J0001

Jvav

Language Specification

Working Draft

REVISION 0
21 SEPTEMBER 2026

This is an incomplete working draft. Pending design questions are identified in the text and collected in Annex D.

Contents

Foreword [foreword]

[1] Jvav is a general-purpose programming language whose design combines functional semantics with imperative expression. Its central distinction is between computing a description of a state change and applying that description to existing state. Ordinary control flow remains available; read access, mutation authority, state transitions, runtime deferral, and contextual dependencies are expressed separately.

[2] This document is the first numbered working draft in the Jvav document series. Its document number is **J0001**, and its revision is **0**. The letter **J** denotes a Jvav document. A subsequent revision of this document may be identified as **J0001R1**, where **R** denotes revision. Document numbers do not designate language releases.

[3] The principal design source is *Jvav Language Design Draft 0.2*, dated 2026-09-21. Earlier Jvav drafts and the historical Mamba implementation supply background for declarations, expressions, and control flow. Where their direction differs, the 0.2 design is the authority for this revision. In particular, `expression T` is not a source-language type modifier.

[4] The organization uses conventions familiar from programming-language working drafts: numbered clauses, stable bracketed clause labels, numbered paragraphs, requirements expressed with *shall*, and a separation between normative wording and informative material. These conventions do not imply adoption by a standards body or compatibility with C++.

[5] This is an incomplete language specification. Established design requirements are stated precisely where the sources permit; unresolved choices remain explicitly **Pending**. Annex D is the authoritative register of those choices. Neither an example nor an implementation sketch resolves a pending issue. No claim is made that an existing compiler implements the mechanisms described here.

[6] The draft is intended for language designers, implementers, library authors, and reviewers. It specifies the semantic boundaries that later syntax, type checking, runtime behavior, and optimization rules must respect. The distinction between a complete specification and a precise specification of an incomplete design is intentional.

1 Scope [scope]

[1] This document specifies the established semantic core of Jvav and records the remaining requirements for a complete language definition. It covers values and computations, function interfaces, capability modifiers, purity, state-transition construction and application, explicit deferral, and contextual dependencies. It also records the conventional language foundations on which those mechanisms depend.

[2] Jvav permits an algorithm to be written with local declarations, functions, classes, blocks, conditional execution, and iteration. These syntactic forms do not, by themselves, determine whether the algorithm has observable effects. Effects are determined by the operations performed and the capabilities exercised.

[3] The state-transition model separates computation from mutation. A transition describes a change to a designated state root. Constructing that description does not apply the change. Application occurs at an explicit boundary and requires appropriate authority over the target.

[4] The value model separates semantic values from their physical representation. An ordinary parameter of type `T` denotes a `T`; it does not require an independent parameter storage slot, a copy operation, or an intermediate aggregate allocation. Implementation choices shall preserve the language-level meaning of the program.

[5] This revision does not define a complete standard library, a portable binary interface, a complete ownership system, an exception mechanism, or a concurrent memory model. It does not specify all lexical productions, all type combinations, or a complete executable grammar. These omissions are tracked in [open]. They shall not be read as grants of unspecified or undefined behavior.

Note 1: LLVM, JVM bytecode, and MSIL are implementation directions named in the design sources. They are not required execution environments, and their object models do not define Jvav semantics. See [platform].

2 References [references]

2.1 Normative references [references.normative]

[1] This revision incorporates no external specification by normative reference. Requirements in another language standard, a compiler implementation, or a backend specification do not become Jvav requirements merely because that material is mentioned here.

[2] The language's future dependence on character-set standards, numerical formats, platform interfaces, or library specifications is **Pending**. A later revision shall identify the applicable edition when incorporating an external specification.

2.2 Design sources [references.sources]

[1] The design sources and their precedence are identified in [sources]. The supplied 0.2 document determines the current semantic direction. Historical material is used to establish continuity and identify unresolved discrepancies, not to restore mechanisms superseded by that direction.

[2] The C++ working draft N4950 is an editorial reference for organization and navigation only. No C++ object model, value-category system, overload rule, undefined-behavior rule, or library requirement is imported by this document.

3 Terms and definitions [terms]

3.1 Program entities [terms.entities]

[1] **Entity:** a language-level item introduced or denoted by a program, including a value, object, function, parameter, variable, or class member.

[2] **Name:** an identifier used to denote an entity. A declaration introduces a name into a scope.

[3] **Value:** the semantic result of a computation or the semantic content associated with an object at a particular state. A value is not necessarily represented by an independently allocated object.

[4] **Object:** an entity whose state can be observed under the access and lifetime rules applicable to it. The complete relationship between objects, storage, identity, and ownership is **Pending**, O04.

[5] **Scope:** a region in which a declaration participates in name lookup. A block and a function introduce scopes. Cross-unit lookup and detailed shadowing rules are **Pending**, O05.

[6] **Compilation unit:** a unit of source processed as a unit of translation. Its outermost scope is its global scope. This does not imply that independently translated units share a mutable global environment.

3.2 Computation and observation [terms.computation]

[1] **Computation:** an evaluation that can produce a value, determine control flow, or perform effects.

[2] **Observable effect:** an externally observable action or state change of a computation, including mutation of existing externally observable state or interaction with an external environment. A complete taxonomy of effects is **Pending**, O08.

[3] **Declared dependency:** an input whose presence is expressed in the computation's interface, including an ordinary argument, a permitted borrow, a transition input, or a context parameter.

[4] **Pure computation:** a computation whose result is determined by its declared dependencies and which produces no externally observable effects or unauthorized mutation. Purity does not by itself establish termination or the absence of failure.

[5] **Materialization:** an implementation step that gives a semantic value or transition a concrete representation suitable for storage or a particular execution boundary. Materialization is not, by itself, a source-language operation.

[6] **Representation transparency:** the requirement that implementation choices concerning materialization and propagation of an ordinary value do not introduce an observable distinction at the source level.

3.3 Access and lifetime [terms.access]

[1] **Capability:** authority to perform a specified category of operation on an entity. Read access and mutation authority are distinct capabilities.

[2] **Borrow:** access to an existing entity without transferring ownership of that entity.

[3] **View:** a read-only, non-owning, non-escaping borrow expressed by `view T`.

[4] **Escape:** retention or transfer of a borrow beyond the lifetime or usage region permitted for that borrow. Returning it, storing it in longer-lived state, or retaining it in a deferred computation can cause an escape. The complete escape analysis is **Pending**, O09.

[5] **Mutable capability:** authority, expressed by `mut T`, to directly modify an existing `T` subject to the applicable effect, aliasing, and lifetime rules.

3.4 State transitions [terms.transition]

[1] **State root:** the designated object or state aggregate to which a transition computation is related. A root is not identified merely by its static type.

[2] **Old state:** the logical input state against which a transition computation reads its root and determines its changes.

[3] **Transition:** a description of changes to a state root, expressed by `transition T` for a root of type `T`. It is not an ordinary replacement value of type `T`.

[4] **Write set:** the set of root-relative state components for which a transition supplies new values.

[5] **Read set:** the set of state components consulted to compute a transition. An implementation can conservatively approximate this set.

[6] **Application or commit:** the operation that applies a transition's changes to its target under the applicable capability and validity rules. The two terms name the same semantic boundary in this draft.

[7] **Lineage:** the relationship between a transition, the root from which it was computed, and the relevant history of that root. The representation and validation of lineage are **Pending**, O12.

[8] **Stale transition:** a transition for which intervening changes or lifetime events can invalidate the relationship between its assumed input state and a prospective application target.

3.5 Deferred and contextual computation [terms.deferred]

[1] **Deferred computation:** a computation whose evaluation is explicitly postponed by a source-language mechanism, as distinguished from an unobservable implementation decision to delay materialization.

[2] **Demand:** an operation that requests the result of a deferred computation. The source syntax and complete set of operations that constitute demand are **Pending**, O17.

[3] **Memoization:** retention of a computed result for subsequent demands on the same deferred computation. Memoization is a design direction for `defer T`; its final requirements are **Pending**, O16.

[4] **Context dependency:** a typed dependency declared using `context T` that may be supplied by the calling environment rather than written as an argument at every call site.

4 General principles [intro]

4.1 Document status and compliance [intro.compliance]

[1] The requirements in the normative clauses describe the established design of this revision. An implementation claiming support for a mechanism specified here shall satisfy that mechanism's applicable requirements and shall identify any extensions or unresolved choices on which its behavior depends.

[2] This document is not sufficient for a claim of complete Jvav language conformance. Such a claim requires, among other things, complete syntax, type rules, evaluation rules, lifetime rules, and a diagnostic contract. Those requirements remain **Pending**, O23.

[3] The use of *Pending* identifies a missing language decision. It does not mean *implementation-defined*, *unspecified*, or *undefined*. This revision does not silently substitute any of those categories for an unfinished rule.

[4] An implementation can experiment with pending mechanisms. An experimental choice is not thereby a normative rule of Jvav. Implementers should record such choices so that programs and test results can be interpreted against the correct language revision.

4.2 Requirements and informative material [intro.structure]

[1] The word *shall* expresses a requirement; *shall not* expresses a prohibition. *May* expresses permission, and *can* describes a possibility. Definitions and unqualified descriptions in the normative clauses contribute to those clauses' meaning.

[2] Foreword, notes, examples, and annexes explicitly identified as informative are not normative. An example illustrates a particular requirement under stated assumptions. It does not settle unspecified syntax or grant capabilities absent from the surrounding rules.

[3] Bracketed labels, such as [trans.old], identify clauses independently of their position. Paragraph numbering restarts within each clause or subclause. A review can therefore cite a label and paragraph number rather than a page number that changes during editing.

[4] Normative wording states the established constraints of an incomplete design. The issue register records what is missing. A pending detail shall not weaken an established constraint; for example, incomplete lifetime syntax does not permit a `view` to outlive its permitted borrow.

4.3 Syntax notation [intro.notation]

[1] Code is set in a monospaced face. In the grammar fragments of Annex A, quoted text denotes a terminal, `|` separates alternatives, square brackets denote an optional element, and braces denote repetition. These meta-characters are grammar notation unless quoted.

[2] The grammar fragments identify the shapes used by this draft's examples. They are informative and intentionally incomplete. They are not an acceptance grammar for a compiler. Names such as `expression`, `type-name`, and `statement` designate categories whose full productions remain to be specified.

[3] In semantic equations, `S` denotes a logical state, `r` a root identity, `p` a root-relative field path, `W` a finite map from field paths to replacement values, and `t` a transition. These symbols are specification notation, not Jvav identifiers with privileged meaning.

4.4 Abstract semantics and implementation freedom [intro.abstract]

[1] A Jvav program has semantic dependencies and observable behavior independently of whether its implementation uses registers, stack storage, heap objects, graph nodes, thunks, or an intermediate representation. An implementation shall preserve the requirements applicable to that behavior.

[2] An implementation need not reproduce a source program's apparent sequence of allocations or copies when those operations are not required by the language semantics. It shall not use representation changes to introduce additional observable effects, bypass a capability restriction, or change a transition's old-state meaning.

[3] An optimization justified by purity is subject to every other applicable semantic requirement. Determinism and absence of external mutation do not alone prove that a computation can be deleted, duplicated, or moved across a failure, nontermination, or lifetime boundary. The complete rules for those observations are **Pending**, O24.

Note 1: Zero-cost abstraction is a design objective, not a promise that every program has no allocations or that every backend performs every optimization.

4.5 Translation model [intro.translation]

[1] Translation determines the structure of the program, resolves declarations and types, checks the applicable semantic restrictions, and produces an executable representation. This description specifies responsibilities, not a mandatory sequence of compiler passes.

[2] Deferring the parsing of a function body does not make the body semantically opaque when its contents are required for checking the program. Likewise, delaying value materialization does not remove a value's type, dependencies, or effect obligations.

[3] The use of a structural scanner, a stateless parser, HIR, MIR, or a particular code generator is an implementation choice. Informative implementation directions appear in [implementation].

5 Lexical and syntactic foundations [lex]

5.1 Source and tokens [lex.source]

[1] Source is analyzed into identifiers, keywords, literals, operators, and punctuators. These categories are inherited from the earlier Jvav design. The accepted source encoding, identifier character repertoire, normalization rules, and treatment of invalid encodings are **Pending**, O01.

[2] The complete keyword set and whether individual modifier words are reserved or contextual are **Pending**. The spelling `expression` is not a type modifier in this revision. An implementation shall not expose the removed `expression T` mechanism as part of this draft's ordinary parameter model.

[3] Whitespace, line endings, comments, and statement boundaries require a complete lexical contract. The earlier implementation makes some line-sensitive decisions, including return-operand parsing. Those decisions do not settle the current grammar; see O02.

5.2 Literal forms [lex.literal]

[1] The historical language includes number, string, and Boolean literals. `true` and `false` denote the two Boolean values. The current spelling of the Boolean type and the mapping between source literals and fundamental types are **Pending**, O03.

[2] The earlier draft describes binary, octal, decimal, and hexadecimal integer forms and quoted strings. Numeric prefixes, leading-zero interpretation, separators, suffixes, floating forms, string escapes, and invalid-literal diagnostics require reconciliation with the current design. They remain **Pending**, O01 and O03.

[3] Examples in this document use small decimal integers and the type names used in the 0.2 design, such as `Int` and `Float`. These examples do not fix bit widths, overflow behavior, floating-point rounding, or implicit conversions.

5.3 Surface syntax [lex.syntax]

[1] This document uses `fun`, `val`, and the modifier spellings of the 0.2 design. The historical implementation uses `let` for a read-only binding and `var` for a mutable binding. Whether `let` remains accepted, whether `val` replaces it, and how compatibility is handled are **Pending**, O02.

[2] The forms `transition(target) = expression`, `return { ... }` for a transition result, and `defer expression` are the working spellings used in this draft. Their semantic requirements are specified where established; their final grammatical integration remains **Pending**.

[3] The `for element in collection` form occurs in the current design. The earlier implementation uses a three-part loop. Their coexistence, exact delimiters, iterable protocol, and binding behavior are **Pending**, O02 and O06.

Note 1: Regularizing an example's spelling is not a language migration rule. Annex C identifies historical differences explicitly.

6 Program and state model [basic]

6.1 Declarations and scopes [basic.scope]

[1] A declaration introduces an entity and associates a name with it. The conventional foundations include declarations of variables, functions, and parameters, and the existence of class members. Detailed class, member, and generic declarations are **Pending**, O06.

[2] A compilation unit has an outermost scope. A function body and a block provide local scopes. The scopes of iteration bodies are entered as execution proceeds through the loop. Complete declaration visibility, shadowing, duplicate-name, and cross-unit rules are **Pending**, O05.

[3] The existence of a global scope is a name-organization property. It does not authorize a pure computation to read mutable global state. Such a read remains subject to [effect.pure] even when the name is visible.

6.2 Semantic values and storage [basic.value]

[1] An ordinary value of type `T` shall have the meaning prescribed for `T` regardless of its physical representation. A function parameter declared as `x: T` receives a semantic `T`. The declaration does not require the caller to construct an independently addressable temporary or the callee to copy that value into a separate slot.

[2] An implementation may represent an ordinary value by a register, an SSA value, existing storage, a caller-owned temporary, or a computation representation when doing so preserves the program's semantics. A source program shall not be given a means to distinguish those representations solely because one was chosen instead of another.

[3] This rule does not decide whether a particular nominal type has value identity, reference identity, copying, moving, ownership transfer, or user-defined resource operations. Those aspects of the object model are **Pending**, O04. Representation transparency shall respect whichever language-level operations apply; it does not erase their required effects.

Example 1: If `Vec3` is a simple aggregate with pure component arithmetic, a call that consumes only its components need not imply a physically allocated `Vec3`. If a future object model gives some construction an observable effect, that effect must also be preserved.

6.3 State and identity [basic.state]

[1] Existing mutable state is observed or changed through the capabilities permitted at the point of use. The ability to name or read an object does not confer the authority to change it.

[2] A state root identifies the subject of a transition. Two objects having the same static type are not, for that reason alone, interchangeable as transition targets. The complete identity and lineage rules are **Pending**, O11 and O12.

[3] A logical state can be described without specifying a physical snapshot. For example, the old state used by a transition can be preserved through read-before-write scheduling when the implementation can prove that the resulting observations are equivalent. The semantic requirement concerns the values read, not the allocation of a snapshot object.

6.4 Lifetimes and retained computations [basic.lifetime]

[1] Representation changes shall not extend a program's permitted access to an entity beyond that entity's valid lifetime. A borrow retained in a closure, transition representation, or deferred computation remains subject to the applicable non-escape and lifetime restrictions.

[2] A materialized transition may outlast the computation that produced it. That fact does not establish that all of its dependencies may be retained, that its root remains alive, or that it may still be committed. The rules governing those cases are **Pending**, O04, O09, and O12.

[3] The language does not yet specify ownership transfer, destruction order, garbage collection obligations, reference counting, or manual deallocation. No one of these mechanisms is mandated by this document merely because it is available on a target backend.

6.5 Execution domains [basic.execution]

[1] The examples and equations in this revision describe ordinary sequential execution unless explicitly stated otherwise. Transition application is not a promise of a transaction, a hardware atomic operation, or isolation from concurrent observers.

[2] The concurrent memory model, alias restrictions needed for concurrent access, synchronization operations, and behavior of conflicting unsynchronized accesses are **Pending**, O20. A compiler's ability to derive read and write sets does not by itself authorize parallel execution.

7 Types and modifier dimensions [types]

7.1 General [types.general]

[1] A type describes the semantic category of an entity or result. The modifiers `view`, `mut`, `transition`, `defer`, and `context` express language-level properties of access, state change, evaluation, or dependency provision. They are not defined as ordinary nominal generic wrappers.

[2] An ordinary user-defined type named `View`, `Mutable`, `Transition`, `Deferred`, or `Context` does not acquire the meaning of a language modifier by virtue of its name. Conversely, the existence of a modifier does not prohibit an otherwise valid ordinary type declaration with a distinct identifier of that spelling.

[3] `pure` describes the effect properties of a function or computation. It is not one of these ordinary value-type modifiers. `expression` is not a source-language modifier at all; its former role is replaced by representation-transparent implementation freedom.

[4] The fundamental types, nominal type rules, subtyping, generics, type aliases, inference, and overload interactions are **Pending**, O03 and O06, except for the constraints stated in this document.

7.2 Independent dimensions [types.dimensions]

[1] Modifier combinations shall be interpreted through their semantic dimensions rather than as an unrestricted list of independent tokens.

Dimension	Forms used in the design	Distinction
Access	ordinary value, `view`, `mut`	Semantic value, read-only borrow, or mutation authority
State	ordinary, `transition`	Ordinary content or a description of state change
Evaluation	ordinary, `defer`	Ordinary evaluation contract or explicit runtime deferral
Dependency	ordinary, `context`	Explicit argument or context-provided dependency

[2] `view mut T` combines conflicting modes in the access dimension and is ill-formed. A read-only view does not simultaneously grant direct mutable access to the same referent through that view.

[3] The existence of different dimensions does not imply that every cross-dimensional combination is valid. The canonical order, normalization rules, and complete legality matrix are **Pending**, O10. In particular, this revision does not define modifier combination by permuting all five words.

7.3 Ordinary types [types.ordinary]

[1] An ordinary type `T` does not promise any particular materialization state. A callee shall not receive different source-language access merely because its argument was supplied by an unevaluated internal graph rather than concrete storage.

[2] An ordinary `T` parameter does not implicitly become `defer T`. Runtime deferral is a distinct, explicit property. See [defer.general].

[3] An ordinary value parameter is also not a declaration of unrestricted mutation authority over a caller's object. The detailed object and parameter rules remain **Pending**, O04, but passing an argument through a different physical representation shall not create a new mutation capability.

7.4 Access modifiers [types.access]

[1] `view T` denotes read-only, non-owning, non-escaping access to an existing `T`. It is intended to avoid a semantic requirement to copy the referent. Its use is subject to [access.view].

[2] `mut T` denotes authority to directly modify an existing `T`. It is subject to the applicable effect, lifetime, and aliasing restrictions. Its use is specified in [access.mut].

[3] Mutable access may be weakened to a read-only view where the borrow rules permit. A view shall not be strengthened into mutable access merely by a type conversion. Exact coercion syntax and borrow duration are **Pending**, O09 and O10.

7.5 State and evaluation modifiers [types.state]

[1] `transition T` denotes a state-change description related to a root of type `T`. A result of that type is not, solely by virtue of its type, a complete new object of type `T`. The roles of transition parameters and results are specified in [trans.input] and [trans.result].

[2] `defer T` denotes explicit runtime deferral of a computation producing a `T`. This is observable evaluation semantics, not a request to use a particular optimizer representation.

[3] `defer transition T` is a design form for a deferred transition computation. It shall not be interpreted as exempt from transition root, validity, or mutation-capability rules. Its complete typing and timing rules are **Pending**, O10 and O18.

7.6 Context dependencies [types.context]

[1] `context T` declares a typed dependency that the context mechanism may supply. It does not designate an unrestricted mutable global variable.

[2] The effect and access properties of a context dependency shall remain applicable when the dependency is supplied implicitly. Omitting explicit argument syntax shall not increase the authority of the receiving function. See [context.capability].

8 Expressions and function evaluation [expr]

8.1 Expressions [expr.general]

[1] An expression specifies a computation. Evaluating an expression can produce a result, contribute to control flow, or perform effects as permitted by its context. An expression's source syntax and its implementation representation are distinct.

[2] The conventional expression forms include literals, name expressions, parenthesized expressions, calls, unary operations, binary operations, and assignment. The transition-application form is a distinct semantic operation even though its working spelling contains `=`.

[3] The complete type rules and operator table are **Pending**, O03 and O06. Earlier operator precedence tables are historical evidence, not a complete current expression specification.

8.2 Names and parentheses [expr.primary]

[1] A name expression denotes the entity selected by name lookup. Whether evaluating that expression merely obtains a value or accesses an object depends on the entity and the applicable access rules.

[2] A parenthesized expression has the type and semantic result of the enclosed expression. Parentheses do not grant additional access, reset a transition's lineage, or turn a deferred computation into an ordinary value.

[3] Parentheses may determine grouping. They do not independently define the order in which otherwise unordered operand evaluations occur. The full sequencing rules are **Pending**, O07.

8.3 Function interfaces [expr.call]

[1] A call associates arguments with a function's declared parameters and evaluates the function subject to its declared effects and dependencies. Recursive calls are part of the conventional language foundation.

[2] For a parameter declared as `x: T`, uses of `x` observe the semantic argument of type `T`. A use of `x` is not, by itself, a source-level request to reevaluate the argument expression. The implementation shall preserve the argument's required effects and semantic result.

[3] An implementation may propagate an internal computation across a call boundary. It shall not thereby expose an additional parameter category, change the function's declared type, or give the callee an operation that tests whether its argument is materialized.

Example 1: The body of `twice` refers twice to the semantic input `x`.

```
pure fun twice(x: Int): Int {
    return x + x
}

twice(produce())
```

Note 1: If `produce` has a required observable effect, duplicating that effect merely because `x` has two uses is not permitted. If `produce` is pure, duplication still requires preservation of all applicable observations, including any specified failure or termination behavior.

8.4 Ordinary evaluation and sequencing [expr.evaluation]

[1] Unobservable delay of representation construction shall not introduce observable runtime laziness into an ordinary expression. In particular, a function body shall not acquire a second evaluation mode depending on whether the caller passed existing storage or a computation representation.

[2] The complete order of callee evaluation, receiver evaluation, argument evaluation, operand evaluation, and assignment-target evaluation is **Pending**, O07. This draft therefore does not require left-to-right argument evaluation or import C++ operand-sequencing rules.

[3] Effects that the eventual source-language evaluation rules require shall be preserved. An implementation shall not justify dropping an effectful argument solely because the corresponding parameter is unused or only partly inspected.

Example 2: Suppose `buildVector` reports a diagnostic and returns a vector. Projecting only the vector's `x` component does not, by itself, justify deleting the diagnostic. This remains true if the vector representation is never materialized.

8.5 Projection and intermediate values [expr.projection]

[1] An implementation may compute only the portions of a value required by its consumers when all applicable observations are preserved. Such projection is an optimization of the ordinary value semantics, not a distinct source-language calling convention.

```
pure fun getX(value: Vec3): Float {
    return value.x
}

getX(makeVector())
```

Example 3: Assume that producing the unused components of the result is pure, terminating, and incapable of any failure or other observation required by the applicable numerical and object rules. An implementation can omit those component computations. Purity alone is not a proof of all these assumptions.

[2] Field access can only be treated this way if the access itself has the required properties. If a future property mechanism invokes a function, the function's effects and dependencies must be included. A pure-looking member spelling is not proof of a pure operation; member and property rules are **Pending**, O06.

8.6 Assignment and application [expr.assignment]

[1] Ordinary assignment changes the state associated with an assignable target, subject to the target's authority and type rules. A transition result, by contrast, describes changes without performing ordinary mutation of its root.

[2] The form `transition(target) = change` denotes an application boundary. It shall not be interpreted as ordinary rebinding of `target`, nor as the assignment of a `transition T` value to an ordinary `T` slot. See [trans.apply].

[3] The value category and result, if any, of an assignment or application expression are **Pending**, O02 and O06. Examples in this document use application in statement position and do not depend on its producing a value.

8.7 Named arguments and indirect calls [expr.named]

[1] Named arguments are a language-design goal. Parameter-name matching, argument evaluation order, defaults, overload selection, and source or binary compatibility after a parameter rename are **Pending**, O06 and O07.

[2] Indirect calls and virtual dispatch do not suspend representation transparency. An implementation may materialize a value to satisfy a call boundary when required by its calling convention, but the source-level function contract shall remain the same.

[3] The source type system for callable values, effect subtyping, virtual methods, and context requirements of higher-order functions is **Pending**, O06, O08, and O10.

9 Declarations and statements [stmt]

9.1 General [stmt.general]

[1] Statements specify control flow and operations within it. Statements execute in sequence except where a control-flow construct specifies otherwise. This statement-level sequencing does not settle the order of subexpressions within an individual statement.

[2] Jvav retains ordinary imperative constructs as part of its language foundation. Their presence is compatible with the separation of pure computation from externally observable state changes.

[3] The complete grammar of declarations and statements is **Pending**, O02. The rules below record their established roles and do not supply omitted syntax by analogy to another language.

9.2 Variable and function declarations [stmt.decl]

[1] A variable declaration introduces a binding and can associate it with an initializer and an explicit type. The historical language permits inference from an initializer when a type is omitted. The scope of current inference is **Pending**, O06.

[2] A function declaration identifies the function's name, parameters, result information, and body. A parameter declaration associates a name with a type and, where applicable, a capability or dependency modifier.

[3] Whether a binding can be rebound is distinct from whether an object reached through that binding can be mutated. Neither the spelling of an immutable binding nor its physical representation substitutes for the access rules of its contents.

[4] The historical implementation treats `let` bindings as read-only and `var` bindings as mutable. Current examples use `val` for a non-reassigned binding. The normative spelling, parameter rebinding rules, and the interaction between local reassignment and purity remain **Pending**, O02 and O08.

9.3 Blocks [stmt.block]

[1] A block groups a sequence of statements into a single statement and introduces a local scope. Leaving a block ends the execution of its remaining statements unless execution subsequently reenters the block through its enclosing control flow.

[2] A block used as a transition result is interpreted according to its expected transition role. Its field clauses describe replacement values; they do not sequentially assign to the root as ordinary statements would. See [trans.result].

[3] Cleanup, destruction, and deferred-resource actions on block exit are **Pending**, O04. The word `defer` in this draft describes runtime-delayed evaluation and shall not be assumed to mean a block-exit cleanup statement.

9.4 Selection [stmt.if]

[1] An `if` statement evaluates a condition and selects the associated body when the condition is true. If an `else` body is present, it is selected when the condition is false. The unselected body is not executed by that statement.

[2] Conditions have Boolean meaning. The complete rules for accepted condition types and conversions are **Pending**, O03. The historical design requires a Boolean condition and does not establish general implicit truth-value conversion.

Example 1: A pure predicate can decide whether a transition should be computed. The subsequent application remains a distinct operation.

```
if entity.isAlive {
    transition(entity) = simulate(entity, context)
}
```

Note 1: This example assumes a valid target capability, a valid root-binding rule, and a suitable `simulate` declaration. Those assumptions are not supplied by the `if` statement.

9.5 Iteration [stmt.iteration]

[1] A `while` statement tests its condition before each execution of its body. A `do` statement with a trailing `while` condition executes its body before the first condition test, and tests the condition before any subsequent iteration.

[2] In the historical three-part `for` form, initialization precedes iteration, the condition controls entry into each iteration, and the iteration expression is evaluated after normal completion of the body. The exact syntax and its relationship to collection iteration are **Pending**, O02.

[3] The current design also uses collection iteration, written `for element in collection`. Its iterator protocol, aliasing behavior, mutation rules, ordering guarantees, and element-binding type are **Pending**, O06 and O09.

[4] Iteration is not automatically parallel. Applying transitions during a loop shall not be taken to make all iterations observe one common initial state. Such a snapshot or bulk-commit rule would require an additional specification and is **Pending**, O20.

9.6 Jump statements [stmt.jump]

[1] A `break` statement terminates its enclosing iteration construct. A `continue` statement transfers control to that construct's next-iteration step. The exact target rules for labels, nested constructs, and the iteration-expression behavior of each `for` form are **Pending**, O02.

[2] A `return` statement transfers control to the caller. If it has an operand, that operand supplies the function's result according to the function's result contract. A returned transition remains a description of change; returning it does not commit it.

[3] The historical language permits an operandless return only for a function without a result. The precise no-result type, implicit-return rules, all-path checks, and line-sensitive parsing are **Pending**, O02, O03, and O06.

9.7 Classes and structured state [stmt.class]

[1] Classes and members are part of Jvav's intended language foundation. The transition examples use structured state with named components, but do not require a particular inheritance model or backend object layout.

[2] Construction, inheritance, visibility, dispatch, properties, operator overloading, and generic classes remain **Pending**, O06 and O21. In particular, this draft does not make ordinary property setters the implementation of transition commit; setter effects require an explicit rule in [trans.failure].

10 Effects and purity [effect]

10.1 Effect contracts [effect.general]

[1] Effects are properties of computations. An ordinary value's physical representation does not determine whether computing that value is pure. A register value can have been produced by an effectful operation; a computation graph can represent a pure operation.

[2] The `pure` marker declares a function or computation to satisfy [effect.pure]. Purity is not an ordinary wrapper around the returned value and does not replace access capabilities in parameter types.

[3] The complete effect system, including effect inference, default purity, callable effect types, and combinations of function-level markers, is **Pending**, O08. The absence of `pure` is not a statement that a function necessarily performs an effect; it is also not sufficient proof that the function is pure.

10.2 Pure computations [effect.pure]

[1] A pure computation shall derive its result from its declared dependencies. For the same semantic dependencies, its result shall be deterministic to the extent specified by the operations it performs. It shall not obtain an additional input through an undeclared mutable global environment.

[2] A pure computation shall not produce an externally observable effect and shall not directly mutate existing externally observable state. Having read access to an entity shall not be treated as permission to modify it.

[3] Calling another computation, accessing a property, demanding a deferred value, or using a context dependency is subject to the same requirements. Indirection and implicit argument provision shall not provide a route around purity restrictions.

Example 1: A damage-resolution function can inspect the target through a view and inspect combat rules through a declared context dependency.

```
pure fun resolveDamage(
    damage: Damage,
    target: view Entity,
    context: context CombatContext
): DamageResult
```

Note 1: The `context` spelling removes repeated argument plumbing, not the dependency itself. A rules change can change the result because the semantic dependency has changed.

10.3 Reads and mutable dependencies [effect.read]

[1] A read-only access path does not establish that the underlying state is eternally immutable. Purity is interpreted with respect to the semantic dependencies used by the computation. An implementation shall not reuse a result across changed dependencies merely because the function is marked `pure`.

[2] Reading a clock, drawing nondeterministic random input, performing input/output, or consulting an undeclared mutable global cannot be treated as deterministic pure computation simply because the operation returns an ordinary value.

[3] Explicitly supplying a time value, a sampled value, or a rules snapshot can make those values ordinary declared inputs to a separate pure computation. The rules for mutable state behind a view or a context during the computation remain **Pending**, O09, O19, and O20.

10.4 Imperative expression within pure computation [effect.local]

[1] An imperative surface form does not, on its own, imply an external effect. A conditional selects a computation, and iteration can express repeated computation. The source design intentionally preserves these constructs.

[2] The exact permission for locally owned mutation, local variable reassignment, allocation, and temporary resource use inside a pure function is **Pending**, O08. This revision neither bans every local state change nor adopts an unrestricted local-mutation exception.

[3] The transition-result clauses of [trans.result] are descriptions of a new state, not direct mutable access to the old root. They can therefore participate in pure computation under the transition rules without granting ordinary `mut` authority.

10.5 Purity and optimization [effect.optimize]

[1] Within the applicable semantic rules, an implementation may inline pure computations, share common results, combine producers with consumers, or eliminate unnecessary representations. The implementation shall account for all dependencies, including root state and supplied contexts.

[2] Purity alone does not establish that a computation terminates, that it cannot fail, or that changing floating-point evaluation is acceptable. The relevant numerical and observation rules remain **Pending**, O03 and O24. Optimization examples in this document state the additional assumptions on which they rely.

Example 2: Two calls that read different versions of an entity are not the same computation merely because the function name and root identity are equal. A cache key or equivalence proof must account for the actual semantic input state.

10.6 State-changing functions [effect.transition]

[1] A function that computes a `transition T` can be pure because its result is a description of change. A function that applies that change performs mutation and shall possess the appropriate authority. The result type alone does not grant that authority.

[2] The 0.2 source uses the function-level spelling `transition fun` for state-changing orchestration. Its exact effect set, relationship to an ordinary effectful function, default permissions, and combinations with other function markers are **Pending**, O08. It shall not be conflated with a `transition T` parameter or result.

11 Views and mutable capabilities [access]

11.1 Read-only views [access.view]

[1] A `view T` provides read-only access to an existing `T`. It does not own the referent and does not grant authority to directly modify it. A function receiving a view shall obey those restrictions independently of the physical representation used to pass the view.

[2] A view shall not escape beyond its permitted lifetime or usage region. In particular, storing a view for later use requires a lifetime rule that permits the retention. An implementation shall not regard a deferred computation or an internal computation graph as an exemption from this requirement.

[3] The design describes a view as a zero-copy borrow. This means that obtaining the view does not semantically require copying the referent into an independent value. It does not prohibit an implementation from performing unobservable loads or other representation changes.

```
pure fun distance(a: view Vec3, b: view Vec3): Float
```

Example 1: The two parameters provide inspection of existing vectors without transferring ownership or granting writes. Whether either argument can be a temporary, and how long such a temporary lives, is **Pending**, O09.

11.2 Mutable access [access.mut]

[1] A `mut T` provides direct mutation capability for an existing `T`. Mutation through that capability shall remain subject to the surrounding computation's effect restrictions.

[2] The presence of a `mut T` parameter does not, by itself, define exclusive ownership, a unique reference, a lock, or a transaction. The aliasing guarantees associated with the capability are **Pending**, O09.

[3] A mutable capability can support transition application when the target and validity rules are satisfied. Constructing a transition from a root does not create a new mutable capability for that root.

11.3 Capability weakening [access.weaken]

[1] Where the lifetime rules permit, mutable access can be used to provide a read-only view. Code receiving the weakened access shall not recover mutation authority solely from that view.

[2] Capability weakening shall not change the identity or semantic contents of the referent. It changes the operations available through the access path, not the object's type-independent history.

[3] The interaction between a live view and mutation through another alias is **Pending**, O09. This revision does not infer stable snapshot semantics from the word `view` and does not infer unrestricted simultaneous access from the word `mut`.

11.4 Retention and indirect escape [access.escape]

[1] Non-escape restrictions apply to indirect retention as well as explicit storage. A closure, context provider, deferred value, or materialized transition that retains a borrow is subject to the same applicable lifetime obligations as a direct use of that borrow.

[2] The complete rules for returning views, storing them in aggregates, reborrowing, passing them through higher-order functions, and using them in generic code are **Pending**, O09 and O10. A valid implementation must resolve these questions before claiming general support for such programs.

12 State transitions [trans]

12.1 General [trans.general]

[1] A transition separates the decision about how state should change from the operation that changes the state. A computation producing `transition T` describes changes concerning a root of type `T`. It does not produce an ordinary replacement `T` solely by returning that transition.

[2] Constructing, returning, or passing a transition shall not, by itself, apply its changes to the root. Application requires the explicit boundary specified in [trans.apply]. This distinction permits pure decision logic to be reused for execution, inspection, simulation, and testing.

[3] A transition may specify only part of the root's state. Components not changed by the transition retain their old-state values under a valid application. This sparse-update property is independent of whether the implementation allocates a transition object.

Note 1: A transition is not defined as a callback with arbitrary mutation rights. Its relationship to a root and its old-state reads provides semantic structure that an opaque effectful callback does not provide.

12.2 Rooted inputs [trans.input]

[1] The current design associates a transition computation with a designated root. In the declaration `entity: transition Entity`, `entity` supplies the rooted state against which the computation describes changes.

[2] Such a parameter permits the pure computation to read the relevant old state and describe a result. It does not confer the ordinary direct-write authority of `mut Entity`. A root read and a mutation capability are separate semantic notions even when they concern the same underlying entity.

```
pure fun move(
    entity: transition Entity,
    dt: Float
): transition Entity {
    val nextVelocity = entity.velocity + entity.acceleration * dt
    return {
        position = entity.position + nextVelocity * dt
        velocity = nextVelocity
    }
}
```

Example 1: The computation reads the root's position, velocity, and acceleration and supplies replacement values for position and velocity. It does not write either member while constructing the result.

[3] The shared spelling `transition T` is used in the source design for both the rooted input role and the produced change role. The complete static distinction between these roles, the rules for lifting an existing object into a rooted input, and the meaning of reading an already computed transition are **Pending**, O11. This example does not settle those rules.

12.3 Root extent and authority [trans.root]

[1] A transition's changes shall be interpreted relative to its designated root. A pure transition computation shall not obtain arbitrary write access to unrelated mutation roots simply because they are reachable through names or references.

[2] When a logical operation needs to coordinate changes to several objects, the source design favors choosing a common containing root, such as a world containing those objects. The legal extent of a root, including whether a referenced object is part of the root's owned state, is **Pending**, O14.

[3] Static type equality is not sufficient evidence that two transition values have compatible roots or compatible old states. Any transformation that combines transitions shall preserve the relevant root and state relationships.

Example 2: A transition concerning one `Entity` cannot be assumed applicable to every `Entity`. Conversely, a `transition World` does not imply authority to modify every object reachable anywhere in a process. Its extent requires a language rule.

12.4 Old-state reads [trans.old]

[1] Within a transition computation, reads of the designated root refer to that computation's logical input state. Describing a replacement for one component shall not cause a later root read in the same computation to observe the replacement as if it had already been stored.

[2] For a root `r` with input state `S0`, the abstract meaning of a root read at a valid path `p` is:

```
read(r, p, S0) = S0[p]
```

[3] This notation does not require a physical copy of `S0`. An implementation may preserve the required observations by scheduling reads and writes, retaining selected values, or using another equivalent representation.

```
pure fun swap(pair: transition Pair): transition Pair {
    return {
        a = pair.b
        b = pair.a
    }
}
```

Example 3: If the old values are `a = 2` and `b = 7`, a valid application produces `a = 7` and `b = 2`. The second clause reads old `a`, not the value supplied by the first clause.

[4] Sequential composition can deliberately establish a new input state for a subsequent computation. That is a different operation from ordinary old-state reads within one computation; see [trans.composition].

12.5 Sparse results [trans.result]

[1] A transition result supplies new values for selected components of the root. A field clause in a transition-result block describes the value to be supplied; it is not an ordinary assignment through a mutable reference to the root.

[2] An omitted component inherits its old-state value. The absence of a field clause shall not be interpreted as default initialization, deletion, or permission to leave the component indeterminate.

[3] For simple independent fields and a valid transition `t` computed against `S0`, let `W` map each explicitly changed field to its computed replacement. The result of a valid application is described by:

```
S1[p] = W[p]    if p is in domain(W)
S1[p] = S0[p]   otherwise
```

[4] The equation describes successful application to the appropriate state. It does not define stale-state rebasing, overlapping paths, property setters, failure behavior, or concurrent visibility. Those matters are **Pending**, O12, O14, O15, and O20.

[5] The empty write set describes no field-value change in this model. Whether an empty application affects versioning, lineage, validation, or other future metadata is **Pending**, O12 and O15.

12.6 Read and write dependencies [trans.dependencies]

[1] A transition computation has dependencies on the state it reads and the ordinary or contextual inputs it uses. Its read set and write set describe different aspects of the computation. A component may be read without being written or written without reading its previous value.

[2] For the `move` example in [trans.input], a simple field-level analysis yields:

```
root:   entity
reads:  {position, velocity, acceleration}
writes: {position, velocity}
other input: dt
```

[3] These sets are semantic analysis facts, not a required runtime object layout. An implementation may infer conservative approximations. If an optimization relies on absence from a set, its approximation shall be sound for the property being used.

[4] An apparent member read may involve additional dependencies if the language later permits computed properties or overloaded access. Analysis shall account for their semantics rather than assuming that all member syntax is a plain storage access.

12.7 Application boundary [trans.apply]

[1] The working form for application is:

```
transition(target) = change
```

[2] This form denotes computation or acquisition of a transition result and application to `target`. The operation shall possess the required mutation capability, shall respect the relationship between the transition and the target, and shall satisfy the applicable validity rules.

[3] The target's changes occur at application, not when a pure function first describes them. A computation without the necessary mutation authority shall not acquire it merely by constructing a value of type `transition T`.

[4] The root reads required to compute the change shall have their logical old-state meaning even if the application is lowered to in-place stores. A sequence of generated stores shall not feed an earlier store back into a later old-state read of the same computation.

[5] The exact target grammar, target evaluation order, root binding, checking of compatibility, and successful-result type of the application form are **Pending**, O02, O07, O11, and O12. Application failure and concurrent visibility are separately **Pending** in [trans.failure] and [trans.concurrent].

12.8 Computed transitions retained as values [trans.stored]

[1] A transition may be computed as a first-class value and retained before application. In this mode, its change computation is logically performed when the value is produced; later application is not an instruction to rerun that computation against whatever state then happens to exist.

```
val movement = move(entity, dt)

// Other work can occur here.

transition(entity) = movement
```

Example 4: `movement` is a computed change. Retaining it does not imply retaining an instruction to call `move` again at the final line. The legality of the intervening work depends on root validity, captured dependencies, and lifetimes.

[2] An implementation may avoid materializing a representation for the retained value if it can preserve the same semantic timing and dependencies. A source-level binding does not force physical storage, but eliminating that storage shall not convert an already computed result into a new computation over changed state.

[3] The circumstances in which retained transitions can be copied, moved, stored, returned, inspected, or applied more than once are **Pending**, O04, O10, O11, and O12.

12.9 Immediate execution context [trans.execute]

[1] The application form can provide a transition execution context for its right-hand computation. In this mode, the application command is executed at its position in program execution, while the implementation can retain and combine the transition computation without separately materializing its intermediate results.

```
transition(entity) = move(entity, dt)
```

[2] The abstract intent is computation followed by immediate application under the same command. The form is not, by itself, an asynchronous task, a queued command, or a `defer` value. Delaying representation construction does not postpone the command to an unrelated future demand.

[3] The source also illustrates nested transition-producing calls within this context. Such nesting provides an opportunity for fusion, but its exact typing and composition meaning remain **Pending**, O11 and O13. A compiler shall not infer a merge or sequential-update rule solely from the possibility of fusion.

[4] The point at which target expressions, ordinary arguments, and contextual dependencies are bound in this execution context is **Pending**, O07, O11, and O19. This revision establishes the distinction between immediate execution and retained precomputed changes without pretending that every binding rule is settled.

12.10 Timing distinctions [trans.timing]

[1] The following table distinguishes three mechanisms that shall not be conflated.

Form	Logical role	Outstanding questions
Compute a `transition T`, then retain it	Compute a change now; apply later	Validity, retention, replay
Application with a transition RHS	Execute a command now; permit fused computation and application	Root binding, sequencing, nested composition
`defer transition T`	Explicitly postpone a transition computation	Capture, demand, lineage, synchronization

[2] The first two modes can have equivalent behavior when their inputs, effects, and application timing are equivalent. They are not interchangeable across an intervening root change merely because their final source text calls the same function.

[3] The third mode has source-visible runtime deferral. It is not an implementation synonym for either of the first two modes. Its timing is subject to [defer.transition].

12.11 Validity and stale transitions [trans.validity]

[1] Retaining a computed transition separates the time at which its assumptions are established from the time at which it is applied. Changes to the root, changes to dependencies, or the end of a relevant lifetime can invalidate those assumptions.

[2] The source design anticipates a lineage or version relationship between a materialized transition and its old state. The exact validity predicate, what events invalidate it, and whether it is checked statically, dynamically, or by a combination of mechanisms are **Pending**, O12.

[3] This draft does not select rejection, recomputation, rebasing, rollback, or any other response to a stale application attempt. It also does not select whether an unrelated-field mutation invalidates a transition. These decisions shall remain visible as pending language work.

Example 5: A movement transition is computed for state `S0`; another operation changes the root to `S1`; the program then attempts to apply the saved transition. This example identifies a validity question. It does not establish an error class or a runtime exception name.

Note 1: Replacing a root's values with equal values does not, by itself, prove that lineage is unchanged. Whether identity, versions, read dependencies, or semantic equality govern validity is part of O12.

12.12 Composition [trans.composition]

[1] The design distinguishes combination of changes based on one old state from sequential state advancement. Those operations have different dependencies and shall not be treated as interchangeable meanings of composition.

[2] In a conceptual **merge**, both input transitions are computed from the same old state. Their write information is combined. A complete merge definition must specify root compatibility, overlapping writes, ancestor and descendant paths, and the effects of any conflict resolver.

[3] In conceptual sequential composition, called **then** in the design discussion, the first computation determines a state `S1` and the second computation is evaluated against `S1`. It is not equivalent to applying two independently precomputed changes from `S0` in an arbitrary order.

```
merge: A(S0) and B(S0) -> combine changes against S0
then:  A(S0) -> S1; B(S1) -> S2
```

[4] The words `merge` and `then` in this clause name semantic operations under consideration. They do not declare built-in functions, operators, or standard-library APIs. Their syntax and full static and dynamic rules are **Pending**, O13.

[5] Where changes overlap, the design does not select an implicit last-writer-wins rule. Whether overlap is forbidden, requires an explicit resolver, or is addressed by another mechanism remains **Pending**. Examples shall not silently choose a policy.

12.13 Fusion and in-place execution [trans.fusion]

[1] An implementation may combine transition computations and application so that no intermediate transition representation survives in executable code. It may retain only the necessary loads, arithmetic, and stores when that transformation preserves all applicable semantics.

[2] Fusion shall preserve old-state reads, sparse updates, dependencies, application timing, and any applicable validity or failure behavior. It shall not merge independent application boundaries when their separation is observable.

[3] For the swap in [trans.old], a valid in-place lowering can retain both old field values before writing replacements. Storing the replacement for `a` and then reading that new `a` as the old value for `b` is not an equivalent lowering.

[4] The phrase *pure in-place state transition* describes a pure change computation followed by an efficient application. It does not mean that arbitrary observable stores are reclassified as pure. Purity belongs to the computation; mutation occurs at application.

12.14 Failure and commit visibility [trans.failure]

[1] Construction of a transition does not perform its root mutation. Whether failure during computation or application can be caught, what kind of failure exists, and whether any application can leave a partially updated target are **Pending**, O15 and O24.

[2] No all-or-nothing exception guarantee, transactional rollback, atomic visibility, or reentrancy guarantee is specified by this revision. In particular, the word *commit* does not import database transaction semantics.

[3] The interaction with user-defined setters, constructors, destructors, validation hooks, resource replacement, and callbacks is **Pending**, O04, O06, and O15. A field-update model over simple values is not sufficient to settle those interactions.

12.15 Concurrent access [trans.concurrent]

[1] Old-state semantics constrains a transition computation's logical inputs. It does not, by itself, provide a lock or a physical snapshot in the presence of uncontrolled concurrent mutation.

[2] The permitted interleavings of reads, validation, and writes; the treatment of competing commits; and the synchronization required to make a root stable are **Pending**, O20. Whole-root transactions are not promised for non-atomic objects.

[3] Read and write information may support future conflict analysis or parallel execution, but such execution shall be justified by the eventual memory and capability rules. A disjoint-looking field set alone does not resolve aliasing, failure, external effects, or root-version conflicts.

13 Explicit deferred evaluation [defer]

13.1 General [defer.general]

[1] `defer T` expresses explicit runtime-delayed evaluation of a computation producing a `T`. It is a source-language property and is distinct from an ordinary value whose implementation representation has not yet been materialized.

```
val result: defer Value = defer expensive()
```

[2] The declaration expresses runtime deferral. It shall not be reduced to an optional optimizer hint that an implementation can ignore when doing so changes the specified evaluation timing.

[3] The exact region delayed by `defer`, the operations that force it, and its type conversions remain **Pending**, O16 and O17. This clause establishes the mechanism's distinction from transparent representation, not a complete lazy-evaluation calculus.

13.2 Demand and memoization [defer.demand]

[1] The source design favors evaluation on first demand followed by reuse of the result. This memoized, call-by-need direction is **Pending**, O16; this revision does not convert that stated preference into a finalized rule.

[2] A complete definition must distinguish a deferred computation that has not been demanded, one whose evaluation is in progress, one that has produced a result, and one whose evaluation has failed. Whether failed evaluations are cached, retried, or otherwise represented is **Pending**.

[3] Repeated demand, recursive demand of the same computation, cancellation, copying a deferred value, and concurrent demand require explicit rules. No once-only execution or thread-safe publication guarantee is established here.

13.3 Capture boundary [defer.capture]

[1] The runtime deferral mechanism shall have a defined capture and evaluation boundary. Its use shall not silently turn every ordinary function call in the program into a source-language lazy call.

[2] The expression below exposes an unresolved capture choice:

```
defer combine(first(), second())
```

[3] One design discussed in the source evaluates and captures the arguments first, and delays only the application of `combine` to those captured arguments. Another possible design delays more of the expression. The selected rule, capture order, capture mode, and treatment of nested expressions are **Pending**, O17. Neither alternative is adopted by this draft.

[4] The capture mechanism shall respect the applicable access and lifetime restrictions. Explicit deferral does not authorize a non-escaping view to be retained past its valid region, nor does it confer mutable authority on a captured value.

13.4 Effects of creation and demand [defer.effects]

[1] Creating a deferred computation and demanding its result are distinct semantic events. Any effects performed while capturing dependencies belong to the creation event; effects of delayed evaluation belong to the demand event under the eventual capture rule.

[2] An effectful operation shall not be treated as pure merely because it is wrapped in a deferred value. A pure computation demanding such a value must satisfy the applicable effect rules; the effect typing of deferred values is **Pending**, O08 and O16.

[3] An implementation may optimize the representation of a deferred value, but shall preserve its source-visible behavior once that behavior is specified. Eliminating a deferred wrapper is valid only if doing so preserves the required creation and demand observations.

13.5 Deferred transitions [defer.transition]

[1] `defer transition T` combines runtime deferral with a change description related to a root. The validity of the eventual transition depends on when the root and its dependencies are bound, which state is read, and when application occurs.

[2] If root state is associated with a deferred computation before the root changes, a later demand or application raises a lineage question. If the root is instead bound at demand, the computation has a different timing contract. The source design identifies the problem but does not settle all binding rules; they are **Pending**, O18.

[3] Read-only observation does not necessarily invalidate a deferred transition. Mutation may invalidate it under the eventual validity rules. No rule for automatic refresh, silent recomputation, or transparent rebasing is selected by this document.

[4] An explicitly deferred transition shall not be assumed to offer transaction isolation. Lifetime, effect, lineage, and concurrency requirements remain separate obligations.

14 Context dependencies [context]

14.1 General [context.general]

[1] A `context T` parameter declares a typed dependency that can be supplied by the calling environment. The purpose is to avoid repeatedly writing the same dependency as an argument through intermediate calls while keeping the dependency expressed in the interface.

[2] The semantic input remains present even when the call site omits explicit argument syntax. A change to the supplied context can change a computation's result without violating purity, because the computation's dependency has changed.

```
pure fun resolve(
    damage: Damage,
    target: view Entity,
    context: context CombatContext
): DamageResult
```

Example 1: Combat rules and difficulty can be part of `CombatContext`. Resolving the same hit under different rules need not yield the same result. Reuse of a result must account for those rules.

14.2 Provision and resolution [context.resolve]

[1] The context system shall not be interpreted as access to arbitrary mutable global state. A dependency's availability and type must be established through the eventual context-provision rules.

[2] Provider declaration syntax, lexical or dynamic lookup, ambiguity, priority, missing providers, explicit overrides, and propagation through indirect calls are **Pending**, O19. This revision does not choose an ambient service locator or a thread-local mechanism as the language definition.

[3] A function's context requirements remain relevant at a call boundary even if it is inlined or specialized. Removing visible plumbing from generated code shall not remove dependencies from the semantic model.

14.3 Capability restrictions [context.capability]

[1] Context dependencies shall obey the same effect and capability restrictions as equivalent explicitly supplied dependencies. A pure function shall not perform a prohibited mutation or external effect through an implicitly provided context.

[2] Read-only context can participate in pure computation when its semantic contents are valid dependencies of that computation. How the context's stability, lifetime, and transitive contents are expressed is **Pending**, O19 and O09.

[3] Whether a mutation-capable context exists is **Pending**, O19. If such a mechanism is introduced, its authority must be explicit in the effect and capability system. It cannot be inferred from the fact that the context was found implicitly.

14.4 Contexts and retained work [context.retention]

[1] A retained transition or deferred computation may depend on contextual inputs. Delaying application or evaluation does not establish whether those inputs are captured now or resolved later.

[2] The context-binding moment for deferred work, the lifetime of a retained provider or context value, and the role of context changes in transition validity are **Pending**, O17, O18, and O19. An optimizer shall not interchange early binding and late binding if they have different semantics.

15 Representation transparency [repr]

15.1 Source-level contract [repr.general]

[1] The representation of an ordinary value shall not be observable as a distinct source-level type or evaluation category. A function declared with `x: T` shall observe the meaning of `T`, not whether the caller supplied a register, borrowed implementation storage, a graph, a thunk, or a materialized temporary.

[2] There is no `expression T` parameter category in this revision, and no source-level reflection or pattern matching on an ordinary value's materialization status is provided by the model.

[3] This requirement does not imply that the optimizer must choose one representation for all calls. It means that varying the representation shall not vary the source-language contract.

15.2 Cross-call propagation [repr.call]

[1] An implementation may retain producer computations across ordinary function boundaries when it can preserve all applicable semantics. Inlining, specialization, component projection, or graph propagation may be used for that purpose.

```
pure fun normalize(v: Vec3): Vec3
pure fun length(v: Vec3): Float

length(normalize(a + b))
```

Example 1: If the operations have the required pure semantics, the implementation can combine them without separately materializing every intermediate vector. The source still describes ordinary `Vec3` values at the declared interfaces.

[2] Cross-call propagation is not mandatory. If an ABI boundary, indirect call, escape, or implementation constraint requires concrete storage, the implementation may materialize a value while preserving its source-level meaning.

15.3 Materialization elision [repr.elision]

[1] Materialization elision is broader than elimination of a copy between two existing storage slots. It can remove the need to construct the intermediate representation at all when no required observation depends on it.

```
pure fun lengthSquared(v: Vec3): Float {
    return v.x * v.x + v.y * v.y + v.z * v.z
}

lengthSquared(a + b)
```

Example 2: For a componentwise + with suitable pure arithmetic, the computation can be expressed using sums of individual components without a complete temporary vector. Algebraic reassociation or a change in floating-point precision is a separate question; neither is automatically authorized by eliminating storage.

[2] The ability to inspect addresses, object identity, resource operations, or backend representations is **Pending**, O04 and O22. Once such observations are defined, an implementation must respect them. An optimization opportunity does not decide the object model.

15.4 Optimization proof obligations [repr.obligations]

[1] Any transformation of the representation shall preserve semantic inputs and results, required effects, relevant evaluation timing, access restrictions, and valid lifetimes. A transition transformation shall additionally preserve its root relation, old-state reads, write meaning, and application boundaries.

[2] No optimization is required merely because it appears possible in an example. Likewise, an implementation's current inability to optimize a form does not justify changing the source-level meaning of that form.

Note 1: A compiler can retain semantic information in a high-level representation and later lower it into ordinary machine operations. Doing so is an implementation strategy for preserving information, not a new runtime guarantee.

16 Libraries and platform boundaries [platform]

16.1 Standard-library scope [platform.library]

[1] The long-term Jvav design includes a substantial standard library, cross-platform facilities, tooling, and interoperability. This revision does not specify library modules, namespaces, containers, concurrency APIs, input/output interfaces, or a runtime distribution.

[2] Examples use application-defined names such as `Vec3`, `Entity`, `World`, `Damage`, and `CombatContext`. These names do not designate standardized library types. Illustrative function names are not declarations of a standard-library API.

[3] The library specification and its relationship to language capabilities, effects, lifetimes, and backend facilities are **Pending**, O22.

16.2 Backends and interoperability [platform.backend]

[1] The language semantics shall not be defined by a single backend's physical object representation. Different backends may lower the same semantic constructs differently while preserving the applicable Jvav requirements.

[2] LLVM, JVM bytecode, and MSIL/.NET are named development directions. This document does not claim that implementations for these targets are complete, available, or mutually interoperable.

[3] Foreign interfaces must eventually specify representation, calling conventions, lifetime transfer, errors, and effects. C or C++ compatibility is a design objective from historical project material, not a portable ABI guarantee in this revision. These matters are **Pending**, O22.

16.3 Compile-time transformation [platform.transform]

[1] Typed, structured compile-time transformation is an exploratory direction. It is intended to preserve more structure than unrestricted textual substitution, but no complete transformation facility is specified here.

[2] Hygiene, staging, generated-name visibility, the order of expansion and type checking, capability access during transformation, and reproducibility are **Pending**, O21. This draft does not reserve transformer syntax or define an executable macro API.

17 Diagnostics and semantic validation [diagnostics]

17.1 Static restrictions [diagnostics.static]

[1] Rules that explicitly describe a form as ill-formed establish static restrictions. Rules that prohibit an operation establish semantic restrictions even where the enforcement mechanism is unfinished. The general requirements for diagnostic timing, number, wording, severity, and recovery are **Pending**, O23.

[2] An implementation claiming support for this semantic core shall not treat the absence of a complete diagnostic specification as permission to ignore its established capability or purity restrictions.

[3] Pending syntax or semantics shall not be documented as a portable guarantee solely because one implementation accepts a program. A diagnostic should identify the relevant experimental feature when acceptance depends on an implementation's choice for a pending issue.

17.2 Review obligations [diagnostics.review]

[1] Semantic review should distinguish a source restriction, a required runtime observation, an implementation strategy, and a pending choice. These categories are not interchangeable evidence of correctness.

[2] The informative review cases in [examples] identify consequences of the established design. They are not claims of execution on an existing compiler and do not constitute a complete conformance suite.

Annex A Grammar fragments [grammar]

Informative. This annex records notation used by the draft. It deliberately leaves unresolved productions abstract. In particular, it is not a complete parser grammar and does not determine token reservation, newline handling, operator precedence, or all legal modifier combinations.

A.1 Declarations [grammar.declaration]

[1] The examples use the following declaration shapes. `type`, `expression`, and `statement` are open categories. The no-result form and the final immutable-binding spelling are pending.

```
function-declaration:
    [ "pure" ] "fun" identifier "(" [ parameter-list ] ")"
        [ ":" type ] block

parameter-list:
    parameter { "," parameter }

parameter:
    identifier ":" type

binding-declaration:
    "val" identifier [ ":" type ] "=" expression

block:
    "{" { statement } "}"
```

[2] Some examples show function signatures without bodies to focus on the interface. This convention does not specify a bodyless-declaration grammar or a foreign declaration mechanism.

A.2 Modifier forms [grammar.type]

[1] The following are the forms individually discussed in the semantic clauses. The list is not an inductive rule that allows every form to be nested arbitrarily.

```
documented-type-form:
      type-name
    | "view" type-name
    | "mut" type-name
    | "transition" type-name
    | "defer" type-name
    | "context" type-name
    | "defer" "transition" type-name
```

[2] The combination `view mut T` is specifically prohibited. Other combinations require the legality and normalization rules tracked by O10. The absence of `expression` from this list is intentional and reflects its removal from the source type system.

A.3 Expressions and transition results [grammar.expression]

[1] Calls, member selection, and parentheses appear in examples with conventional shapes. Named-argument grammar is not fixed by this annex.

```
call-shape:
    expression "(" [ expression { "," expression } ] ")"

member-shape:
    expression "." identifier

parenthesized-shape:
    "(" expression ")"

application-shape:
    "transition" "(" target ")" "=" expression

defer-shape:
    "defer" expression

transition-result-shape:
    "return" "{" { field-clause } "}"

field-clause:
    field-designator "=" expression
```

[2] The target and field-designator categories, separation between field clauses, and disambiguation of a transition result from an ordinary block or another aggregate expression are pending. The production shows structure; it does not resolve parsing at these boundaries.

A.4 Control flow [grammar.statement]

[1] The following forms are used to discuss control flow. The bodies are shown as blocks for clarity; whether single-statement bodies are also accepted is pending.

```
selection-shape:
    "if" expression block [ "else" block ]

while-shape:
    "while" expression block

do-while-shape:
    "do" block "while" expression

collection-for-shape:
    "for" identifier "in" expression block

jump-shape:
      "break"
    | "continue"
    | "return" [ expression ]
```

[2] The historical three-part `for` form is not standardized by this fragment. Its exact delimiters differ between older prose and implementation evidence; see [implementation.history].

Annex B Semantic examples and review cases [examples]

Informative. These cases test the meaning of the established design on paper. They do not claim compiler execution. Any incomplete source syntax is explanatory notation. Each case states the assumptions necessary to avoid resolving an open language question accidentally.

B.1 A movement transition [examples.move]

[1] Assume an entity has plain scalar fields `position`, `velocity`, `acceleration`, and `health`; arithmetic on the chosen values is exact; the root binding is valid; and no intervening mutation occurs. Apply the body of `move` from [trans.input] with `dt = 1`.

Component	Old state	Computed change	State after valid application
position	10	15	15
velocity	3	5	5
acceleration	2	omitted	2
health	80	omitted	80

[2] The intermediate `nextVelocity` is 5, so the new position is 15. During pure construction, the entity still has position 10 and velocity 3. The state changes only at application. Acceleration and health are retained because no replacement was supplied.

[3] An implementation can obtain this result with selected loads and stores. Neither a full copied entity nor a heap-allocated transition object is necessary to satisfy the model. Whether either representation is actually eliminated is an implementation question.

B.2 An in-place swap [examples.swap]

[1] Assume two independent integer fields and a valid immediate application of `swap` from [trans.old]. The following abstract lowering preserves the required values:

```
oldA = load root.a
oldB = load root.b
store root.a, oldB
store root.b, oldA
```

[2] The following lowering is not equivalent:

```
store root.a, load root.b
store root.b, load root.a
```

[3] Starting from `(a, b) = (2, 7)`, the first form produces `(7, 2)` and the second produces `(7, 7)`. The second form violates the established old-state rule. This case does not depend on choosing a stale-transition policy or a concurrent memory model.

B.3 Same-state merge and sequential advancement [examples.composition]

[1] Assume a scalar state component starts at 10. Computation A supplies the old value plus 1, and computation B supplies the old value multiplied by 2. Arithmetic and each independent change are well-defined for this example.

[2] If both computations use the same old state, A supplies 11 and B supplies 20. Their write sets overlap. This draft does not determine a merged value; choosing 11, choosing 20, rejecting the merge, or invoking a resolver requires the unresolved composition policy.

[3] If a future sequential composition explicitly runs A first and computes B from the resulting state, the mathematical progression is `10 -> 11 -> 22`. Merely applying B's already computed replacement 20 after A would instead produce 20. These are different operations.

[4] This example is a reason to distinguish the two compositions, not a declaration of public `merge` or `then` APIs.

B.4 Stored changes and intervening mutation [examples.stale]

[1] Consider the following event sequence, where the first step computes a transition and does not commit it:

```
1. Root r has state S0.
2. Compute change t from r at S0.
3. Another authorized operation changes r to S1.
4. Attempt to apply t to r.
```

[2] The draft establishes that step 2 is a computation of a change and that step 4 is the prospective application boundary. It does not establish whether the attempt at step 4 is valid, diagnosed, rejected at runtime, or handled in another way. That is O12.

[3] A useful future test suite should cover mutation of a read field, mutation of a written field, mutation of an unrelated field, replacing an object with equal contents, reusing a storage location, and repeated application of the same change. The expected outcomes require a selected validity model before these cases can become conformance tests.

B.5 Ordinary arguments and hidden representations [examples.value]

[1] Assume `observe` performs one required external observation when evaluated, and `twice` uses its input twice as in [expr.call]. An implementation cannot duplicate the observation merely by representing the input as a producer to be expanded at each use.

[2] Similarly, assume `makeVector` has one required external effect and `getX` uses only its `x` component. It can be possible to avoid storing the other components, but their lack of storage does not justify deleting the required external effect.

[3] For a pure producer, a projection or duplication still needs to account for numerical behavior, failure, nontermination, and lifetimes once those are defined. The case establishes a representation boundary; it does not define the unfinished observation model.

B.6 Runtime deferral and capture [examples.defer]

[1] Use `defer combine(first(), second())` as a question for a future revision. There are at least two distinct timelines to examine:

```
Argument-capture direction:
    creation: evaluate argument computations and retain results
    demand:   evaluate combine with retained results

Whole-expression direction:
    creation: retain whatever environment the rule requires
    demand:   evaluate the delayed expression
```

[2] The source discussion favors a narrow, explicit delayed region, but does not finalize the capture rule. These timelines therefore have no selected winner in J0001. The order within each event is also subject to the sequencing rules.

[3] A future specification must make examples involving observable argument effects, expiring views, changed context providers, failures, and repeated demand unambiguous before implementations can be compared.

B.7 Separating damage resolution from execution [examples.damage]

[1] An application can separate deterministic rule evaluation from mutation and presentation. Damage, a valid read-only view of a target, and combat rules form the inputs to a pure resolution function. Its result can describe the decision without reducing hit points or starting an animation.

```
damage + target state + declared rules
                      |
                      v
              pure resolution
                      |
                      v
         result or rooted change
                      |
                      v
         authorized application
                      |
                      v
               updated state
```

[2] The same rule computation can be used to display a preview, evaluate a simulated action, or decide a real update. A real application must still satisfy root and capability rules. Presentation effects such as animation are distinct operations and do not become part of the pure resolver simply because they follow its result.

[3] Randomness should be made explicit in the relevant input or handled at an effect boundary. This architectural separation does not prescribe a random-number API, a game framework, or a library event system.

B.8 Review checklist for future implementations [examples.checks]

[1] The following is a compact set of semantic review cases. *Pending* means that no portable expected outcome has yet been specified.

Case	Consequence of this draft
Construct a pure transition	Does not apply the root change
Omit a field from a valid sparse change	Retains its old-state value
Swap two fields	Both replacement expressions use old state
Convert a view into write authority	Cannot gain authority solely from the view
Use `view mut T`	Ill-formed access combination
Inspect an ordinary parameter's materialization status	No such source-level distinction is provided
Duplicate an effectful producer through two parameter uses	Required effects must be preserved
Apply a saved change after root mutation	Pending validity policy
Merge two writes to the same field	Pending conflict policy
Demand the same deferred value twice	Pending memoization and failure policy
Mutate while another thread observes a commit	Pending concurrent memory model

Annex C Implementation directions and historical continuity [implementation]

Informative. This annex distinguishes possible compiler architecture from required language behavior. None of the following intermediate operations, pass names, or parser interfaces is a source-language or binary-interface requirement.

C.1 Retaining semantic information [implementation.pipeline]

[1] The design proposes retaining purity, access capability, transition root, dependency, and value-flow information until the compiler no longer needs that information for validation and optimization. Erasing it immediately into unrestricted loads and stores can make later analysis unnecessarily difficult.

```
Source structure and declarations
              |
              v
HIR: types, effects, capabilities, roots, dependencies
              |
              v
MIR: value flow, transition operations, borrow facts
              |
              v
Semantic analysis and legal optimization
              |
              v
Lowered operations and backend emission
```

[2] An implementation need not use these names or this number of stages. The important opportunity is to retain the facts needed to justify transformations, then lower them without changing source observations.

C.2 Transition operations [implementation.transition]

[1] The source sketches separate intermediate operations for materializing a change and for immediately executing one. The following notation makes the distinction explicit without defining an IR standard:

```
%change = transition.materialize %root {
    %old = transition.read %root, position
    transition.make %root { position = %new }
}
...
transition.commit %root, %change

transition.execute %root {
    ... compute from old state and apply ...
}
```

[2] The first form can retain a change across an interval; the second can expose a computation and its application as one execution context. The validity rules for the first form remain pending even if the IR carries a version or lineage token.

[3] An optimizer can preserve root reads until enough information is available to schedule loads before conflicting stores. It can then eliminate unneeded transition representations. Such a pass needs semantic proofs for fusion; the name `transition.execute` is not itself a proof.

C.3 Transparent value producers [implementation.values]

[1] Ordinary arguments can be represented internally as concrete values or as producers with known effects and dependencies. A consumer can request fields or components without the implementation first building an entire aggregate.

```
%value = value.producer <computation of a Vec3>
%x = value.project %value, Vec3.x
%result = use %x
```

[2] The producer is internal compiler information. It must not become a source-level `expression Vec3` parameter, a materialization-status reflection operation, or an implicit call-by-name contract. The fallback of materializing a value is semantically valid when required observations are preserved.

[3] Effectful producers require careful sequencing and sharing. Pure producers require correct dependency identity and the remaining observation rules. Both require lifetime information if a producer retains access to existing storage.

C.4 Frontend structure [implementation.frontend]

[1] The 0.2 source proposes local, value-oriented parser interfaces of the conceptual form `parse(source_view) -> ParseResult<T>`. Consumed ranges and remaining input can be returned explicitly instead of being hidden in a long-lived mutable parser object.

[2] A structural scanner can identify delimiter pairs, strings, comments, blocks, and opaque regions. Declaration headers can then be associated with body spans, allowing an IDE or incremental compiler to parse bodies on demand. A composed structural grammar and a local Pratt expression parser are considered implementation options.

[3] Local parsing can report spans and error codes, while a higher layer supplies source identity, paths, and complete diagnostics. None of these choices establishes a language-level effect rule or authorizes skipping semantic checks on required bodies.

C.5 Historical reconciliation [implementation.history]

[1] The public historical repository was inspected at commit `5bc1211ecde382046bb7cb1c04e7aa2be1abe92f`. Its older drafts and implementation provide evidence for conventional language constructs. They do not implement the full 0.2 semantic model.

Topic	Historical evidence	Treatment in J0001
Immutable binding	Parser accepts `let`; binder marks it read-only	Current examples use `val`; compatibility pending
Mutable binding	Parser accepts `var`	Role retained; purity interaction pending
Three-part `for`	Parser uses parentheses and colon separators; older prose also shows semicolons	Exact syntax pending
Return parsing	Parser considers whether an operand is on the keyword's line	Complete newline contract pending
Fundamental type spelling	Legacy code uses forms such as `int` and `bool`; 0.2 uses `Int` and `Float`	Canonical names and rules pending
Value categories	Jvav 25 explores lvalue, xvalue, and prvalue terminology	Not imported into the current object model
Access and effect experiments	`const`, `auto`, and `widen` appear in historical discussions or drafts	Historical; not adopted core modifiers
Compiler architecture	Historical parser advances a mutable token cursor	Pure local parsing remains an architectural direction

[2] The older Jvav 24 variable-declaration prose contains inconsistent descriptions of its two binding forms. The implementation's explicit read-only check supports the historical interpretation of `let` as read-only. This observation resolves the historical description used here; it does not select the current spelling.

[3] The local `Jvav` directory inspected for this revision contained a minimal C++ entry point, build setup, and skeletal headers. It was not treated as evidence that the current semantic mechanisms are implemented. Historical details above were checked against the public repository instead.

Annex D Open design issues [open]

Informative. Every issue in this annex has status **Pending**. Inclusion identifies a missing decision, not an endorsed solution. The existing normative constraints remain applicable while the issue is open. No implementation-specific choice is promoted to a language rule by this register.

D.1 Source text and lexical rules [open.001]

[1] **O01 - Pending.** Specify source encoding, accepted Unicode characters, identifier normalization and equality, reserved versus contextual keywords, comments, escapes, token boundaries, and invalid-input handling. Reconcile integer prefixes and leading-zero notation with the historical lexer and draft. A complete rule must distinguish malformed text, malformed tokens, and well-formed tokens used in invalid syntax.

D.2 Complete surface grammar [open.002]

[1] **O02 - Pending.** Decide `val` versus `let`, newline significance, separators, block and expression boundaries, optional bodies, and the complete forms of both `for` constructs. Specify parsing of transition-result blocks and application targets. Resolve line-sensitive return operands, nested selection, jump targets, and the statement or expression status of transition application. Annex A is not a substitute for these decisions.

D.3 Fundamental types and numerical behavior [open.003]

[1] **O03 - Pending.** Specify canonical type names, widths, ranges, Boolean conditions, numeric literals, arithmetic conversions, overflow, division by zero, floating-point formats and rounding, and the no-result type. Optimization rules must agree with these choices; eliminating an aggregate does not imply permission to reassociate arithmetic.

D.4 Objects and ownership [open.004]

[1] **O04 - Pending.** Specify identity, value and reference behavior of nominal types, storage duration, allocation, copying, moving, ownership transfer, destruction, and resource effects. Decide which address and identity observations are available. Preserve the distinction between an ordinary semantic argument and a mandatory physical copy.

D.5 Name lookup and program assembly [open.005]

[1] **O05 - Pending.** Specify declaration visibility, forward references, shadowing, duplicate names, imports, modules, cross-unit resolution, initialization order, entry points, and linking. A global name scope must not become an undeclared mutable dependency channel for pure computation.

D.6 Conventional type and declaration system [open.006]

[1] **O06 - Pending.** Complete function signatures, result inference, parameter rebinding, overload selection, named arguments, defaults, member declarations, property access, callable values, basic generic facilities, and iteration protocols. Decide which surface forms are inherited from the earlier language and which need migration rules.

D.7 Evaluation order [open.007]

[1] **O07 - Pending.** Specify sequencing of receivers, callees, arguments, operands, assignment targets, transition targets, and transition right-hand sides. Distinguish semantic evaluation from optional physical materialization. Required observable effects cannot be duplicated or removed merely because an argument is propagated as an internal producer.

D.8 Effect system and local state [open.008]

[1] **O08 - Pending.** Specify default purity, effect inference, effect composition, function-type compatibility, and the exact meaning of `transition fun`. Decide when local reassignment, locally owned mutation, allocation, failure, and temporary resource operations are admissible in pure code. Establish effect rules for deferred demand and higher-order calls.

D.9 Lifetimes and aliasing [open.009]

[1] **O09 - Pending.** Specify borrow formation, non-escape regions, reborrowing, temporary lifetimes, transitive access, and the relationship between live views and mutable aliases. Determine whether any mutable capability implies exclusivity and how stable reads are guaranteed. Cover closures, aggregates, contexts, and deferred computations that retain borrows.

D.10 Modifier combinations [open.010]

[1] **O10 - Pending.** Define a complete legality and normalization table for modifier combinations, their order, and their use in parameters, results, fields, locals, aliases, generics, and callable types. Distinguish structural language modifiers from ordinary nominal generic types. The rejection of conflicting `view mut T` access is already established.

D.11 Transition input and result typing [open.011]

[1] **O11 - Pending.** Distinguish a rooted transition input from a produced change while retaining or revising their shared `transition T` spelling. Specify root binding, conversion from existing objects, reading transition values, return-block typing, inspection, storage, and the typing of nested transition-producing calls. Decide how multiple potential roots are represented and disambiguated.

D.12 Lineage and stale applications [open.012]

[1] **O12 - Pending.** Define transition identity, relevant state versions, invalidation events, root lifetime checks, repeated application, and cross-target compatibility. Decide whether changes to unread or unwritten components matter. Choose a static, dynamic, or combined enforcement model and specify the behavior of invalid application attempts, without silently replacing precomputed changes with fresh computation.

D.13 Composition and conflicts [open.013]

[1] **O13 - Pending.** Specify same-state merge and sequential advancement separately. Define overlapping write behavior, explicit resolution if any, identity elements, order, and legality of composition across roots. Choose source syntax or library APIs only after those semantic distinctions are fixed. No implicit last-writer-wins rule has been adopted.

D.14 Root extent and field paths [open.014]

[1] **O14 - Pending.** Specify whether root extent follows ownership, containment, identity, or another relation. Define references, collection elements, dynamic indexes, parent and child paths, aliases between paths, and changes to object topology. Determine how a sparse update interacts with nested aggregates without granting arbitrary access to unrelated reachable objects.

D.15 Application failure and visibility [open.O15]

[1] **O15 - Pending.** Specify validation order, the observable stages of application, partial update behavior, failure propagation, rollback if any, reentrancy, and user-defined resource operations. Decide how setters or callbacks interact with a sparse state update and whether empty applications affect state metadata. The term *commit* currently promises no transaction semantics.

D.16 Deferred value lifecycle [open.O16]

[1] **O16 - Pending.** Finalize or replace the memoized first-demand direction. Specify success caching, failure caching or retry, recursive demand, repeated demand, identity and copying of deferred values, cancellation, and effects at creation and demand. Determine whether any concurrency guarantee exists for evaluation and publication.

D.17 Deferred capture and demand syntax [open.O17]

[1] **O17 - Pending.** Define exactly which part of `defer E` is postponed, when arguments and receivers are evaluated, how captures are represented semantically, and whether contexts are captured or resolved later. Specify explicit or implicit forcing, nested deferral, and admissible borrowed captures. Keep runtime deferral distinct from internal representation delay.

D.18 Deferred transitions [open.O18]

[1] **O18 - Pending.** Specify when a deferred transition binds its root, which old state it observes, when lineage is established, and how changes before demand or application affect validity. Coordinate these rules with capture, memoization, root lifetimes, replay, and concurrency. Do not infer a transaction mechanism from the combined modifiers.

D.19 Context resolution and authority [open.O19]

[1] **O19 - Pending.** Specify provider syntax, scoping, resolution priority, ambiguity, missing contexts, explicit overrides, lifetimes, and propagation through indirect calls. Decide whether mutable context capabilities exist and how they are declared. Establish the binding moment and dependency identity of contexts used by retained or deferred computations.

D.20 Concurrency and memory model [open.O20]

[1] **O20 - Pending.** Specify permitted concurrent reads and writes, synchronization, memory ordering, data races, conflicting commits, visibility of multi-field changes, and deferred evaluation on multiple threads. Determine the proofs needed to parallelize transitions or collection iterations. Old-state semantics alone does not provide concurrent snapshot isolation.

D.21 Advanced language facilities [open.O21]

[1] **O21 - Pending.** Specify the broader class and generic model, inheritance and dispatch where applicable, operators, compile-time evaluation, and typed AST transformation. For transformation, define hygiene, stages, expansion order, generated declarations, and effects. Historical ideas such as `widen`, `const`, and `auto` require explicit reconsideration before adoption.

D.22 Library and foreign interfaces [open.O22]

[1] **O22 - Pending.** Define standard-library scope, runtime obligations, backend profiles, foreign-call effects, ABI rules, ownership transfer, resource management, and error interoperability. Distinguish language portability from identical object layouts or calling conventions. Development goals for LLVM, JVM, and .NET are not implementation-completeness claims.

D.23 Conformance and diagnostics [open.023]

[1] **O23 - Pending.** Establish the complete conformance contract, diagnostic obligations, implementation limits, extension rules, version identification, and the evidence required for feature support. Define executable acceptance and rejection tests after the relevant grammar and semantics are settled. Do not treat pending design as undefined behavior.

D.24 Failure and termination observations [open.024]

[1] **O24 - Pending.** Specify which failures, traps, nontermination, resource-exhaustion events, and other exceptional outcomes are part of observable behavior. State how those outcomes constrain elimination, duplication, projection, and reordering of pure computation. A deterministic result on successful termination is not, by itself, a complete optimization contract.

Annex E Source record and revision history [sources]

Informative. This annex records the materials used to prepare J0001 and makes their roles explicit. It is not a set of normative references and does not extend the language by reference to implementation behavior.

E.1 Primary design source [sources.primary]

[1] *Jvav Language Design Draft 0.2*, dated 2026-09-21, supplied as `Jvav_Language_Design_Draft_0.2.docx`. This is the principal source for functional semantics, explicit effects and state changes, modifier dimensions, old-state transitions, transparent ordinary arguments, explicit deferral, contextual dependencies, and implementation directions.

[2] The supplied document's SHA-256 digest is recorded below to identify the exact input independently of later file edits:

```
8eb57ca22a51f792484cd18b61f01430b61
cdf4b2ee33a326c444448059e392d
```

[3] The two lines concatenate to form one digest. The source's 57 short sections have been reorganized into semantic clauses. Tentative directions remain pending; implementation sketches have been moved to an informative annex.

E.2 Historical drafts and implementation [sources.legacy]

[1] The public heckerpowered/Jvav repository was inspected at commit `5bc1211ecde382046bb7cb1c04e7aa2be1abe92f` on 2026-09-21.

[2] The source files Jvav 24.typ and Jvav 25.typ were read directly. The first provides conventional syntax and control-flow foundations. The second contains exploratory value-category and access-widening material. Neither supersedes the 0.2 semantic direction.

[3] The historical Mamba files `SyntaxFacts.cpp`, `Parser.cpp`, `Binder.cpp`, and parser tests were inspected to distinguish draft prose from implementation behavior. They provide evidence for legacy keywords, operator precedences, loop parsing, return-line handling, read-only bindings, and conventional declarations. They were not built or executed for this document.

[4] The local project directory supplied by the owner was also inspected. Its minimal entry point and skeletal parser header did not provide a complete historical draft or a current implementation of the new core. No local implementation file was changed.

E.3 Editorial reference and public site [sources.editorial]

[1] N4950, Working Draft, Standard for Programming Language C++, dated 2023-05-10, was consulted for document organization. Numbered clauses, stable labels, requirement wording, and informative annexes are editorial conventions used here; C++ language rules are not incorporated.

[2] The public Jvav website was inspected on 2026-09-21 for historical project context and links to earlier material. Promotional statements on that site were not treated as proof of implementation support, performance, or ABI compatibility.

[3] The website redesign is separate project work and is not specified by this language document.

E.4 Coverage of the 0.2 source `[sources.coverage]`

[1] The following map identifies where the source's design topics are carried forward. Source section numbers refer to the supplied 0.2 document.

Source sections	Subject	Destination
1-3, 30-32, 49-57	Vision, state pipeline, design priorities	[scope], [intro], [examples.damage], [open]
4-8	Purity, modifiers, views, mutable access	[types], [effect], [access]
9-17	Rooted changes, old state, application, composition	[trans]
18-24	Ordinary values and transparent computation	[basic.value], [expr], [repr]
25-27	Runtime deferral and deferred changes	[defer]
28-29	Context and capability restrictions	[context]
33-44	IR, lowering, backends, frontend design	[platform], [implementation]
45-48	Conventional language, named arguments, transformation, library	[lex], [stmt], [expr.named], [platform]

E.5 Changes introduced by J0001 `[sources.changes]`

[1] J0001 introduces a formal document structure, an explicit distinction between normative and informative material, a consolidated vocabulary, stable clause references, a register of pending decisions, and worked semantic review cases. It does not claim to finalize the source's unresolved type, lifetime, deferral, composition, or concurrency rules.

[2] The ordinary-parameter model of revision 0.2 is preserved: source-visible `expression T` is absent. The core state distinction is also preserved: describing a change does not apply it, root reads have old-state meaning, and sparse results do not require full replacement objects.

[3] The revision makes previously implicit editorial boundaries explicit. A backend plan is not an implementation claim; an optimization opportunity is not a compulsory lowering; old-state semantics is not a concurrent transaction; and pending semantics is not undefined behavior.

E.6 Revision record `[sources.revision]`

Document	Date	Status	Change
J0001	2026-09-21	Revision 0	First numbered English working draft based on the supplied 0.2 design and historical sources

[1] Future revisions of this document should preserve stable labels where the subject remains the same, identify resolved issue numbers, and describe semantic changes separately from editorial corrections. A revision identifier such as `J0001R1` denotes a revision of J0001, not a claim of language release 1.