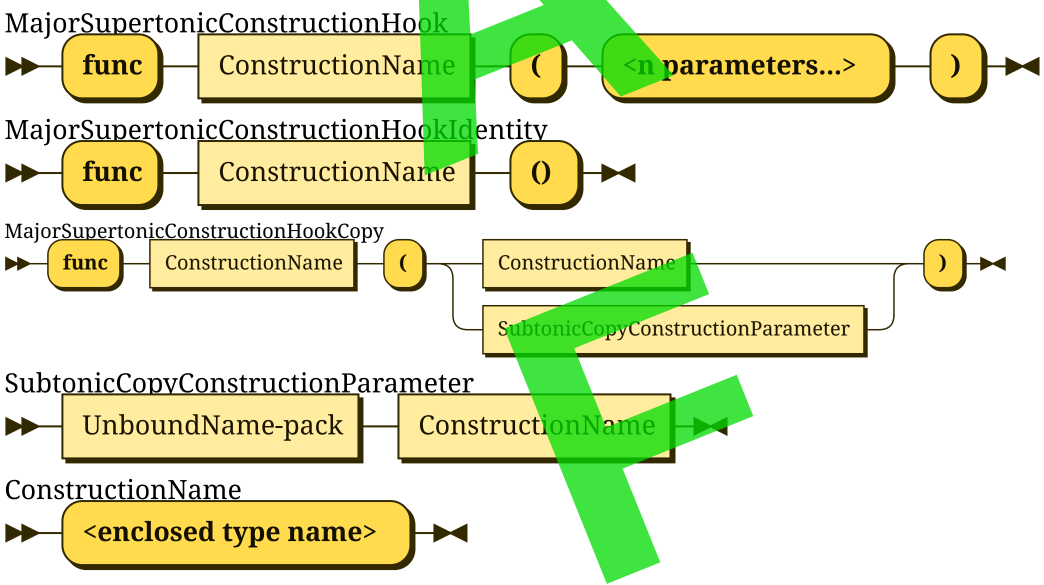
The Tensah enclosed @jective @name contains the result of the wildcard match in total. There are no capturing matches as there are in regex. Present-tense string manipulation functions should be used to interpret the matched name if such trickiness is needed. The @name qtom will never be empty, just like how no atom-node can be zero characters long. If an object is used in the verb position, and there is no verb-hook, then that trips an unatonable condition, rather than implicitly, and confusingly, calling a redirection-hook.

The parameters are entirely up to the hook implementation. There are no special AM parameters for redirection hooks.

Hooks, especially redirection-hooks, are never redirected. This prevents almost untraceable infinite recursion, and discourages overly complicated redirection schemes.

Redirection hooks cannot be inherited. This solves the issue of redirection if multiple types with redirection hooks are inherited from. This simplifies the implementation of wrapper types (the main use case for redirection hooks), especially in the case of wrappers of wrappers. The programmer has complete control of how redirections are forwarded, instead of fighting TONAL rules.

Figure 18.6: Construction hooks.


MajorSupertonicConstructionHook
func
ConstructionName
(
<n parameters...>
)
MajorSupertonicConstructionHookIdentity
func
ConstructionName
()
MajorSupertonicConstructionHookCopy
func
ConstructionName
(
ConstructionName
SubtonicCopyConstructionParameter
)
SubtonicCopyConstructionParameter
UnboundName-pack
ConstructionName
ConstructionName
<enclosed type name>

[a]

All types have an identity, copy, and default construction hook. They are initially generated by the AM. If the programmer creates funcs with the exact signature as those hooks, then they replace the generated hooks.

[a]

Sometimes programmers just wants to use the AM-generated versions of those hooks, but doing something extra after construction. They don't want to have to re-implement the natural versions of those constructions. In the identity, copy, and default construction hook, evaluating the @type @-jective with their respective arguments will use the respective AM-generated hook.

Construction hooks can delegate to other construction hooks, but not recursively, whether direct or indirect.

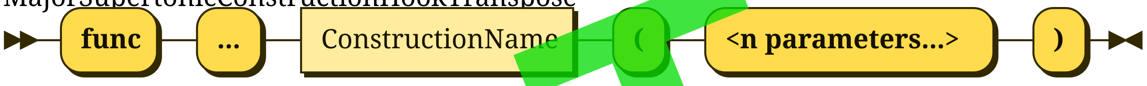
TONAL does not bifurcate construction and assignment in the way languages like C++ do. There does need to be accommodation for potential optimizations that takes advantages of the difference between constructing an entirely new object, as opposed to replacing an existing object. In the copy-construction hook, the @object is invalid in present-tense, throughout the entire construction, when constructing a new object. The @object is valid in present-tense if replacing an existing object. Even if the @object is invalid in copy-construction, its enclosed objects can be touched once they are constructed.

The @object can only be invalid in constructions of new objects. In every other enclosed func, the @object is always valid in present-tense and beyond.

Types that are solely-derived from one archetype have AM-generated identity construction hooks that creates a future-tense value of the derived type. The type must provide its own default-construction hook that takes a present-tense value to construct its archetype base-object in order to construct a present-tense value.

Figure 18.7: Transpose construction hook.

MajorSupertonicConstructionHookTranspose



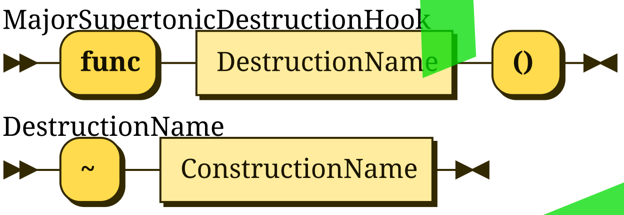
a

[a]

There sometimes can be performance gains for objects stored in arrays if there is an array for each enclosed object, instead of one array for whole objects. This is known as the structure-of-arrays, as compared to arrays-of-structures. Some benefits include tighter packing of data. Some access patterns, such as the same enclosed objects of separate objects are compared in some way, is better if those members are all in their own array. Another use is the architecture know as Entity-Component-System. An Entity could be a logical object, with its enclosed objects as Components spread out over arrays.

The transpose-construction happens in the same order as ordinary construction, but the AM pauses the construction after each enclosed object. The new constructed enclosed object is then copied into the rest of the array. In special cases, this could be a highly optimized byte-wise copy, instead of copy-construction of the enclosed object.

Figure 18.8: Destruction hook.



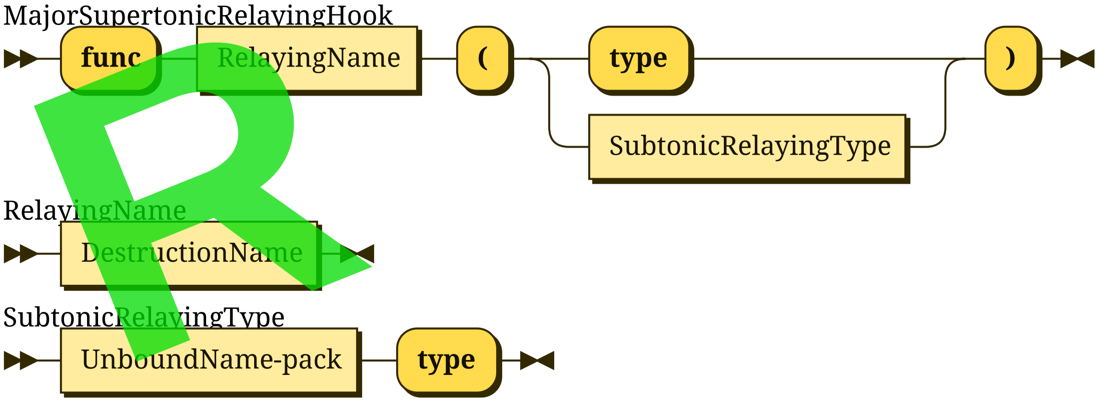
a

a

All types have a destruction hook. It is intially generated by the AM, but a programmer can replace it. The same destruction hook applies for objects created using the transpose-construction hook with the same-enclosed-object-at-the-same-time semantic.

Unlike in languages like C++, objects cannot be viewed after a destruction hook runs - such as the object being pushed-in to another func - which means destruction hooks don't have to clear the values of objects, saving some performance. Only objects with sensitive data, or some book-keeping cleanup func, need extra attention in destruction.

Figure 18.9: Relaying hook.



[a]

The func-name syntax is the same as destruction because relay is semantically like a destruction of the old object and replaced with a new object of a different type.

For relay type-pairs that are derived from one, and the same, base-type without additional enclosed objects and/or custom construction or destruction hooks, in the derived-type - called a persona-pair, the AM generates a relaying hook, which can then be replaced by a custom hook with the same signature.

For relay type-pairs that aren't a persona-pair, but have the same type, order, and alignments of enclosed objects, they are also treated as a pseudo-persona-pair. The AM does not generate a hook, since the type-pair's layout could be coincidental.

For relay type-pairs that aren't persona-pairs and pseudo-persona-pairs, but are of the same size - called a silhoutte-pair, generally should not be relayed. Some machine specific operations may require treating an object of one type as a sequence of bits of some other type. Relay hooks for silhouette-pairs is tantamount to low-level type-casting that you would find in languages like C.

[a]

Unlike C, silhouette-pairs are TONAL's way of short-circuiting the process of formalizing machine semantics into its type-system, rather than simply of way of working around the type-system. As is the TONAL style, it is much prefered to capture a machine-semantic into a TONAL concept as soon as possible, and then exploit it to full power of TONAL to encode related behaviours as part of a type; as opposed to trying to do everything in the AM-plementation language.

Custom relaying hooks can only touch the existing object before the relayed object. As soon as the custom hook touches the relayed-object, the expiring-object is considered destroyed, and touching the expiring-object will trip an unatonable condition.

For custom hooks for persona-pairs and pseudo-persona-pairs, in general, no data movement is necessary. Such a custom hook is useful simply for asserting that the relaying of the expiring-type to the relayed-type does not break invariants. Once the relayed-object is touched, or the custom hook has finished evaluation without touching the relayed object, then the AM considers the relaying to be complete.

Only custom hooks for silhouette-pairs are allowed to be implemented with a hypertonic in order to do implementation-specific things like hardware access.

For persona-pairs or pseudo-persona-pairs that have incompatible invariants; silhouette-pairs; and dissimilar pairs; in general, programmers should not use relay-hooks unless necessary. Construction and conversion hooks to achieve the same effect is preferred. If, after consideration, it is decided that relay-hooks are still necessary, then the relay-hook should adopt the idiom of moving whatever enclosed objects are needed into temporaries, then applying them to the new object.

Corollary 1. *Funcs in the derived type, not to mention virtual funcs, must not be called while constructing the base type, because the derived type hasn't been constructed yet. Some type designs rely on the base type being* *able to orchestrate some setup, of which the derived type provides setup customizations. This is typically solved, in languages like C++, by having a separate setup function that has to be manually called, which can be error-prone. Alternatives like the Curiously Recurring Template Pattern are clunky, non-obvious, and hard to teach.*

In languages like Java, virtual methods can be called by the superclass, which relies on the subclass being implemented correctly, which can be error-prone.

TONAL solves this dilemma by allowing both limited derived-type funcs to be evaled in the base construction, and for the programmer to nominate a func as the separate post-construction setup function.

Any func from the derived type can be evaled in base construction as long as it doesn't touch any enclosed object. Such funcs are mainly factories designed to be customized by derived types.

To nominate a func as the setup function, it must be the last func evaled in a construction, and it must explicitly use @derived as the dominant object for the func. The nominated func's evaluation is postponed until the most-derived type is fully constructed. Such a func must only be called once during the entire construction process, but each construction may evaluate one such func, even if constructions are evaluated by other constructions. In the case of nominated virtual funcs, the most-derived override is the only one that's evaluated. The order in which the total set of nominated funcs is evaluated in the construction order. Nominated funcs can touch any enclosed object.

Destruction has the same restrictions, but inverted. Some types needs to be cleaned up from derived-to-base before being destroyed, and at each destruction, parts of the object will have already been destroyed. Any derived-type func that is evaluated in destruction must not touch any enclosed object. The nominated func must be the first func evaled in each destruction. Each nominated func must only be evaled once. The order in which the total *set of nominated funcs is evaluated in destruction order, immediately before destruction. Nominated funcs can touch any enclosed object.*

If one nominated func, in construction or destruction scenarios, are called from another nominated func, then it would also trip an unatonable condition if it is called more than once in the entire construction or destruction phase.

Relay-hooks employ the same pre-destruction and post-construction func nomination scheme for the expiring-type and relayed-type, respectively. The pre-destruction nominated func is the @derived-dominant func as the first tonic, and the post-construction nominated func is the @derived-dominant func as the last tonic.

18.0.2 Access control

The classic access modifiers - used to guard against improper use and accidental coupling - are *public*, *private*, and *protected*. Languages like Java and C# have even more, for packages and other levels of organization. Other keywords in other languages that also serve as access modifiers of sorts are *abstract*, *default*, *extern*, *fileprivate*, *final*, *friend*, *inline*, *internal*, *open*, *override*, *readonly*, *sealed*, *static*, *thread_local*, *using*, *virtual*.

Many of these keywords are strictly not categorized as access control modifiers in those languages. TONAL takes the point of view that the notion of access is more extensive than just access to names; names are just one of the observable structures of the AM. The other main observable structure of the AM is the dispatch. Being observable means access to them must be disciplined and restricted.

The classic access modifiers can at times be too broad, and at other times be too narrow, requiring workarounds like friend access, or access token objects, or very intricate rules with intricate implications that is easy for computers to check but impossible for programmers to remember.

Access-control in programming languages are not for security, but the same principles from access-control-lists can be re-used for clarity. Access-control is simple: what we are controlling access to, what access capabilities are allowed, and what is allowed to access. They should be specified as explicitly as possible, while avoiding detail bamboozlement.

Name access-control uses a stack-like database structure. Every type has its own access-control database. Every access-control database for a type starts empty from the beginning of the base subtonic. Modifying the access-control database can utilize the stack-like semantics to avoid repetition, while still being reasonably simply to reason about whatever has access to whatever name, with whatever capability.

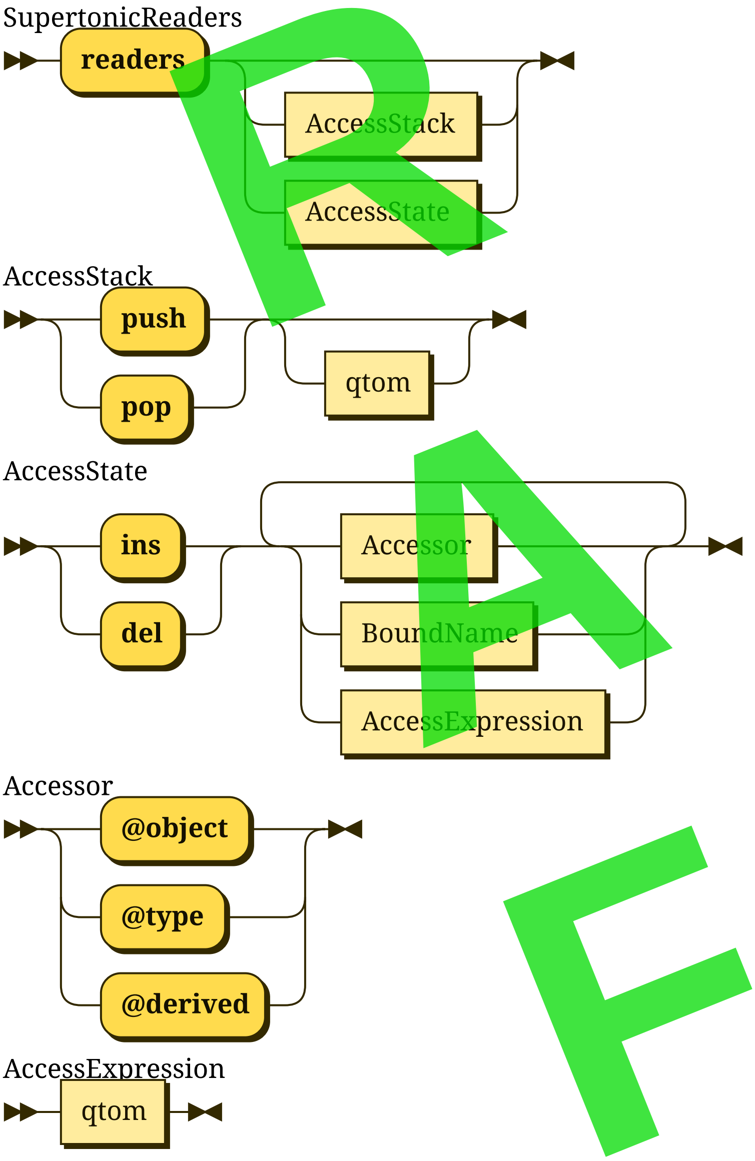
The stacked approach also makes it easier to group enclosed objects by permitted access, which often coincides with grouping by related functionalities.

Dispatch access-control follows the Java-like scheme of a single-use supertonic that only applies to the subsequent enclosed Tensah. Unlike name access-control, it is much more important to prevent accidental granting of dispatch access.

Name-search and pruning are performed before access-control checking. This prevents access-control from being intercepted with a new overload with more capabilities.

18.1 Supertonic Readers

Figure 18.10: Supertonic Readers railroad diagrams.



a

[a]

The *readers*-supertonic without subjects resets the access-control to its initial state. The initial state only allows access to names from within tonics enclosed by the type. This is exactly equivalent to C++ classes, where everything is *private* by default. Additionally, in the initial state, names can only be accessed implicitly or explicitly through the @object. This is exactly equivalent to C++ functions and data being non-static members of a class.

It is a bit weird that the default class membership in C++ is differentiated from static membership by the term “non-static”, when they are more common than static members. In TONAL, it is termed @object-accessed name/enclosed object, but can be assumed if omitted in exposition.

The current state of access-control is applied to all names following a *readers*-supertonic until the next *readers*-supertonic modifies the state. The current state may be a useful configuration for a subset of names, so it can be pushed onto the logical stack of the access-control database to use again later. The pushed-state may be named for documentation and targetted popping.

Popping an empty stack trips an unatonable condition.

Popping to a named stack-entry pops all intervening states until the named stack-entry is reached. If the name does not exist, it trips an unatonable condition.

Readers can be inserted or deleted from the current access-control state. Readers are somewhat the equivalent of friends in C++, in that you specifically nominate what is permitted access. In C++, friends of a class can access any member in a class, regardless of access-control. However full access is almost never necessary, and opens the door for unchecked, incorrect, usage. Readers can only access the names for which they were permitted access. If a reader is both inserted and deleted, it trips an unatonable condition.

There are special shortcuts to specify a large number of readers that are familiar to other languages.

@object has access to all names by default. The object dominant of an atom is the @object. Names can be accessed via an @object, either explicitly or implicitly, with the @object having the TOWEL of all the names. If a func is evaluated via an @object (implicitly or explicitly), then name access from within the func is also through the @object, unless explicitly stated otherwise. This is the equivalent of non-static members in languages like C++.

@type has no access to all names by default. The type dominant of an atom is the @type. If @type is given access, this is the equivalent of static members in languages like C++. C++ allows static members to be accessed via an object's pointer, but always refers to the static member. Unlike languages like C++, both the @object and the @type can enclose objects with the same name, and they view separate objects. This is to aid generic programming. TONAL types are present-tense singleton objects, so there is no reason why types cannot be passed into generic funcs and have algorithms that work on objects also work on types.

TONAL does not have a specific feature that directly mirrors C++'s function-scope static variables. The same effect can be achieved by giving reader-and/or-writer access to only @type, and the enclosed func that the enclosed object is intended for. It is slightly more cumbersome, but it does have the nice property that the enclosed object will be initialized when the enclosing type is, rather than the random time whenever the func is evaluated.

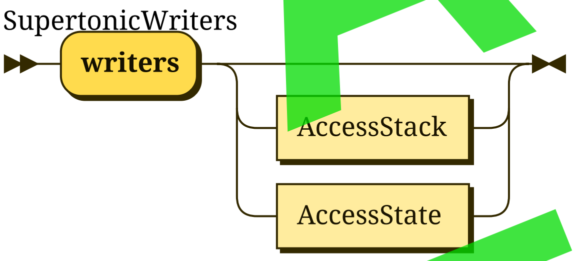
For names with both @object and @type access, type dominants cannot access names through an @object, but object dominants must only access names through the @type explicitly.

Access for derived types is granted through @derived readers. It is the equivalent of protected access, in languages like C++. Permitting access only to derived types may not be as high-coupling as permitting everything access, but it is still notoriously dependency-inducing. It would still be better to name types and funcs directly to give access to.

@object and @type access is the equivalent of private access if provided just on their own. To give general access beyond select types, an access-expression can be given, which is just a qtom wildcard. TONAL tries to reserve as few characters as possible, which means there is limited option for wildcards in atoms. Qtoms have no such restriction. Since the @ character is forbidden in non-@jective names, and parentheses are reserved for tonic delimiters, they are the natural choice for wildcard characters. @, for single character wildcard. @@, for any-length characters non-greedy wildcard. Parentheses, for any-length limbs non-greedy wildcard. The wildcard expression is matched starting at the complete rooted branch of a name.

18.2 Supertonic Writers

Figure 18.11: Supertonic Writers railroad diagram.



a

a

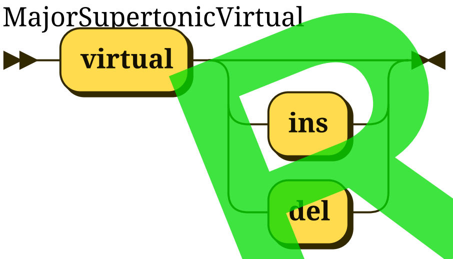
All writers have reader access. If a writer is deleted from the current state of the access database, and was also never added to the readers in the current state, then the implicit reader access is also revoked.

Writers, applied to funcs, means the func can be evaluated with a TOWEL-owning dominant. TOWEL transfer rules still apply, so the func will still have to pull-in its @object before it can modify anything.

Deleting writer access to a let-name is the equivalent of const data members in languages like C++.

18.3 Supertonic Virtual

Figure 18.12: Major Supertonic Virtual railroad diagram.



[a]

Hooks cannot be given dispatch access, or will trip an unatonable condition.

Access to a dispatch is not stateful, so the virtual-supertonic must be followed immediately by a, and only applicable to that, Tensah. This prevents accidental granting of access to the dispatch of more Tensahs than expected.

The applicable func is inserted into the dispatch-access database to enable it to be overriden by funcs in derived types, at any level of derivation. This is equivalent to the virtual keyword in languages like C++. If it would have the affect of overriding a virtual func in the base type, then it trips an unatonable condition.

The applicable func can be deleted from the dispatch-access to prevent being overriden by funcs in derived types. This is equivalent to the final specifier of member functions in languages like C++.

The applicable func attempts to override the base type's virtual func, if neither inserted nor deleted. If there is no func in the base type that is overridden by the applicable func, this trips an unatonable condition. This is equivalent to the override specifier in languages like C++.

A derived type's access to override a virtual func is not affected by reader or writer access. It is in fact recommended to keep virtual funcs

[a]

only and solely @object-accessible. This prevents derived classes from evaluating the base virtual func directly, likely outside of the constraints of the base type. The base type typically provides virtual funcs as customization hooks that it will evaluate in some precisely orchestrated manner. The Template-Method Pattern is one such example, and one could argue that the Non-Virtual-Interface Pattern is a special case.

There are design patterns (some would argue anti-pattern) that requires overriding funcs to understand why and when to call the base type's virtual func, but these typically have very narrow type hierarchies intended only for highly specific implementation needs.

Destruction is automatically handled with dispatching if a type has any virtual funcs, so the virtual-supertonic is not needed for destructors, unless the destruction is the only virtual func. If a destruction is deleted from virtual-access, it only prevents derived-types from providing overrides. The implicit destruction of the derived-type is still virtual.

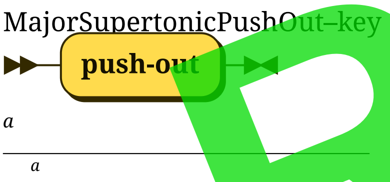
All hooks, aside from destruction hooks, cannot be tagged virtual, otherwise it trips an unatonable condition.

The applicable base object, whether it be a Tensah or a present-tense value, is neither inserted nor deleted from dispatch access. Rather, it is to signify that the base object participates in diamond inheritance.

All types are dispatch-acessible by default, in the sense that it is always possible to derive from a type with, or to impart, dispatch ability, so inserting into dispatch-access is an unatonable condition. If deleted from dispatch-access, then the type can not be derived from. This applies whether or not the type has any virtual funcs defined. This is the equivalent of the class/struct final specifier in languages like C++.

18.4 Supertonic Push-Out

Figure 18.13: Major Supertonic Push-Out railroad diagrams (key).

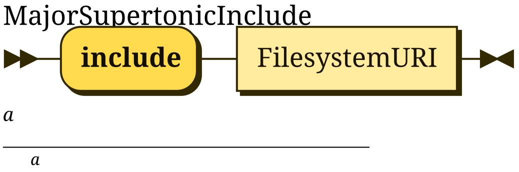


The immediately following let-name's TOWEL is not tied to the enclosed object, unlike normal let-names in types. This also disables any TOWEL transfers for any object of enclosing-type, tripping an unatonable condition if attempted. The let-name must be constructed with a value provided outside construction, otherwise it trips an unatonable condition. This is the equivalent of reference-qualified data members in languages like C++.

If the provided value was pulled-out, then any TOWEL round-trip on the enclosed let-name occurs after the destruction of the enclosing object. Such usage patterns are useful for self-contained, slightly longer-lived, operations on an object, such as builders, otherwise known as fluent-interfaces.

18.5 Supertonic Include

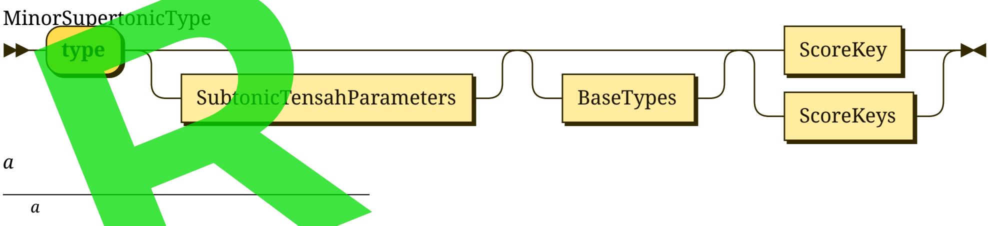
Figure 18.14: Major Supertonic Include railroad diagram.



This section intentionally left to this one sentence.

18.6 Minor Form

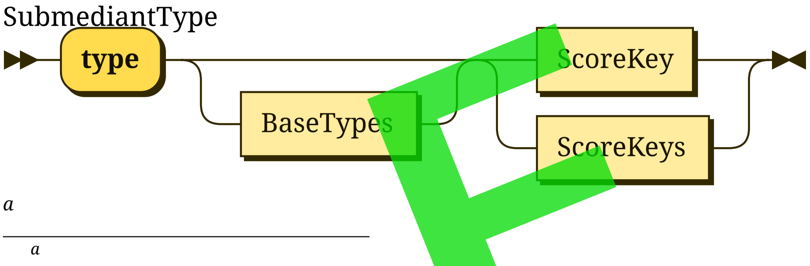
Figure 18.15: Minor Supertonic Type railroad diagram.



Some generic algorithms may take a type argument, but the type may not have any usage outside of a func, so create a type inline if it otherwise would be hard to justify creating a fully-fledged type.

18.7 Submediant Form

Figure 18.16: Submediant Type railroad diagram.



Create an object of a nameless type.

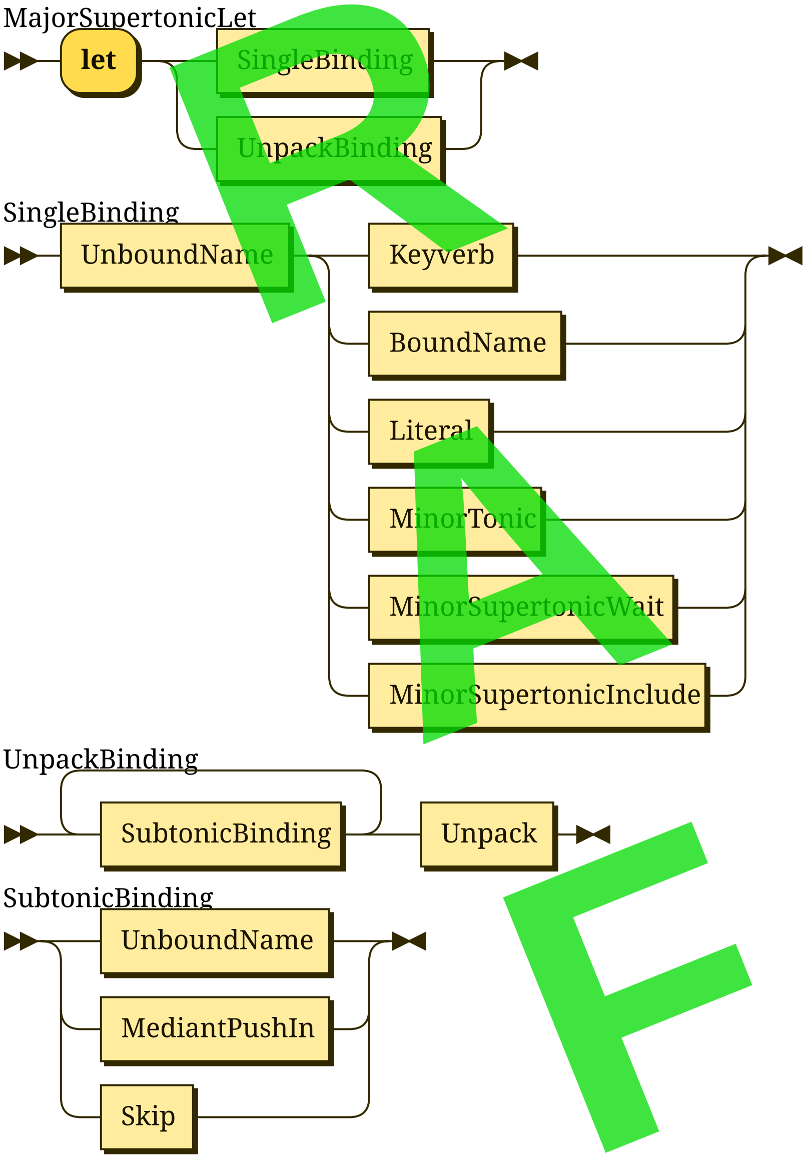
Not much point for access control supertonics, but it’s harmless to do so, so it is allowed, to simplify AM implementation.

Possible to inspect @type object.

Like Java.

19 Supertonic Let

Figure 19.1: Major Supertonic Let railroad diagrams.



[a]

Names introduced by the *let*-supertonic can bind to either an object constructed as its subject, or to an already existing let-name, including parameter names. Of the latter kind, the let-name does not have the ob-

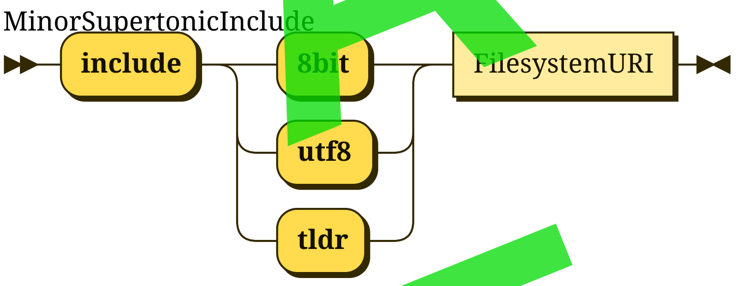
[a]

ject's TOWEL, much like with parameter names. The former and latter kind are referred to as pull-in binding and push-out binding, respectively. Pull-in binding has the object's TOWEL, where as push-out binding needs to be pulled-in - creating a copy - just as with parameters, when desiring an object that can be modified.

An object can have many push-out bound let-names, but only one pull-in bound let-name. This prevents aliasing, and allows for certain memory optimizations, similar to restricted pointers in languages like C. Naturally, it also prevents many logic errors that normally arise when reading and writing to objects using different name aliases.

19.1 Minor Supertonic Include

Figure 19.2: Minor Supertonic Include railroad diagram.



a

a

Programs consists not just of instructions - commands - but also of data. Data can become part of the AM process so that TONAL programs can be driven in the present-tense.

TONAL only understands a limited number of data types: raw binary, raw text, and TLDR. Raw binary is interpreted as a List of Longs - each long representing an octet. Raw text is interpreted as a Qtom. TLDR is interpreted as an object of unspecified type. The data cannot be modified and, being present-tense, will not be preserved into the final program.

The TONAL program derives meaning from the data, so the data lives on in the structure of a TONAL program.

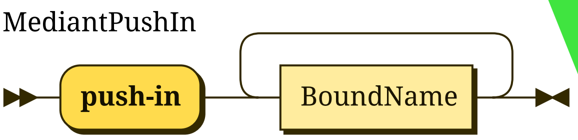
One use of present-tense data is for digital assets. The AM is thus made aware of digital assets as part of the program itself, potentially saving the program from having to load and reparse the data every time the program runs.

Structured present-tense text data can be used as configuration. This is not your average configuration of program options. Entire structured documents can be embedded in a program.

Raw text data is completely free-form. Combined with present-tense parsing, TONAL programs can be augmented with domain-specific languages with wildly different syntax, all outside of TONAL. As with all languages, no single syntax can hope to provide safety, terseness, and expressiveness for all types of problem domain. For example, declarative languages specify relationships, while imperative languages (like TONAL) specify algorithms. Imperative code can sometimes obscure relationships between entities in the quest for performance, so having a declarative DSL can help the programmer see an architecture from multiple facets.

19.2 Mediant Push-In

Figure 19.3: Mediant Push-In railroad diagram.



[a]

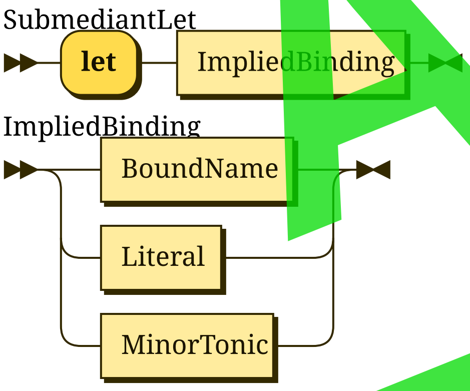
[a]

An object can be unpacked into new let-names and/or existing let-names. Unpacking into existing let-names requires pushing-in, as is the case with regular push-in. The mediant push-in makes it easier to mix the targets of the unpack.

Nested unpacking is not directly supported by TONAL, to keep the ergonomics of names maintainable. If unpacking of objects multiple levels of enclosure is desired, then it is up to the programmer to explicitly do it for each object that is unpacked. TOWEL rules ensures that no frivolous copies are made.

19.3 Submediant Form

Figure 19.4: Submediant Let railroad diagrams.



a

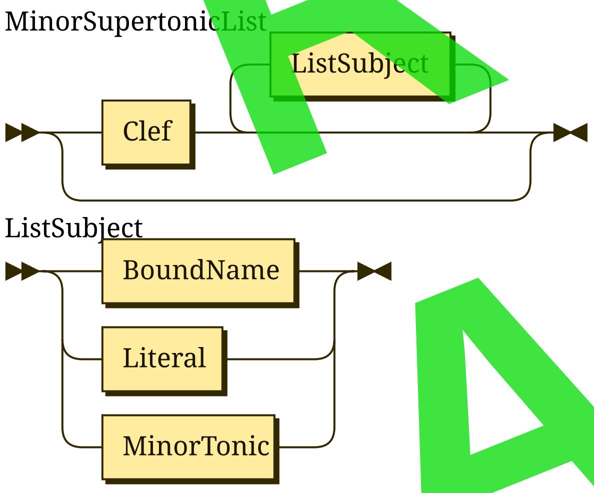
a

Only used for default parameters for Tensahs and the range sequence in loops. In both cases, the name is supplied outside of (immediately preceding) this tonic.

The name is bound as late as possible, which means when the evaluation happens. Default parameters are created with the values at the location of the Tensah's evaluation, as opposed to the construction of its enclosing object.

20 Minor Supertonic “()” (the list specifier)

Figure 20.1: List Specifier railroad diagrams.



[a]

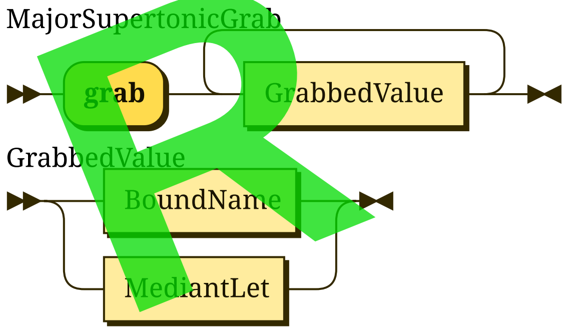
[a]

The atom “list” is a very useful generic word, so TONAL avoids using that atom as a verb. The clef is obviously a list, so that is used as the verb to construct a list.

TONAL lists do not act as a container, like arrays or vectors or lists in many other languages. TONAL lists are more like a way of being able to manipulate syntactical sequences of tonics, names, and literals. If a list subject is pushed-in, for example, it is not pushed into the list, but rather pushed-into whatever other tonic that list is used by. Names in lists are searched-for in the immediate scope to avoid accidently picking up the name that would be found starting from where the list ends up.

21 Supertonic Grab

Figure 21.1: Major Supertonic Grab railroad diagrams.



[a]

[a]

Grabbing names from outside into a Tensah is a great way of composing data and customizing behaviour that can be passed into generic interfaces. It is a fast way of creating cheap, stateful, objects.

Grabs are just like parameters when it comes to TOWEL rules. Grabs must be pulled-in to make a local copy. Grabs are candidates for TOWEL roundtrips.

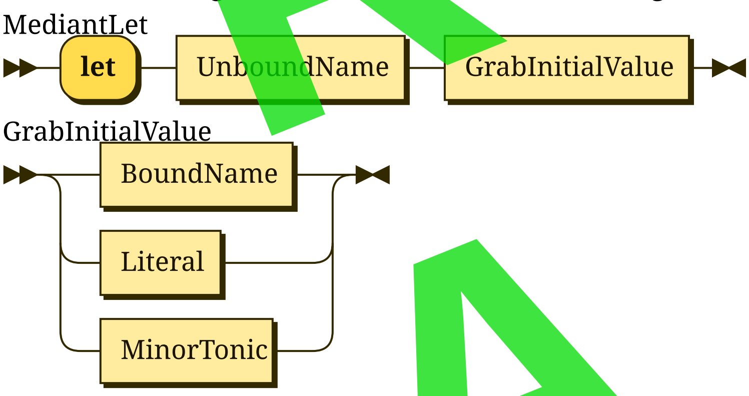
Grabs that aren't pulled-in, or are roundtripping, render the grabbing Tensah TOWEL-untransferrable. They themselves cannot be pulled-in, unless also pushed-out and triggering the roundtrip of the Tensah.

Grabbing the dominant @object or @type makes the name-search behave the same as a regular type-enclosed func. All dominant funcs implicitly grab its @object. TONAL does not support the concept of partial-objects, so Tensahs that grab the @object cannot transfer TOWEL, unless it is a TOWEL roundtrip.

A grabbed @object need not take up memory in the AM (eg, a pointer) if the grabbing Tensah is accessed via a dominant. Such an optimization is possible if the grabbing object is enclosed by the @object's type, as is the case with enclosed funcs. Enclosed objects that grab the @object acts as proxies. These lighter-weight proxies make designing extended object-systems, or meta-objects[7], such as those found in Smalltalk or LISP CLOS, much more resource efficient.

21.1 Mediant Let

Figure 21.2: Mediant Let railroad diagrams.



a

[a]

Being able to grab a value that is constructed in the $grab$-supertonic itself is provided for convenience. It is equivalent to constructing an object, and then pushing-in to the grabbing Tensah, but with the added benefit that the name is only lexically valid inside the Tensah. The corresponding feature in languages like C++ is the init-capture in lambdas.

[7] In the sense of observable message passing between object ports.

Part V

Control Flow

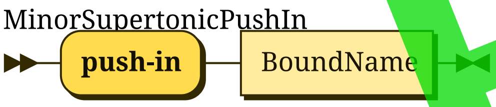
22 Tonic

The most fundamental method for principled flow of control is the subprogram. Many programming introductions start with choice and repeat, but the subprogram should be considered the primary tool for programmers. Breaking down a large task into smaller ones, simplifies, and tightens the focus of sections of code. In certain cases, such break-downs allows a machine to perform each section concurrently.

Subprograms can be evaluated in such a way that no actual change in the flow of a program actually happens, such as inlining. Subprograms in this context still have value in tightening focus. In TONAL, some subprograms can be evaled in present-tense and can be completely eliminated so inlining can be made unnecessary.

22.1 Minor Supertonic Push-In

Figure 22.1: Minor Supertonic Push-In railroad diagram.



a

a

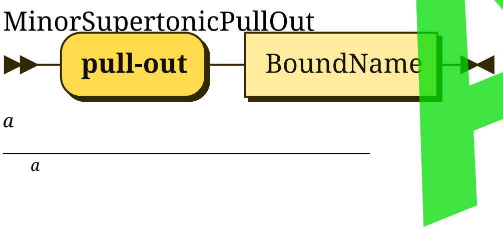
All objects constructed by a minor tonic is implicitly pushed-in, because there isn't a way to reference the object. In all other cases, objects must be explicitly pushed-in. Objects must be pushed-in by let-name, to avoid the complication of having to track the TOWEL of objects being passed into other tonics and then passed back out.

A pushed-in let-name is in past-tense immediately at the opening a parenthesis of the enclosing major tonic. It prevents a let-name being used in any other position in the enclosing major tonic and minor tonics. This avoids any intricacies about whether a let-name can be used as another subject prior to the push-in, and what happens in the receiving tonic where multiple parameter names refer to the same object, but one name has lost TOWEL. There can be such a rule, but for ergonomics, it's better left as a condition.

The tonic that the object is pushed-in to now has that object's TOWEL.

22.2 Minor Supertonic Pull-Out

Figure 22.2: Minor Supertonic Pull-Out railroad diagram.



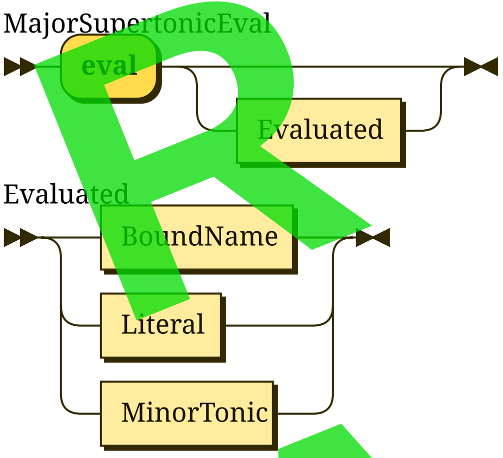
Pull-out marks the let-name for a potential TOWEL round-trip. The receiving tonic may or may not utilize the round-trip, but it may not take the object's TOWEL. At most, it can take a copy of the object identified by the let-name, and have the copy's TOWEL, if the object's type allows copies. Whatever happens to the object inside the receiving tonic is invisible unless the TOWEL round-trip is complete.

The let-name being pulled-out must have the object's TOWEL, otherwise it trips an unatonable condition.

A pull-out minor supertonic can be considered to be implicitly pushed-in for the duration of the enclosing major tonic. This means all subjects of the same enclosing major tonic cannot view the same let-name, because the let-name is considered to have lost its TOWEL due to being pushed-in. TOWEL is restored after the major tonic has been evaluated.

23 Supertonic Eval

Figure 23.1: Major Supertonic Eval railroad diagrams.



a

a

The *eval*-supertonic defines what a func evaluates to. All funcs are evaluated to some value. Sometimes a func just ends when there are no more commands left to process. This implicitly evaluates to an empty present-tense list. An *eval*-supertonic with no subject also evaluates to an empty present-tense list. An *eval*-supertonic may evaluate to an explicitly provided value.

A func can have zero, to many, *eval* supertonics. All *eval* supertonics in a func must evaluate to the same type in the present-tense. This might involve an explicit conversion to a base type. In such a case, if all the derived types of all the possible evaluations are known, then it is possible for the AM to reserve space in the present-tense that can hold the largest of them.

A func can evaluate to a value at any point in its bars, in which event the func is finished and no other commands after are evaluated. If there are commands after an *eval* that never get evaluated in any circumstance, that trips an unatonable condition.

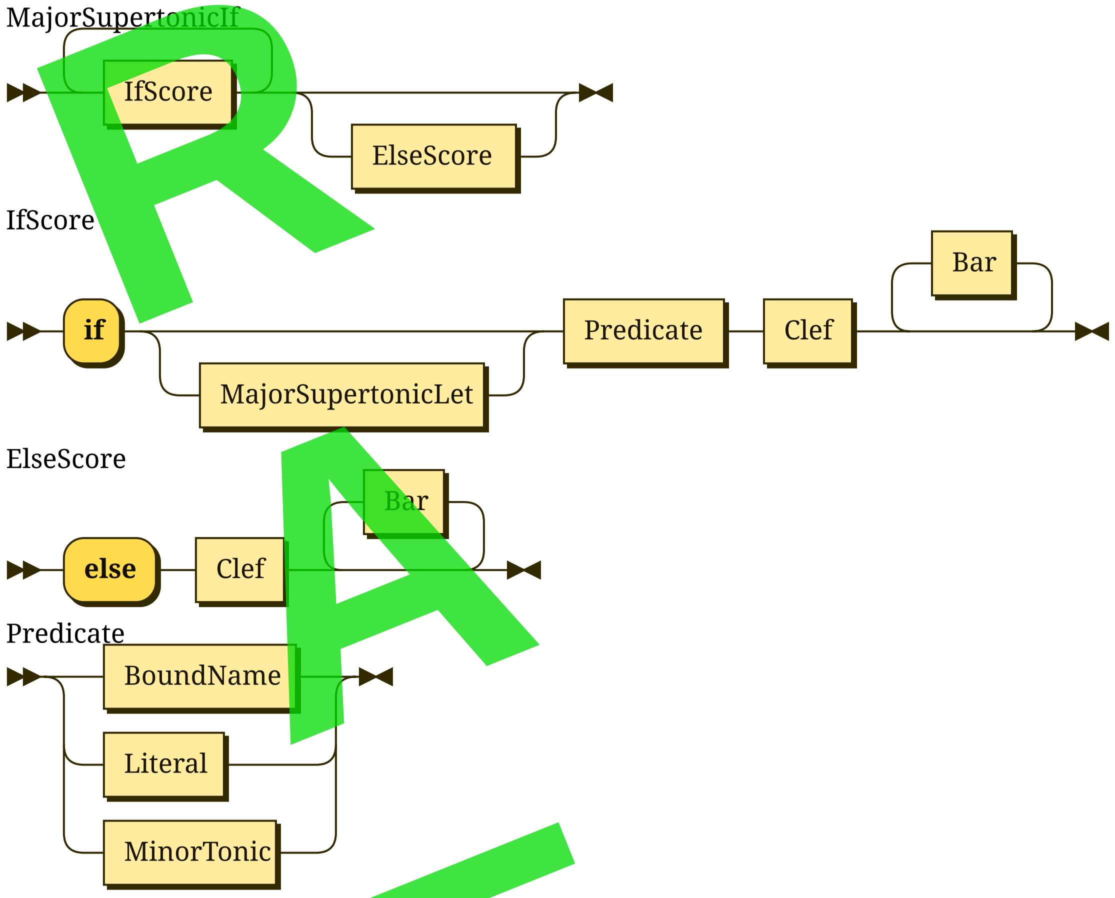
A func may be resumable if it also evaluates a *give*-supertonic. The *eval*-supertonic then finalizes the evaluation of a resumed func. All *give* supertonics must also evaluate to the same type as all the *eval* supertonics.

A func can evaluate to multiple values at the same time if the *eval*-supertonic subject is convertible to an archetype list of values.

Sometimes a func evaluates to present-tense values. The AM can optimize further evaluations of the func in present-tense and bypass it completely if there is a one-to-one mapping from the func's enclosing object and arguments types and/or values.

24 Supertonic If/Iff

Figure 24.1: Major Supertonic If railroad diagrams.



a

a

Making decisions based on the state of the world is the mainstay of control flow. The predicate of an if-score tests that state for meeting certain criteria and evaluates only the bars of the score that meets that criteria, or the $else$-score if none of the criteria are met.

The predicate evaluates to some value that is convertible to a $long$ archetype value, where the value 0 means that the criteria is not met, and any other value means the criteria is met. If the value is not convertible to a $long$ archetype value, then it trips an unatonable condition.

Every if-score is permitted a let supertonic; the object(s) to be created is completely bound to the if supertonic's evaluation. This avoids objects being created solely for the purposes of the if supertonic to have its TOWEL longer than necessary. A common case is the creation of an object that holds the lock for a mutex that should only be held for the shortest amount of time possible. The object's TOWEL continues to the end of the supertonic evaluation, so can be used in subsequent if-scores (if the name isn't shadowed by another).

The design of TONAL strips away the extra $else\,if$ that other languages have, since the S-expression syntax renders it unnecessary. Any ifatom that appears as the top-level subject of the if supertonic is considered the start of a new decision branch; and the $else$ atom as considered the start of the fallthrough decision branch. There is no ambiguity, as plain atoms are not allowed in the bars.

The clef to introduce the bars of the score is not designed to be omitted in the way that some languages allow omitting braces for single-statement blocks. This is to aid visual scanning, as well as tools; and with automatic delimiter matching in most IDEs, saving a tiny little bit of typing is no longer a valid reason. Missing clefs trip an unatonable condition.

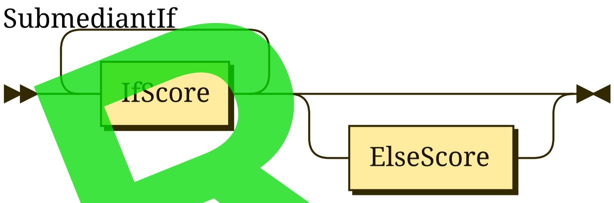
The iff variant of the supertonic requires that one branch must be taken, its name being a reference to the logical qualifier if-and-only-if. This can be used to force the complete check of ranges of criteria to ensure that no gaps in logic exists. If the criteria-ranges are known in present-tense, then an uncovered gap trips an unatonable condition.

In certain cases, the AM can optimize an if-supertonic to a jump-table. This can mildly speed up some code.

TONAL does not have a switch or match statement as in other languages. The features of both - jump-tables, and forced gap coverage - are accounted for by the if and iff supertonic.

24.1 Submediant Form

Figure 24.2: Submediant If railroad diagram.



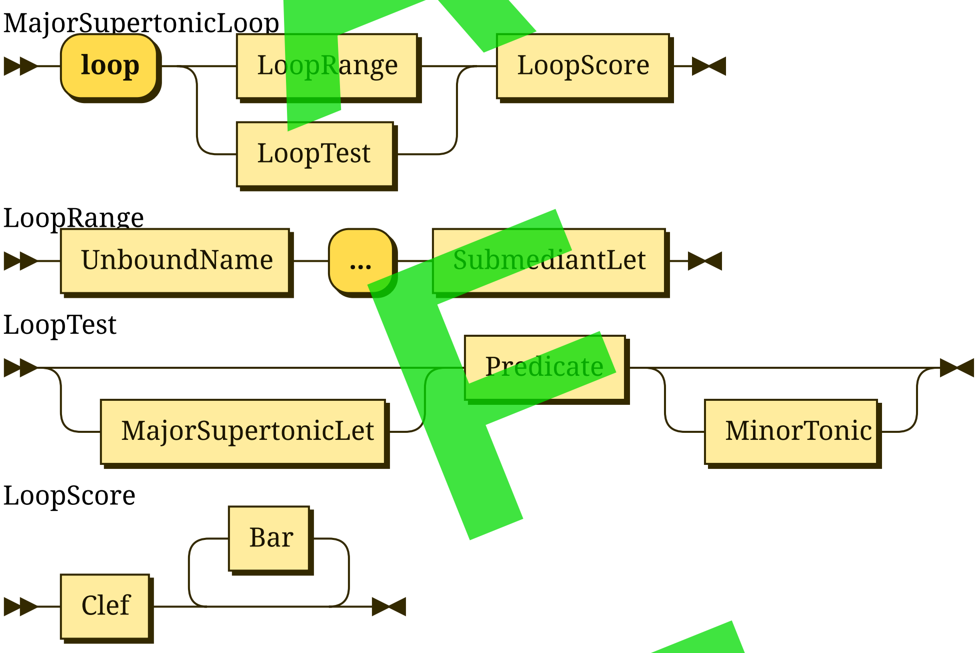
[a]

[a]

Subsection intentionally left to this one sentence.

25 Supertonic Loop

Figure 25.1: Major Supertonic Loop railroad diagrams.



[a]

[a]

Doing the same task over and over is the purview of machines. Repeated tasks complete when some state is reached. A common class of task repetition is the application of some action(s) over a sequence of things until the sequence is complete.

The $loop$-supertonic handles the general repetition of tasks, and the common repetition over a sequence - called a range.

The range is specified with the same kind of let-submediant that is used for defaulted func arguments. The range loop implicitly introduces a let-name that views the current object in the range being looped over. TOWEL rules means that if the object in the range itself has TOWEL, then it can participate in a TOWEL round-trip. Otherwise, pulling-in then pushing-out the implied let-name follows the usual rules for copying.

The range object must be convertible to a $list$ archetype value. It may be empty. If the value is not convertible to a $list$ archetype value, then it trips an unatonable condition.

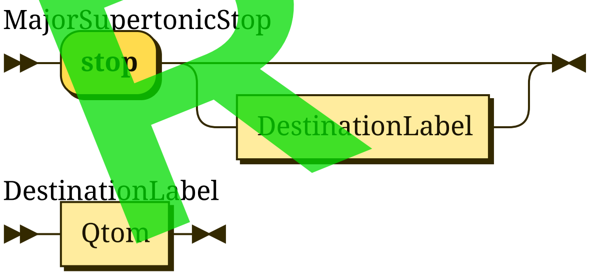
The general loop has an optional let-supertonic and predicate that is the same as an if-supertonic, but with an optional extra after-loop action that can evaluate some state to update the loop's completion status. The predicate checks the loop's completion status, stopping the loop if it evaluates to 0.

The after-loop action is evaluated after the bars. Then evaluation continues back at the predicate, followed by the bars, if the loop has not completed.

A loop may be unrolled for optimization. Loops may also be partially unrolled if the objects in the range are of mixed type.

25.1 Supertonic Stop

Figure 25.2: Major Supertonic Stop railroad diagrams.



[a]

[a]

Loops may need to exit early. The *stop*-supertonic exits the immediately enclosing loop, if no subject is given. The after-loop action is not evaluated if a loop is stopped.

Sometimes it is necessary to exit nested loops to any arbitrary level. Looped tasks that support this can name the loop using a label supertonic that immediately preceeds the *loop*-supertonic. The *stop*-supertonic's label subject specifies which enclosing level of labeled nested loop to exit to. Evaluation continues immediately after the loop named for stopping.

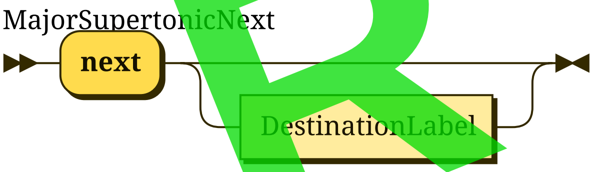
If there are tonics that are never evaluated due to being jumped over under any circumstance, that trips an unatonable condition.

Labels that are outside of the enclosing func-supertonic, or the enclosing verbed-submediant, are not eligible destination labels, and will trip an unatonable condition.

If the destination label is outside of an enclosing *trap*-supertonic, then the $finally$-bars must still be evaluated.

25.2 Supertonic Next

Figure 25.3: Major Supertonic Next railroad diagram.



[a]

Sometimes it is only necessary to evaluate a loop's task only partially and fast-forward to the subsequent iteration. The *next*-supertonic causes the evaluation to fast-forward to the after-loop action of the immediately enclosing loop without evaluating any of the bars subsequent the *next*-supertonic. The loop further evaluates as normal.

Sometimes it is necessary to fast-forward an outer loop from within a nested loop. The label's name is also used for this purpose. The enclosing loops are stopped up to the level of the loop named for fast-forwarding. Then evaluation continues at the after-loop action of the named-loop. The loop further evaluates as normal.

If there are tonics that are never evaluated due to being jumped over under any circumstance, that trips an unatonable condition.

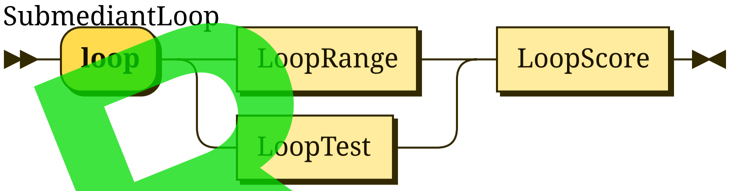
Labels that are outside of the enclosing func-supertonic, or the enclosing verbed-submediant, are not eligible destination labels, and will trip an unatonable condition.

If the destination label is outside of an enclosing *trap*-supertonic, then the *finally*-bars must still be evaluated.

[a]

25.3 Submediant Form

Figure 25.4: Submediant Loop railroad diagram.



a

[a]

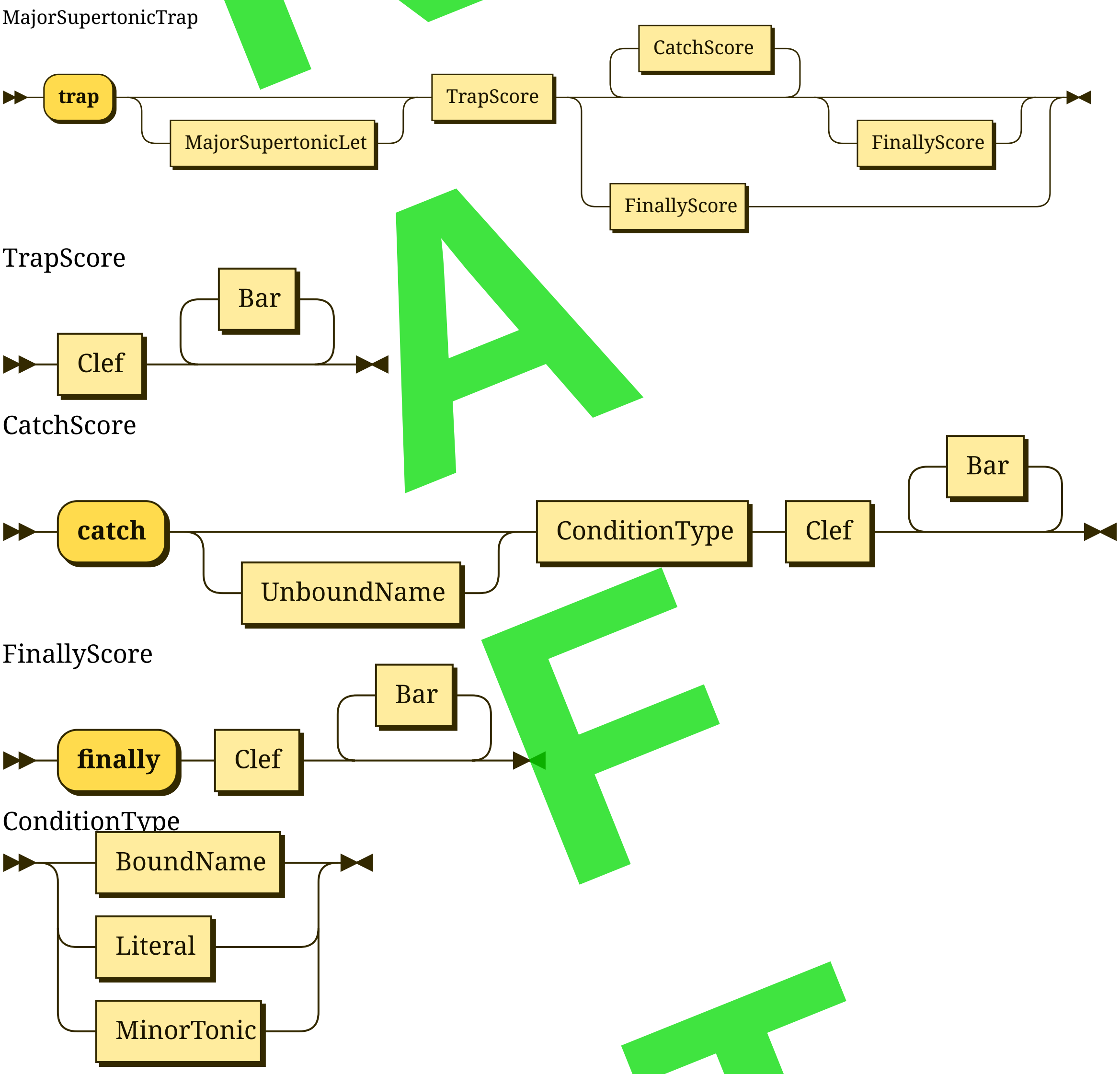
Subsection intentionally left to this one sentence.

Part VI

Control Jump

26 Supertonic Trap

Figure 26.1: Major Supertonic Trap railroad diagrams.



MajorSupertonicTrap

TrapScore

CatchScore

FinallyScore

ConditionType

[a]

[a]

The optional let-supertonic has the same purpose as in the if-supertonic. The let-name allows the object to be used in the trap-handlers; whereas objects constructed in the bars cannot be used in the trap-handlers.

If the let-name initializer trips a condition, then the immediate $trap$-supertonic just introduced is able to catch it if it has the correct trap-handler. This avoids requiring another enclosing level of a trap-supertonic just for objects intended to be used by the trap handlers.

If the initializer trips a condition, then the let-name is not accessible. Trying to access it from a trap-handler trips an unatonable condition. This even applies to trap-handlers that coincidentally handle the same condition(s) that is/are tripped inside the bars. In those trap-handlers, trying to access that let-name trips an unatonable condition.

The type or value of the condition specified must be known in present-tense. Multiple-dispatch overload-pruning rules are implicitly applied to find the best handler.

TOWEL is the recommended way to ensure clean-up operations are always performed, however it may over-complicate the look of the code. For example, maybe only local state needs to be set to a known state, which will only require a few simple tonics, rather than a type that has to be defined. These clean-up actions in the $finally$-score are always evaluated, whether or not a condition was trapped.

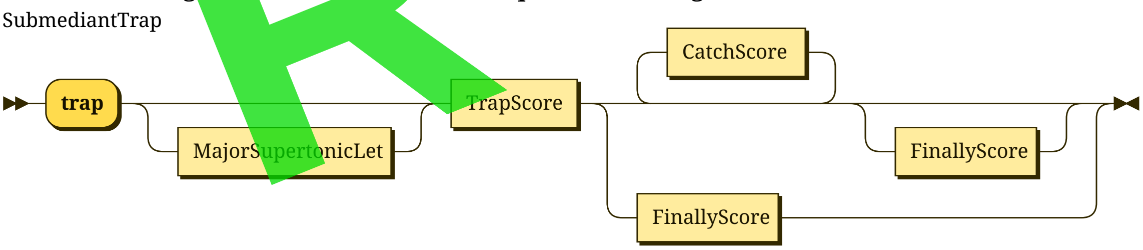
The condition object can be pulled-in, and even participate in the TOWEL round-trip as if it was pulled-out by the AM. This is useful for nested conditions that need to provide specific evaluation status for diagnostic purposes.

The AM tracks all conditions that can possibly be tripped while evaluating a func, and also those that are successfully trapped. Any trap-handler specified that will never be encountered by the known set of possible conditions of a func trips an unatonable condition.

Not all conditions must be trapped, nor reported to be tripped. This reduces code clutter by removing the necessity for all funcs to account for conditions that they have no meaningful action to perform.

26.1 Submediant Form

Figure 26.2: Submediant Trap railroad diagram.



a

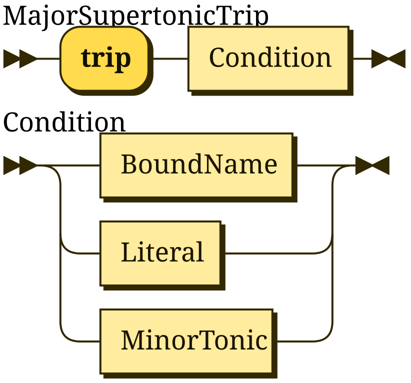
---

a

Subsection intentionally left to this one sentence.

27 Supertonic Trip

Figure 27.1: Major Supertonic Trip railroad diagrams.



a

---

a

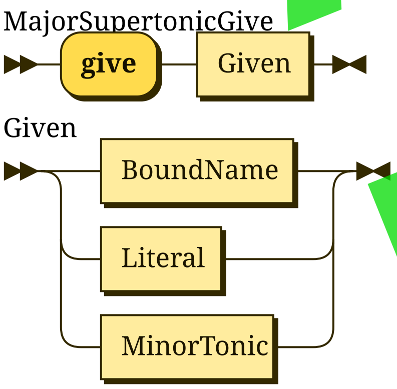
Any object type can be a trip condition. All evaluations are aborted, all TOWELs cleaned-up, until an enclosing $trap$-supertonic, with a relevant handler is found.

A trap-handler can trip a condition. Reasons to do this would include translating an error to a more relevant, more specific, condition to relay better information about the intent of an error. Or the handler merely wishes to notify, or be notified, about a condition, but continue to pass it on.

Present-tense values that are tripped as conditions must be handled in present-tense, or else it trips an unatonable condition. Present-tense conditions allow programmers to extend program correct-ness checks beyond language rules. API usage/semantic errors and regressions reported in present-tense forces programmers to fix them as early as possible before they make it out into the wild.

28 Supertonic Give

Figure 28.1: Major Supertonic Give railroad diagrams.



[a]

[a]

The presence of a $give$-supertonic causes the enclosing func to retain its resumability.

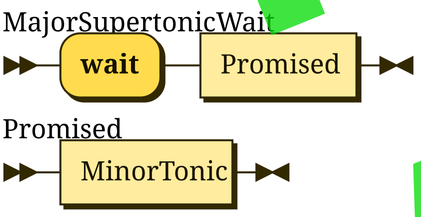
Resumable funcs can give multiple evaluations before the final evaluation. The given type must be compatible with the final evaluated type, just like with *eval*-supertonics.

The func is always resumed after the *give*-supertonic last evaluated.

Bars enclosed by control-flow and control-jump supertonics inside submediant forms, are not allowed to contain *give*-supertonics.

29 Supertonic Wait

Figure 29.1: Major Supertonic Wait railroad diagrams.



a

a

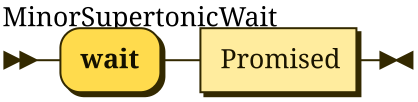
The presence of a *wait*-supertonic causes the enclosing func to retain its resumability.

The func is always resumed at the subject of the *wait*-supertonic last evaluated.

Bars enclosed by control-flow and control-jump supertonics inside submediant forms, are not allowed to contain *wait*-supertonics.

29.1 Minor Form

Figure 29.2: Minor Supertonic Wait railroad diagram.



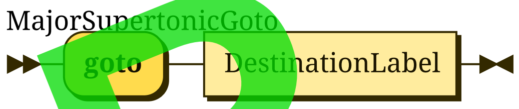
a

a

It is possible to wait on a tonic being evaluated to some object.

30 Supertonic Goto

Figure 30.1: Submediant GoTo railroad diagram.



a

a

Modern programming techniques eschew the need for unstructured jumps for all reasons, including cleaning-up after handling an error. Some workarounds are still worse than unstructured jumps, such as bailing out of nested branches. Flags are required to orderly cascade out of nested branches if there are no unstructured jumps, which could force the programmer to follow the logic in a sea of indentation. Some would argue that branches should not be so nested in the first place.

A simple two-level nested branch that requires a quick exit is significantly more messy using flags, so the *goto*-supertonic is the last vestige of unstructured jumps that can still be used for this purpose.

When jumping from a nested structure, all let-names with TOWEL in the enclosing levels up to the level of the destination is destroyed. Evaluation continues at the destination label.

Labels that are outside of the enclosing func-supertonic, or the enclosing verbed-submediant, are not eligible destination labels, and will trip an unatonable condition.

If the destination label is for an enclosing loop, the evaluation does NOT behave like a *next*-supertonic. Evaluation continues from before the loop.

If the destination label is outside of an enclosing *trap*-supertonic, then the *finally*-bars must still be evaluated.

If an evaluation jumps past a *let*-supertonic, trying to access that let-name trips an unatonable condition. This is one of the dangers of unstructured jumps. The other control-flows and jumps ensure that all clean-up and let-supertonic evaluations happen properly. This avoids the problem of having let-names that have no object altogether.

If there are two viable destination labels with the same qtom, regardless of nesting level, this trips an unatonable condition. Let-names can shadow other let-names from enclosing scopes because the rules for name-search are simple enough to follow. Jumps, on the other hand, can get very messy to follow at the best of times, so allowing labels to shadow would lead to complete spaghetti.

If there are tonics that are never evaluated due to being jumped-over under any circumstance, that trips an unatonable condition.

Part VII

Generic

Liskov/Wing Subtype Requirement is defined as follows[8]:

Algorithm 1 Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Liskov sub-typing is not to be confused with inheritance. Inheritance can be used to model sub-typing, but sub-typing is about purpose, not genealogy. Inheritance is its own operation that incidentally overlaps with the purpose of sub-typing. To wit: a sub-type relationship can exist without inheritance.

Pastry is a general type of food. Doughnuts and croissants are a specific type - a sub-type - of pastry. The sub-type relationship is a categorization. There isn't inheritance in the way that you or I inherit the genetic information from our respective biological parents. In code, we may use inheritance to represent pastry, doughnut, and croissant; but an alternative would be to infer a sub-type relationship in the present-tense. We consider them pastries because they have the same core ingredients and are baked with similar methods.

Sub-typing, whether inferred or inherited, is both a constraint on the relevant and provable properties of a type, but also a promise to not demand of a type beyond what was explicitly required and provable.

Take the classic problem of Rectangles and Squares. A Square IS-A Rectangle where the height and width are the same; so a straightforward representation of that relationship would be for a Square type to inherit from a Rectangle type. The Square will necessarily override behaviour to maintain the equivalence of height and width.

[8] https://doi.org/10.1145%2F197320.197383

One argument[9] goes that Square isn't a proper sub-type of Rectangle because the behaviour of a Rectangle - that height and width can be modified separately - is not preserved by Square. Such behaviour isn't some temporary internal state, but externally observable behaviour[10].

What that analysis misses is that the Liskov/Wing subtype requirement specifically requires "provable" properties. Taken in its most compact formulation, there is no mention of accounting for all properties, all observable behaviour, in every situation. The idea that height and width of a Rectangle-like type are independent is not actually provable, even if no sub-typing is involved. After-all, the simplest mathematical description of a Rectangle does not address mutability. The most basic guarantee of a Rectangle-like shape is that it has four sides at right-angles, and from that follows the area is $height \times width$ and the perimeter is $2 \times (height + width)$.

Squares sub-type of Rectangles actually demonstrates why sub-typing is distinct from inheritance. A Square has no need of both a height and width, yet inheriting from Rectangle would make them mandatory. One can imagine an algorithm that takes a buffer of Rectangle-like objects. An algorithm that demands inheritance from Rectangle, rather than being an inferrable sub-type of a Rectangle, will force a buffer of Squares to be a buffer of Rectangles, either doubling the space-cost by having to keep everything a Rectangle-derived type, or a time-cost of having to convert between Rectangles and Squares. An inference-based sub-typing scheme eliminates both costs entirely.

The moral of the story is there are no single set of properties that will fulfill the Liskov/Wing sub-type requirements for all uses. A base-type has its own set of provable properties that it expects of its derived types. Algorithms have their own set of properties they require provable. These sets may not always be proper subsets of each other, especially in the case of

[9] "The Real Problem"

[10] https://web.archive.org/web/20230314234519/https://www.hyrumslaw.com/

multiple inheritance. When modelling the real world, categories overlap, which is not something that inheritance can efficiently represent, necessitating a separate mechanism for specifying sub-type relationships.

31 LIskov-liKe Equivalence

Composition establishes a HAS-A relationship. Inheritance establishes an IS-A relationship. Sub-typing is a much broader relationship and demands a similarly broad capability to establish the relationship. TONAL mediant forms achieves sub-typing by introducing the LIKE-A relationship that enables a programmer to say more precisely what they mean when something should be “like a” type. LIKE is an acronym for **“LI**skov-li**K**e **E**quivalence”.

It may seem like a good idea at first to enforce that the only allowed operations on an object are those that are entirely specified by the mediant, but that would actually hinder the ability to make code generic. Algorithms are divided into somewhat independent constituent algorithms, perhaps each with their own sub-typing requirements, such as stricter constraints that allow optimized specializations. It would be difficult if not impossible for the higher levels of an algorithm to know all specializations of its constituent algorithms.

Nevertheless, mediants are present-tense objects in their own right, so predicate funcs can be composed appropriately to cover the entire space of constituent algorithm requirements if so desired.

In limited cases, requirements about stateful behaviour can be captured as present-tense programs operating on present-tense objects. A func can indeed document when it needs a rectangle-LIKE object to have independent height and width, without impacting all rectangle subtypes to require this property. This is limited in the sense that it cannot guarantee the behaviour is preserved in future-tense, but a programmer would have to go out of their way to cheat the system by providing a special behaviour just to get around present-tense constraints.

All mediants must be able to run in present-tense, otherwise it trips an unatonable condition. Once a subtyping relationship is proven on a type, or even a value, the AM can optimize away future encounters of the mediant/type-or-value.

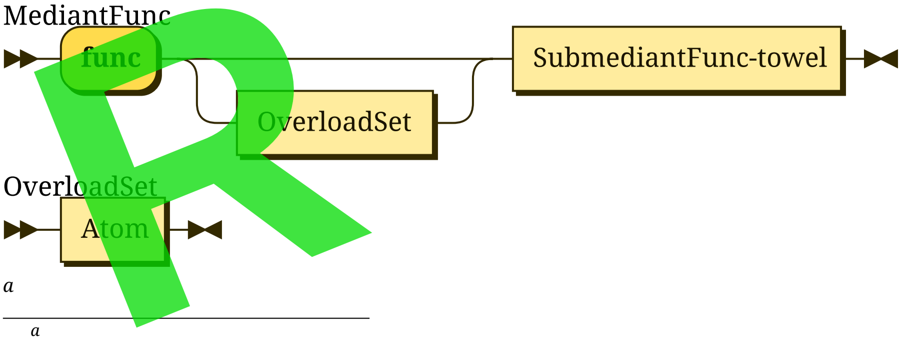
Mediants can view let-names outside of itself, but TOWEL round-trip is disabled for those let-names because it would be too confusing to allow the state of a program to change in present-tense when we're just trying to verify a type or object's LIKE-ness. Let-names introduced inside the mediant are not restricted, since there could be some useful sub-typing properties that capture the transfer of TOWEL.

Mediants are the primary way to programmatically prune overloads. The mediant operates on one overload at a time, as controlled by the AM. The name of the mediant, whether introduced by a let-supertonic, or a Tensah parameter subtonic, is the let-name by which the current overload candidate is accessed. Mediants communicate with the AM by tripping atonable conditions to control overload pruning in an application/domain specific manner. Candidates that survive this pruning process are considered LIKE the what mediant describes.

Named mediants cannot be overloaded, since there are no parameters, other than the implicit overload set. Names that overload a mediant - of any archetype - trip an unatonable condition.

32 Mediant Func

Figure 32.1: Mediant Func railroad diagrams.



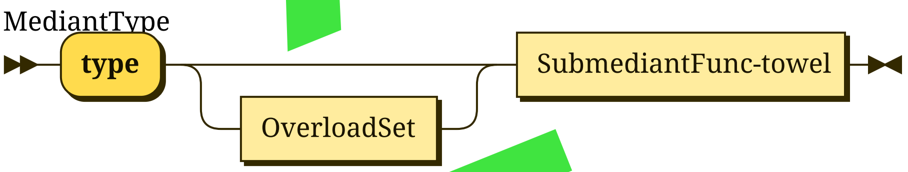
[a]

[a]

Subsection intentionally left to this one sentence.

33 Mediant Type

Figure 33.1: Mediant Type railroad diagram.



[a]

[a]

Subsection intentionally left to this one sentence.

Part VIII

Hyperspace

Table 12: *Foreign Function Interface keyverbs*

@@func
@@infx
@@tool
@@tune
@@type
@@utf8

TONAL code lives in normal space, which has TOWEL, name-search, and overload-pruning rules. Jumping into hyperspace allows access to the underlying universe of the AM and abandoning the rules of TONAL space. Hypertonics are the supertonic-equivalent in hyperspace[11].

It is too easy to end up writing entire blocks of code in the AM-native language, so to dissuade the creeping of bloat, hypertonics are limited to calling AM-native functions, and not simply writing arbitrary amounts of AM-native code. Hypertonics thinly wrap AM-native code - enough for the use of normal TONAL constructs to impart TONAL space semantics to the wrapped entities. This makes it easier to avoid errors trying to recreate TONAL-space semantics in AM-native code.

TONAL does not specify the AM-native language, although C99 is the machine model the TONAL AM is based on. C99 is chosen because the machine model is simple, mostly forwards compatible with later standards, and compilers are mature. The C99 features allowed in the implementation are those allowed by CompCert C compiler. This means any program generated by a TONAL AM can be compiled by CompCert C, even if other compilers are used for normal development. C99 can also be compiled to WASM, for browser based environments.

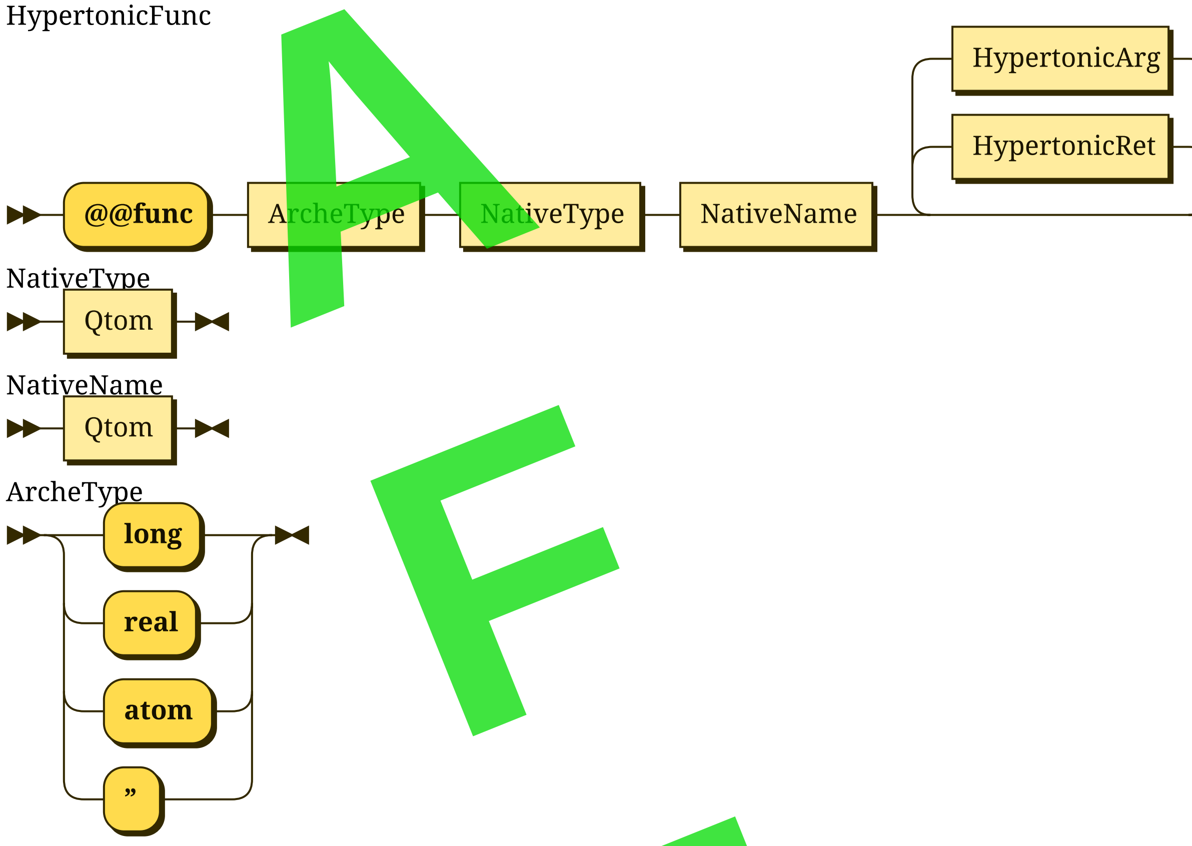
[11] They can also be used to tame lions.

TONAL does not use C++ because its RAII and template semantics are not a fitting match for TONAL's TOWEL and mediant semantics.

The lack of AM-native language specification allows compatible implementations on varied environments, like the Java, Javascript or .NET virtual machines, as well as compute modules, like OpenCL, SPIR-V, CUDA, etc.

34 Hypertonic Func

Figure 34.1: Hypertonic Func railroad diagrams.



HypertonicFunc

NativeType

NativeName

ArcheType

[a]

[a] ” is an empty string, signalling void

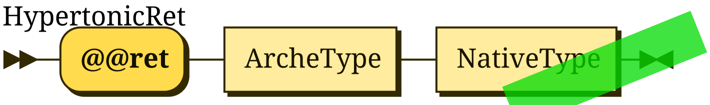
TONAL objects going into a native function's arguments needs to be converted from a TONAL archetype to the function's parameter's type. Native objects returned from a native function needs to be converted from the native return type to a TONAL archetype.

Archetypes are used to interface between native and TONAL code to enable the implementation of present-tense native functions, as with the hypertonics library.

The native function's name is the mangled-name used in native libraries for linking. This avoids the need of having to include native headers or import native modules in the hypertonic syntax. This means it is technically possible to interface with C++ code by calling the mangled name of a function. Care must be taken to understand the ownership semantics of the C++ function's parameters and return value.

It is preferred to use C-bindings for C++ code rather than C++ code directly, since the C-binding will presumably have accounted for C++ object lifetimes. C-bindings also have the advantage of being able to shepherd C++ exceptions into archetype compatible values for tripping conditions.

Figure 34.2: Hypertonic return value reference railroad diagram.



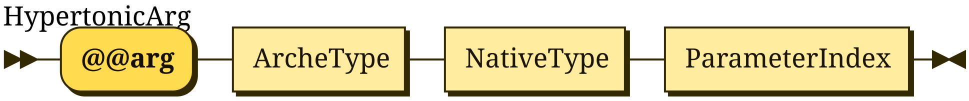
a

[a]

The TONAL principle is to always have funcs evaluate to a value, with errors and statuses reported via tripping a condition. Some native functions have output parameters instead of returning a value, with the return value being used to report error codes.

Native functions can participate in the TOWEL round-trip by constructing its output parameter in the place where the TONAL evaluation will reside, instead of copying or moving a value.

Figure 34.3: Hypertonic argument reference railroad diagram.



[a]

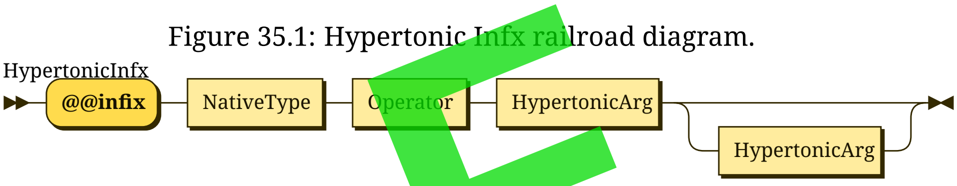
[a]

The wrapping TONAL func's arguments are referenced by position, starting at 0. The wrapping TONAL func may have pack arguments as long as no argument reference exceeds the number of arguments.

Pack arguments cannot be unpacked for the hypertonic, and are not usable for C variable argument lists. Arguments in a pack are referenced by position just like normal arguments.

35 Hypertonic Infx

Figure 35.1: Hypertonic Infx railroad diagram.



[a]

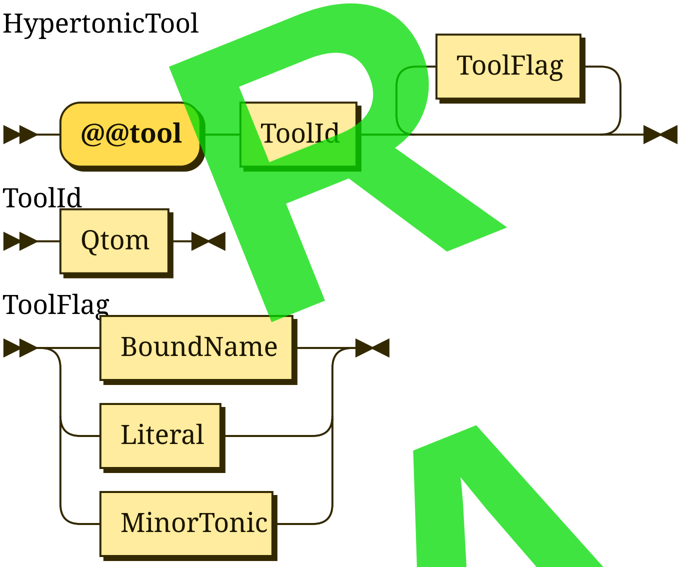
[a]

TONAL does not have the concept of mathematical operators: all mathematical operations are merely funcs. The infx-hypertonic avoids having to write AM-native shim functions that use operators.

Prefix unary operators are chosen if only one hypertonic argument is provided. There is no support for postfix operators.

36 Hypertonic Tool

Figure 36.1: Hypertonic Tool railroad diagrams.



[a]

[a]

Some development processes are better left to dedicated tools. Even though TONAL is designed to be much simpler to parse, and the TONAL collection provides parsers, it isn't very productive to expect to programmers to have to fully parse a program just to gather a small bit of statistics. Too much analysis can also slow build times and therefore rapid prototyping, so it's more productive to do extra analysis on the side every so often, instead of every build.

Rather than simply dumping out compiler internals like incomplete syntax trees or other data structures, the tool-hypertonic enables programmers to dump out the exact data they need in the exact format they need. No need to dig through documentation of some obscure file-format. Any present-tense value can be passed along to the tool.

Common uses for external tools include: documentation/diagram generators; project-specific library/interface usage and pitfall analysers; correctness provers; deadlock finders; callgraph summarizers; code generators for other languages; translation cataloging; and many more.

The external tool is specified by some ID, rather than by an executable's actual name. It avoids complicating the TONAL AM and build processes by requiring that command-line options be supplied for search paths, etc. It also enables a project to be more flexible, such as using the same ID to run a tool with different analysis options, different versions of the tool, and of course different tools. Altogether.

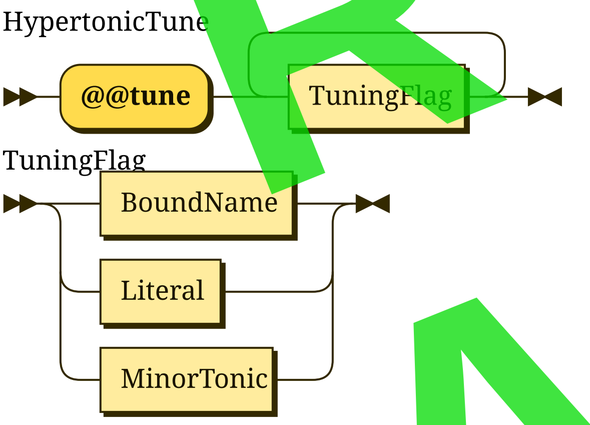
Using a tool ID, rather than an executable name or command line, reduces the need to change the hypertonic, causing recompiles and possibly even cause version control churn.

Tools are not applied in present-tense. The tool-hypertonic merely outputs the tool ID and the values provided. The tools are applied on the tool-hypertonic output when the project's chosen build system runs them.

One of TONAL's paradigms is to avoid a mish-mash of languages - everything is described by one source language - to make it easy to manage a project with minimal mental context-switching, and textual traceability of names. The tool-hypertonic breaks this paradigm, but it is acceptable in this case precisely because it explicitly bridges TONAL source with tool usage which maintains textual traceability.

37 Hypertonic Tune

Figure 37.1: Hypertonic Tune railroad diagrams.



[a]

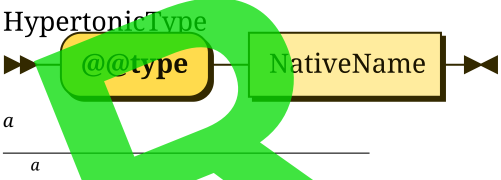
[a]

Music exists on a score, but needs to be played on an instrument to be heard. Instruments need to be tuned to the correct keys for the music to be played as intended. Some instruments can have alternate tunings to allow music to be written for unconventional key signatures.

Tunings do not modify the behaviour of a TONAL program. They are used to tweak the generated executable for time or space performance. Some tweaks can include: AM-level optimizations, such as data alignments; inlining, such as present-tense pruning vs future-tense switching and dummy arguments or variadic arguments; consolidation; telemetry code; machine and platform specific generation, such as hypertonic AM-language, operating system versions; execution hints, such as likely branches.

38 Hypertonic Type

Figure 38.1: Hypertonic Type railroad diagram.

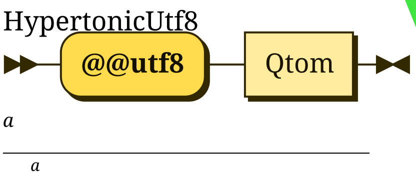


When referring to C struct or enum types, the native name must be preceded by struct or enum, just like in C, when not using typedefs. For other AM-languages, the respective language rules are followed also.

C does not have the concept of member functions like in C++, so func-hypertonics enclosed by a type that is defined by a type-hypertonic must pass the @object to the native func explicitly. C structs do have data members, so TONAL types that are hypertonic structs can only have enclosed objects whose let-names are exactly the same as the C struct's members, and have the same hypertonic type.

39 Hypertonic Utf8

Figure 39.1: Hypertonic Utf8 railroad diagram.



Writing arbitrary native code in TONAL defeats the purpose of TONAL-space. For TONAL implementations, however, features requiring native language, operating system, or machine support may require the ability write inline native code.

This capability is reserved for TONAL implementations only, and use outside of TONAL provided libraries trips an unatonable condition.

Part IX

Libraries

Powerful systems programming languages does not do everything, but anything can be done with it. Languages like LISP and C++ achieve this by being a language to implement libraries.

40 HYpertonic Present-tense Elements

Languages cannot be turtles all the way down. The **Hy**pertonic **P**resent-tense **E**lements library is the last turtle, standing on top of the AM. The AM can theoretically be implemented in any language, and the HYPE library does the dirty work of translating TONAL's semantics onto the AM implementation language.

Implementing as much of the language as a library simplifies the compiler development and maintenance process. The compiler then only needs to understand the scale-degrees, then generate code, then execute the generated code, in present-tense.

Every item in the HYPE library forms the backbone of the present-tense programming. They are the archetypes and the conditions tripped by the AM if there are ambiguities, inconsistencies, or errors, in the program's definition. They do not have a fixed machine representation since they are present-tense and are discarded from the final executable, even if a type is derived from them.

41 Grand Unified Toolkit of Generally Useful Things

The GUT builds on top of the HYPE library and provides concrete representations of the archetypes. Most things in the GUT library are usable in present-tense. The GUT library provides the guts of the language and every conceivable application.

The GUT identifies common tendencies in data structures and algorithms that occur in any interesting program. Such structures and algorithms are so common that TONAL and GUT should also utilize them itself. Dogfooding brings to light any gaps in generality or concrete implementation.

The toolkit design encourages the definition of highly modular and extremely composable units. Makers use whatever tools are available to put systems together in any way imaginable. The more generalized and self-contained the design of these tools, the greater number of combinations is achievable, without having to be explicitly designed into the library.

This stands in opposition to framework designs. A framework is a mostly complete thing. It is a black box with holes in which programmers plug-in their own customizations, but the overall activity of the framework is unchanged. They are, by design, not really composable with other frameworks, and tend to be hard to use components as individual modules.

Memory pool. Cache awareness. Copy-on-Write. Persistency. State machines. Parsers. Generators. Actors. Serialization. Maths (algebraics, numerics).

41.1 TONALly Legible Data Representation

The experience of Javascript Object Notation shows a great need to have a succint, but readable, data representation format within a language itself. The experience of JSON also shows that there is a lot of value in XML's S-expression design.

TONALly **L**egible **D**ata **R**epresentation is the native format for all kinds of transport in TONAL. Even the TONAL compiler and other parts of the ecosystem would use TLDR as a communication and storage format. TLDR is natively supported even as a present-tense TONAL source when included as such.

41.2 System Entities and Hypertonics Repository

Useful programs do things that have side-effects, which means invoking some platform functionality: future-tense. The platform could be an operating system, or it could be the direct hardware, or it could be a simulator or emulation of some sort.

TONAL is a cross-platform language, but platforms differ wildly. The aim of the SEHR library is to represent any platform's primitives faithfully within TONAL types. When a platform's semantics is faithfully exposed to TONAL, then the present-tense facilities of TONAL can be used to accurately reflect on specialized behaviour and produce high-quality generalized types. This includes any versioning differences on the platform side.

IO. Concurrency. Memory mapping. Signals. Events. Interrupts. Coprocessing. Dynamic linking.

41.3 Common Appliance Toolkit

Beyond data structures and algorithms, there are some larger-scale organizational principles that arise from time to time. A kitchen has utensils like knives, forks, spoons, pots, pans, spatula, but also appliances like toasters, fridges, blenders, stoves, ovens, rice cookers. Likewise productivity applications have a few commonly used subsystems that perform more complex, less formal, functionality.

CRUD. Scheduler. Edit history. GUI. Client/server. Peer. Settings. Workflow. Entity-Component. Command history. Notifications.

41.4 Microcosm of Basic Input and Output Modular Executables

The UNIX model showcases the power of single-purpose command-line tools that takes input of one format and churns output in another format.

The major problem with those command-line tools is that they were designed only with command-line in mind, so they are only usable from shell-scripts. The next major problem is portability due to a lack of standard surrounding syntax of arguments and input and output formats.

The MicroBIOME is a project to utilize the full power of TONAL libraries - GUT, TLDR, SEHR and CAT - to formally specify a suite of scripting tools as a library of code. There are high level processes common to many programs, such as pipelines of data processing, dependency management, continuous integration, platform management, version control, scripting environments, etc. Making them available as code gives TONAL the power of scripting, while giving tooling the power of verification.

41.5 Future Extension Experimental Library

The experience of C++ shows the benefits of disciplined, considered, succession plan. Good standards need good implementations. Standards don't get implemented. No one commits to implementations without a standard. The FEEL library is the fertile ground for which serious experiments in new types and libraries get official support, with an eye towards eventual standardization.

There will never be two libraries doing the same thing, but there will be constant competition to supplant lesser-quality libraries with better ones.

Two libraries may approach the same problem with different strategies. Effort should be made to analyse all approaches and whether there can be a unified approach and/or a common core that would be useful outside of the two approaches. The unified approach should not be much more complex than either individual libraries.

Standard - must have backwards compatibility tests and migration tools for ABI breakages.

Proposal - must have conformance tests.

Experimental - go wild.

42 Standards

Software must ultimately be interoperable with the real world, whether they be people, machines, or software, new and old. It is an important goal of TONAL to have official libraries for internationally recognized standards, or lacking standards, the closest things to standards coming from various industry consortiums.

The lack of libraries that check for errors in standards application in present-tense has cost hundreds of millions of dollars in accidents. Standards libraries should be designed in such a way as to map to the standards themselves as directly as possible. This further reduce errors by reducing friction between the library's usage and the standard's text.

All libraries should document the source of its information.

These libraries inform the general design of GUT libraries. For the purposes of traceability, every generally-useful component of a standard should be developed in the respective STD library, and then identified as being generally-useful, and migrated or design-merged into a GUT library.

Not wrappers over sockets or network stacks, but a replacement. Drivers. Operating Systems. Packet sniffers. Diagnostics. Simulators.

42.1 SI units (International Bureau of Weights and Measures)

Figure 42.1: International System of Units library layout.

Type	Description	Specification	Version
SI.units	The symbols that make up the SI units, and prefixes. Symbolic unit dimensional analysis.	2022	9th edition
SI.constants	The seven defining constants, and their symbols.	2019	9th edition
SI.compatibility	Unofficial units within SI usage.	2019	9th edition
SI.USA	Units and constants for converting to the SI system.	2019	
SI.UK		2019	

The BIPM brochure for the SI units contains rules about displaying as text that should be implemented.

42.2 ISO standards

Figure 42.2: ISO Standards library layout.

Type	Description	Specification	Version
ISO.quantities	International System of Quantities. Formatting rules. Symbolic unit dimensional analysis.	ISO-80000	2022
ISO.datetime	The one true date/time format.	ISO-8601	2022
ISO.languages	2 and 3 letter language codes.	ISO-639	2010
ISO.countries	Country codes and subdivisions.	ISO-3166	2020
ISO.currencies	Alphabetic and numeric currency codes.	ISO-4217	2015
ISO.paper	A and B Series of paper sizes.	ISO-216	2007
	Raw A sizes.	ISO-217	2013
	C Series - envelope sizes. Withdrawn, but implement for remaining usage.	ISO-269	1985
	Hole punch.	ISO-838	1974
ISO.Unicode	Only character set supported by TONAL.	ISO-10646	2020
ISO.Z	Formal verification support.	ISO-13568	2007
ISO.IS??	ISAN. Audiovisual Number.	ISO-15706	2008
	ISBN. Book Number.	ISO-2108	2017
	ISIL. Identifier for Libraries. Libraries, archives, museums.	ISO-15511	2019
	ISMN. Music Number. Printed music.	ISO-10957	2021
	ISNI. Name Identifier. Contributors to media.	ISO-27729	2013
	ISRC. Recording Code. Sound and music video recordings.	ISO-3901	2019
	ISSN. Serial Number. For serial publications.	ISO-3297	2022
	ISTC. Text Code. Text-based. Withdrawn, but implement for remaining usage.	ISO-21047	2009
	ISWC. Musical Work Code. For collections.	ISO-15707	2022
ISO.QR	QR codes.	ISO-18004	2015
	Micro QR codes.	ISO-23941	2022

42.3 Ecma standards

Figure 42.3: Ecma Standards library layout.

Type	Description	Specification	Version
Ecma.terminal	Control character sequences for VT-100, VT-220, VT-420 terminals and emulators.	ECMA-48	5th edition
Ecma.JSON	Parsing only.	ECMA-404	2nd edition
Ecma.Script	EcmaScript. Strict, no polyfills.	ECMA-262	13th edition
Ecma.CD	Read and generate CD images.	ECMA-168	2nd edition
	Universal Disk Format.	ECMA-167	3rd edition

42.4 IEEE standards

Figure 42.4: IEEE Standards library layout.

Type	Description	Specification	Version
IEEE.float	binary16 to binary256. decimal 32 to decimal128. Recommended functions. Textual conversions.	IEEE-754	2019
IEEE.interval	Floating point interval arithmetic. Error bounds and reliable testing.	IEEE-1788	2015
IEEE.Ethernet	Low level networking.	IEEE-802.3	2022
IEEE.WiFi		IEEE-802.11	2022
IEEE.JTAG	Hardware debugging.	IEEE-1149.1	2013
IEEE.VHDL	Programmatically generate hardware from TONAL.	IEEE-1076	2019
IEEE.SystemVerilog		IEEE-1800	2017
IEEE.PSL	Parse to generate bindings to hardware, or emulations. Property Specification Language. Hardware verification.	IEEE-1850	2010
IEEE.POSIX	Issue 7.	IEEE-1003.1	2017
IEEE.RTOS	µT-Kernel.	IEEE-2050	2018

POSIX and RTOS library may range from simple interface bridging, to API compatibility layer for emulation or testing purposes.

42.5 IETF

42.5.1 Internet standards

Figure 42.5: IETF Internet Standards library layout.

Type	Description	Specification	Version
IETF.STD.UTF-8	UTF-8 is the only character encoding used in TONAL. UTF-8 everywhere.	STD-63	
IETF.STD.PPP	A reasonable modern network stack for implementing services, streams, and clients, at all levels of the internet.	STD-51	
IETF.STD.IP		STD-5 STD-8[6-9]	4 6
IETF.STD.UDP		STD-6	
IETF.STD.TCP		STD-7	
IETF.STD.RTP		STD-64	
IETF.STD.DNS		STD-13	
IETF.STD.HTTP		STD-9[7-9]	1.1
IETF.STD.POP		STD-53	3

42.5.2 Proposed standards

Figure 42.6: IETF Proposed Standards library layout.

Type	Description	Specification	Version
IETF.RFC.SOCKS	Proxy protocol. GSS-API authentication.	RFC-192[8-9] RFC-1961	5
IETF.RFC.MIME	Classify file types generally.	RFC-204[5-7,9] RFC-4289	
IETF.RFC.DHCP	IP assignment.	RFC-2131 RFC-8415	4 6
IETF.RFC.SLP	Service Location Protocol.	RFC-2608	2
IETF.RFC.IGMP	IP multicast groups.	RFC-3376	3
IETF.RFC.Kerberos	Secure authentication.	RFC-4120	5
IETF.RFC.UUID	Name mangling that's reasonably quick and unique for all kinds of linking.	RFC-4122	
IETF.RFC.SSH	Secure tunnelling.	RFC-425[0-4]	
IETF.RFC.BGP	Border Gateway Protocol for Classless Inter-Domain Routing	RFC-4271	4
IETF.RFC.LDAP	Organizational directory services.	RFC-4510	
IETF.RFC.Base64	Base 16, 32, and 64 encoding. Prefer URL safe.	RFC-4648	
IETF.RFC.PGP	OpenPGP Message Format.	RFC-4884	
IETF.RFC.WebDAV	Document editing over HTTP.	RFC-4918	
IETF.RFC.SMTP	Mail messaging.	RFC-5321	
IETF.RFC.message	Internet Message Format.	RFC-5322	
IETF.RFC.XMPP	Open real-time messaging.	RFC-6120	
IETF.RFC.RPC	Basis for NFS.	RFC-5531	2
IETF.RFC.NFS	Transparently access file servers.	RFC-786[2-3]	4.2
IETF.RFC.TLS	Encrypted connections.	RFC-8446	1.3
IETF.RFC.IMAP	Mail management.	RFC-9051	4
IETF.RFC.NTP	Accurate time keeping.	RFC-9109	4
IETF.RFC.HTTP	Future compatibility with the WWW.	RFC-9113 RFC-9114	2 3
IETF.RFC.SCTP	Stream Control Transmission Protocol.	RFC-9260	

42.5.3 Informational

Figure 42.7: IETF Informational RFCs library layout.

Type	Description	Specification	Version
IETF.RFC.ZLIB	Compressed file format.	RFC-1950	3.3
IETF.RFC.DEFLATE	Common compression scheme.	RFC-1951	1.3
IETF.RFC.GZIP	Common compressed file format.	RFC-1952	4.3
IETF.RFC.PNG	Common image compression.	RFC-2083	
IETF.RFC.UTF-16	Less common unicode encoding.	RFC-2781	
IETF.RFC.CSV	Quick and dirty tabular data.	RFC-4180	

Informational RFCs, mostly about file formats and data encodings, implemented to consume files in those formats and encodings. For producing, TONAL will settle on newer, more standard, more performant formats and encodings.

42.6 ITU recommendations

Figure 42.8: ITU Recommendations library layout.

Type	Description	Specification	Version
ITU.UTC	Definition of standard time.	TF.460	2002
ITU.ASN.1	Specify binary format layouts.	X.68[0-3] X.69[0-7]	2021 2021
ITU.UHDTV	8K and 4K TV standard.	BT.2020	2015

42.7 NIST standards

Figure 42.9: NIST Standards library layout.

Type	Description	Specification	Version
NIST.constants	Constants and units for physical properties, with error bounds.	CODATA-2018	2021
NIST.AES	Strong symmetric encryption.	FIPS-197	
NIST.DSS	Digital signatures.	FIPS-186	5
NIST.HMAC	Hash authentication.	FIPS-198	1
NIST.SHA	SHA-3	FIPS-202	
	SHA-224 to SHA-512	FIPS-180	4
	SHA-1	FIPS-180	4

42.8 OASIS standards

Figure 42.10: OASIS Standards library layout.

Type	Description	Version
OASIS.RELAX-NG	Schema language for a lot of standards.	3
OASIS.OpenDocument	Office productivity authoring and interchange.	1.3
OASIS.DocBook	Technical manual authoring.	5.1
OASIS.AMQP	Enterprise messaging.	1.0
OASIS.MQTT	Embedded device messaging.	5
OASIS.PKCS	Cryptographic Message Syntax.	7
	Cryptographic hardware API.	11
OASIS.SAML	Single Sign On.	2.0

42.9 W3C recommendations

Figure 42.11: W3C Recommendations library layout.

Type	Description	Specification	Version
W3C.XML	Hierarchical document structure and interchange.	1.0	5th edition
	Namespaces	1.0	2nd edition
	XInclude	1.0	2nd edition
	Information Set		2nd edition
	id		
	Base		2nd edition
W3C.XQuery	Hierarchical database querying.	3.1	
	XQueryX	3.1	
	XPath	3.1	
	XQuery	3.1	
	XQuery and XPath Data Model	3.1	
	XPath and XQuery Functions and Operators	3.1	
	XSLT and XQuery Serialization	3.1	
	XQuery and XPath Full Text	3.0	
	XQuery Update Facility	1.0	
W3C.WASM	Core	2.0	2023-01-18
	JavaScript Interface	2.0	2022-04-19
	Web API	2.0	2022-04-19
	WASI	Snapshot.	
W3C.RDF	Knowledge tagging.	1.1	
W3C.WOFF	Web fonts.	2.0	
W3C.CSS	Decoupled visual styling.	3	
W3C.SVG	Vector graphics.	2	
W3C.WebCGM	Technical vector graphics.	2.1	
W3C.SCXML	State machine language.	1	
W3C.MathML	Mathematical formula representation and formatting.	3.0	2nd edition
W3C.EPUB	Electronic books.	3.2	

42.10 WHATWG Living Standards

Figure 42.12: WHATWG Standards library layout.

Type	Description
WHATWG.WebIDL	Programmatic description of interfaces.
WHATWG.URL	Web address.
WHATWG.DOM	Document model.
WHATWG.HTML	Document semantics and presentation.
WHATWG.WebSockets	Efficient persistent web connections.

42.11 Industry consortiums

42.11.1 Khronos

Figure 42.13: Khronos Standards library layout.

Type	Description	Version
Khronos.SPIR-V	Shader language intermediate representation.	1.6
Khronos.OpenCL	GPU compute.	3.0
Khronos.Vulkan	Low level 3D graphics. SC. For high reliability vehicles.	1.3 1.0
Khronos.WebGL	OpenGL profile for web.	2.0
Khronos.EGL	OpenGL and native integration.	1.5
Khronos.COLLADA	3D modelling assets.	1.5.0
Khronos.glTF	3D asset transfer.	2.0
Khronos.KTX	Texture container for distribution.	2.0

42.11.2 Bluetooth

Figure 42.14: Bluetooth Standards library layout.

Type	Description	Version
Bluetooth.Core	Short-range wireless peripheral communications.	5.3

42.11.3 HDMI

Figure 42.15: HDMI Standards library layout.

Type	Description	Version
HDMI	Audio/Video transmission.	2.1

42.11.4 USB

Figure 42.16: USB Standards library layout.

Type	Description	Version
USB	Peripheral communications.	4 3.2 2.0

42.11.5 JEITA

Figure 42.17: JEITA Standards library layout.

Type	Description	Version
JEITA.Exif	Camera media format.	2.32

42.11.6 Xiph.Org

Figure 42.18: Xiph library layout.

Type	Description	Specification	Version
Xiph.Ogg	Audio/Video container format.	RFC-3533	0
Xiph.Vorbis	Audio lossy compression.		I
Xiph.Opus	Audio lossy compression supplanting Vorbis.	RFC-6716	
Xiph.FLAC	Audio lossless compression.		1.4.2

42.11.7 Matroska

Figure 42.19: Matroska library layout.

Type	Description	Specification	Version
Matroska.EBML	Binary XML.	RFC-8794	
Matroska.MKV	Audio/Video container format.		1.63 WebM

42.11.8 AOMedia

Figure 42.20: AOMedia Standards library layout.

Type	Description	Version
AOMedia.AV1	High performance video compression.	1.0.0
AOMedia.AVIF	Image format based on AV1.	1.1.0

42.12 Vendors

42.12.1 Linux

Figure 42.21: Linux library layout.

Type	Description	Version
Linux.ELF	Executables.	
Linux.DWARF	Debugging information.	5
Linux.io_uring	Asynchronous IO.	
Linux.epoll	Event notification.	

42.12.2 BSD

Figure 42.22: BSD library layout.

Type	Description
BSD.kqueue	Event notification.

42.12.3 Microsoft

Figure 42.23: Microsoft library layout.

Type	Description	Specification	Version
Microsoft.IOCP	Asynchronous IO.		
Microsoft.fiber	Lightweight concurrency.		
Microsoft.OpenType	Common font format.		1.9
Microsoft.BMP	Common lossless raster format.		
Microsoft.PE	Executables.		2022-24-06
Microsoft.OOXML	Read MS Office documents.	ECMA-376	5th edition
Microsoft.XPS	Read printable documents.	ECMA-388	1st edition

42.12.4 MacOS

Figure 42.24: MacOS library layout.

Type	Description
MacOS.Mach-O	Executables.
MacOS.dispatch	Lightweight concurrency.

42.12.5 OpenMP

Figure 42.25: OpenMP library layout.

Type	Description	Version
OpenMP	Parallel annotations	5.2

42.12.6 Adobe

Figure 42.26: Adobe library layout.

Type	Description	Specification	Version
Adobe.Postscript	Document printing language.		3
Adobe.PDF	-/A, -/E, -/R, -/X, -/UA, -/VCR, -/VT. Archiving, engineering, raster, printing, accessibility.	ISO-32000	2.0
	ECMAScript for PDF.	ISO-21757	2.0
Adobe.XFDF	XML Forms Data Format.	ISO-19444	3.0
Adobe.XMP	eXtensible Metadata Platform	ISO-16684	2021

42.12.7 ILM

Figure 42.27: ILM library layout.

Type	Description	Version
ILM.OpenEXR	Deep raster images.	3.1

42.12.8 QOI

Figure 42.28: QOI library layout.

Type	Description	Version
QOI	Performant lossless image compression.	1.0

42.12.9 7z

Figure 42.29: 7z library layout.

Type	Description
7z	Archive format.
7z.LZMA	Archive compression algorithm.

42.12.10 BitTorrent

Figure 42.30: BitTorrent library layout.

Type	Description	Specification
BitTorrent	Peer-to-peer file mirroring protocol.	BEP-3
BitTorrent.DHT	Distributed Hash Table.	BEP-5
BitTorrent.tracker	UDP tracker.	BEP-15
BitTorrent.uTP	UDP-base protocol.	BEP-29

42.12.11 SQLite

Figure 42.31: SQLite library layout.

Type	Description	Version
SQLite	In-memory database and interchange.	3

42.12.12 Princeton

Figure 42.32: Princeton library layout.

Type	Description	Version
Princeton.WordNet	Dictionary with API.	3.0

Index