YASS

Abstract Syntax Tree Generation

IAST Node Structures
within ZPE Programming Environment

Assignment, branching, looping and function definitions

Document metadata	Value
Status	Working draft for the official YASS whitepaper series
Language	YASS 26.9
Compiler/runtime	ZPE 1.14.8
Basis	Current YASSCompiler and IAST implementation, 25 July 2026

Purpose

This document records the compiler-facing shape of selected IAST nodes. Each entry defines the semantic role of every populated field and provides a diagram of the references created during compilation.

WHITEPAPER / INTRODUCTION

The purpose of this series of whitepapers is to provide a detailed technical description of how the YASS language is represented internally by the ZPE Programming Environment. In particular, the papers document the Inline Abstract Syntax Tree (IAST) structures produced by the YASS Compiler and subsequently interpreted by the ZPE Runtime Environment (ZRE).

When YASS source code is compiled, its textual syntax is transformed into a linked collection of IAST nodes. Each node represents a particular language construct, such as a variable assignment, expression, conditional branch, loop, function declaration, function call, structure, or module. The meaning of a node is determined not only by its type but also by the way information is stored within its fields, including `id`, `value`, `left`, `middle`, `next`, `scope`, and other associated metadata. The precise use of these fields varies according to the construct being represented.

These papers describe that structure on a node-by-node basis. Each entry explains the purpose of the node, the YASS syntax from which it may be produced, the meaning of its fields, its relationship with neighbouring or nested nodes, and the way in which it is processed by the compiler and runtime. Diagrams are provided to make the hierarchy and relationships within each structure easier to understand.

The documented node layouts form an internal contract between the YASS Compiler and the ZPE Runtime Environment. The compiler is responsible for constructing each node according to an agreed layout, while the runtime must interpret that same layout consistently. For example, if the compiler stores a loop condition in a node's `value` field and its body in the `left` branch, the runtime must retrieve those elements from precisely those locations. A disagreement between the two components could cause a construct to be interpreted incorrectly, produce unexpected behaviour, or prevent a program from executing altogether.

This contract also provides a stable reference for the continued development of ZPE. Changes to the compiler, runtime, debugger, transpilers, optimisation systems, or other tools that consume the compiled IAST can be assessed against a clearly documented representation. Where the layout of a node must change, the documentation identifies the corresponding contract that must be updated across every affected component.

In addition to supporting implementation and maintenance, these papers are intended to make the internal operation of ZPE easier to study. They provide a common technical reference for understanding how high-level YASS syntax becomes an executable representation, how individual nodes are connected to form complete programs, and how the runtime traverses those nodes to carry out the program's instructions.

The scope of these papers is therefore the compiled representation of meaningful YASS language structures and expressions. Constants used solely as datatype identifiers, parser symbols, runtime flags, or other non-structural metadata are not treated as independent IAST node contracts unless they participate directly in the structure of a compiled node. This distinction keeps the documentation focused on the structures that define executable YASS programs.

Together, the papers provide a formal and practical account of the boundary between YASS compilation and execution. Their purpose is to ensure that this boundary remains explicit, consistent, maintainable, and sufficiently well-defined for the compiler, runtime, and related ZPE tooling to evolve without losing compatibility with one another.

This document from here on has been generated with the assistance of AI.

FOUNDATIONAL CONCEPT / READING THE STRUCTURE

What the compiled structure means

The YASS compiler does not execute source text directly. It transforms the program into an inline abstract syntax tree (IAST): a graph of IAST records whose shapes encode the meaning of assignments, expressions, branches, loops, functions and values.

A semantic representation

An AST omits punctuation and surface syntax once those details have served their purpose. For example, brackets and keywords identify an IF statement during parsing, but the compiled IF node retains only the condition, satisfied body, alternatives and continuation.

> **Central rule.** The byte code in type determines how every other populated field must be interpreted. A left reference has no universal meaning: on IF it is the true branch; on CAST it is the source value; on INDEX_ACCESSOR it is the value being indexed.

Tree structure plus linked sequences

Relationship	Meaning
Tree edges	left, middle and IAST-valued value fields describe ownership, operands, branches and metadata.
Sequence edges	next joins statements, parameters, arguments, list elements, match arms and other ordered collections.
Wrappers	Some constructs wrap earlier nodes. Nested indexes and chained dots therefore preserve the result already compiled.
Sentinels	Builders sometimes retain a leading placeholder; the executable sequence then begins at sentinel.next.

Why the contract matters

The compiler and runtime meet at this representation. Compilation is correct only when it populates the shape expected by runtime traversal; execution is correct only when the runtime reads each node according to that same contract. The node pages in this whitepaper document that boundary explicitly.

FOUNDATIONAL CONCEPT / DATA MODEL

Layout of an IAST

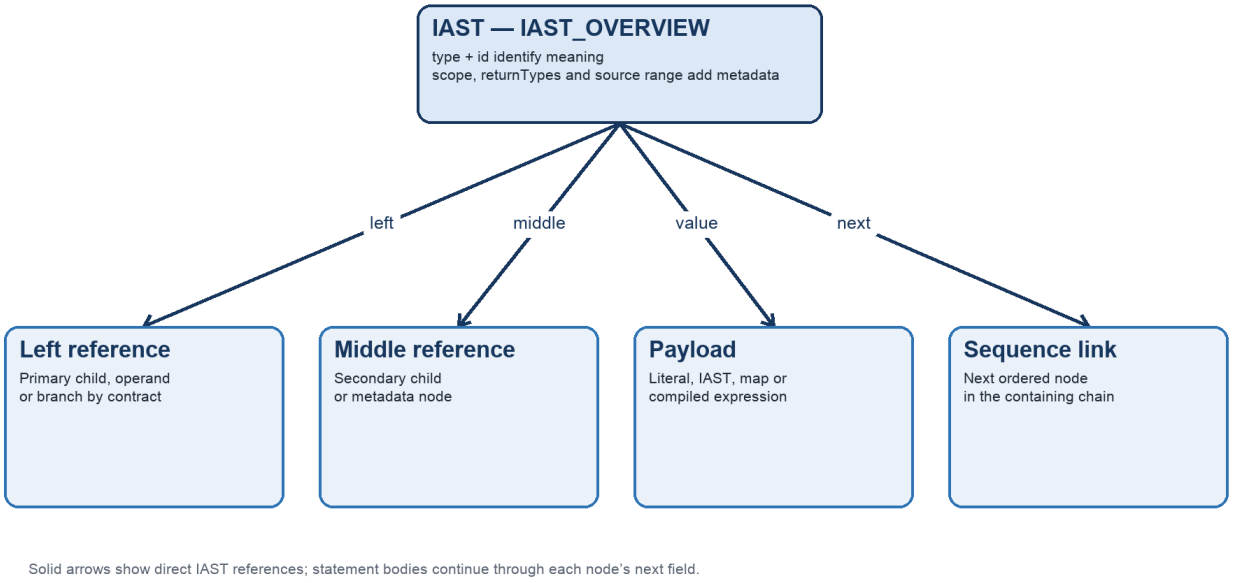

Figure A. General record layout; individual node contracts specialise each field.

Record fields

Field	General role
`type`	Byte code that selects the node's semantic contract and runtime handling.
`id`	Name or identifier used by nodes such as VAR, FUNCTION, NEW and WHEN.
`left`	First structural reference; its meaning is node-specific.
`middle`	Second structural reference or metadata holder; its meaning is node-specific.
`value`	Generic payload: a literal, IAST, compiled expression or supported serialisable container.
`next`	Ordered-sequence link to the following node.
`programStart / programEnd`	Source positions retained for correspondence with the original program.
`scope`	Visibility/access byte used by definitions and variables.
`returnTypes`	Optional byte array of permitted function return types.
`transient caches`	Runtime-only property/function lookup acceleration; not serialised as program meaning.

Persistence note. IAST serialisation writes semantic fields and recursively writes left, middle and next. Transient lookup caches are rebuilt at runtime and are intentionally absent from the executable format.

NODE 01 / COMPILED STRUCTURE

ASSIGN

Represents assignment of an evaluated expression or tuple to a target variable. The same layout is used by LET, with only the root type changed.

```
$total = $total + $value
```

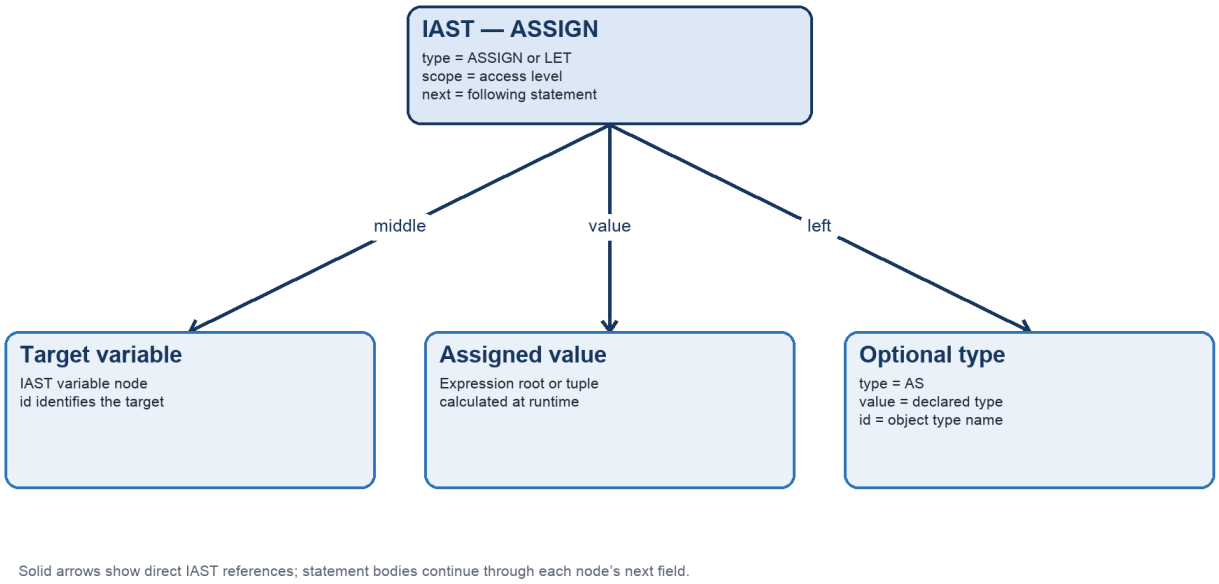

Figure 1. Compiler-produced IAST layout for ASSIGN.

Field contract

Field	Meaning
`type`	ASSIGN, or LET when the declaration form is used.
`middle`	The variable IAST receiving the value.
`value`	The compiled expression tree, or a compiled tuple.
`left`	Optional AS node for strong typing; otherwise null.
`scope`	Visibility copied from the variable declaration.
`next`	The next statement in the containing body.

Implementation note. The compiler first creates the assignment root, then stores the target in middle. This is a deliberate non-binary layout: middle is the destination and value is the source.

Source basis: `YASSCompiler.compileAssign(IAST, boolean, byte, String, boolean)`

NODE 02 / COMPILED STRUCTURE

IF

Represents the head of a conditional chain. The condition, satisfied branch and alternative branches are stored independently.

```
if ($score >= 50) print("Pass") else print("Fail") end if
```

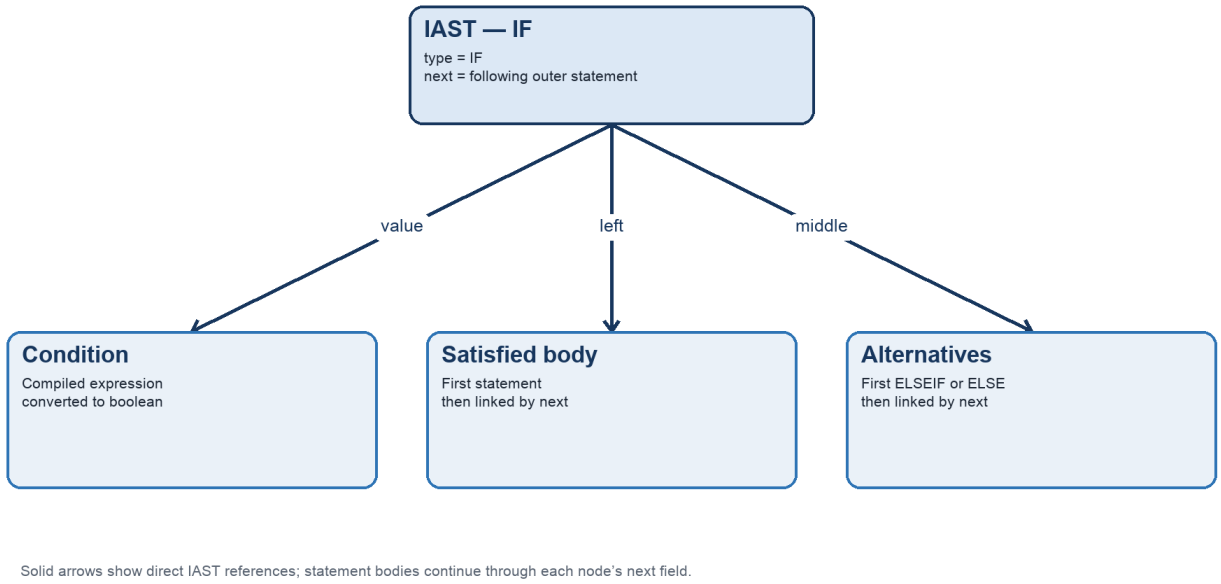

Figure 2. Compiler-produced IAST layout for IF.

Field contract

Field	Meaning
`type`	IF.
`value`	The condition expression.
`left`	First statement executed when the condition is true.
`middle`	First ELSEIF or ELSE node; null when no alternative exists.
`next`	The statement following the complete conditional.

Implementation note. Statements inside the satisfied branch form their own next-linked list. The alternative chain is separate and begins at middle.

Source basis: `YASSCompiler.compileIf()`

NODE 03 / COMPILED STRUCTURE

ELSEIF

Represents one conditional alternative in an IF chain. Each ELSEIF owns its condition and body, while next links to the following alternative.

```
elseif ($score >= 40) print("Resit")
```

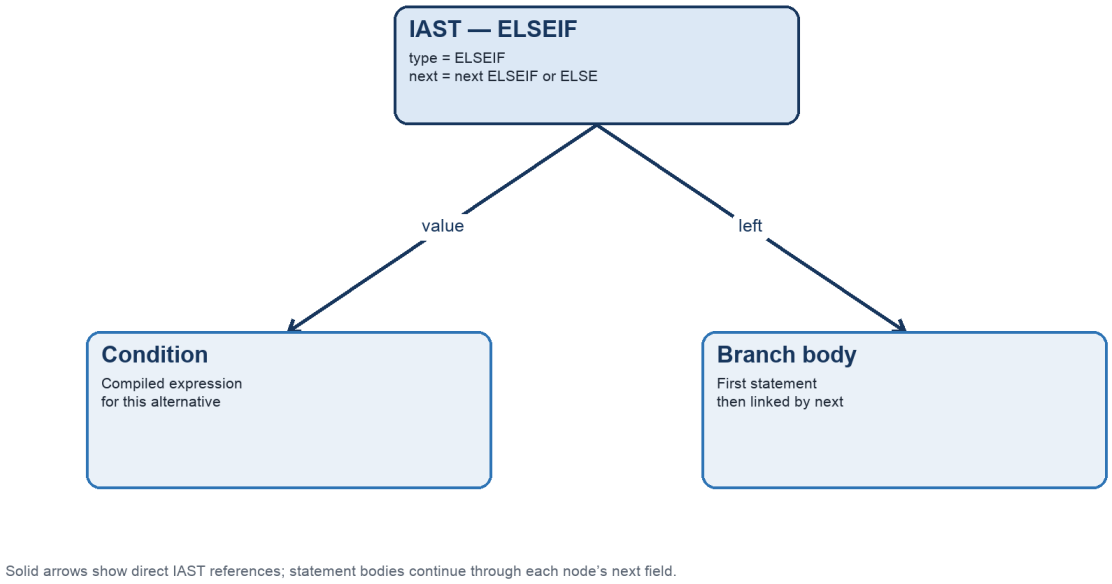

Figure 3. Compiler-produced IAST layout for ELSEIF.

Field contract

Field	Meaning
`type`	ELSEIF.
`value`	The alternative condition expression.
`left`	First statement executed when this condition succeeds.
`next`	The next ELSEIF or final ELSE in the chain.

Implementation note. ELSEIF nodes are not nested inside one another. They form a linear chain beginning at the parent IF node's middle field.

Source basis: `YASSCompiler.compileElseIf(boolean, boolean)`

NODE 04 / COMPILED STRUCTURE

ELSE

Represents the unconditional final alternative of an IF chain.

```
else print("Fail")
```

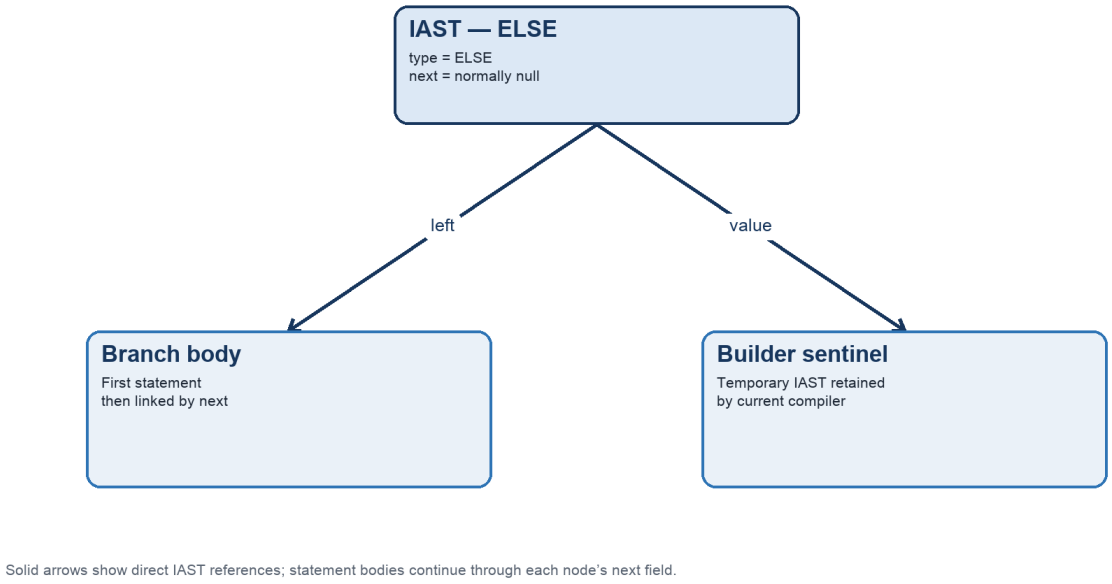

Figure 4. Compiler-produced IAST layout for ELSE.

Field contract

Field	Meaning
`type`	ELSE.
`left`	First statement in the unconditional branch.
`value`	A sentinel IAST used while constructing the body; left points to value.next.
`next`	Normally null because ELSE terminates the alternative chain.

Implementation note. The retained value sentinel is an implementation detail rather than semantic data. Runtime traversal uses left for the executable body.

Source basis: `YASSCompiler.compileElse(boolean, boolean)`

NODE 05 / COMPILED STRUCTURE

WHILE

Represents a pre-condition loop. Its condition is evaluated before every traversal of the body.

```
while ($index < length($items)) $index++ end while
```

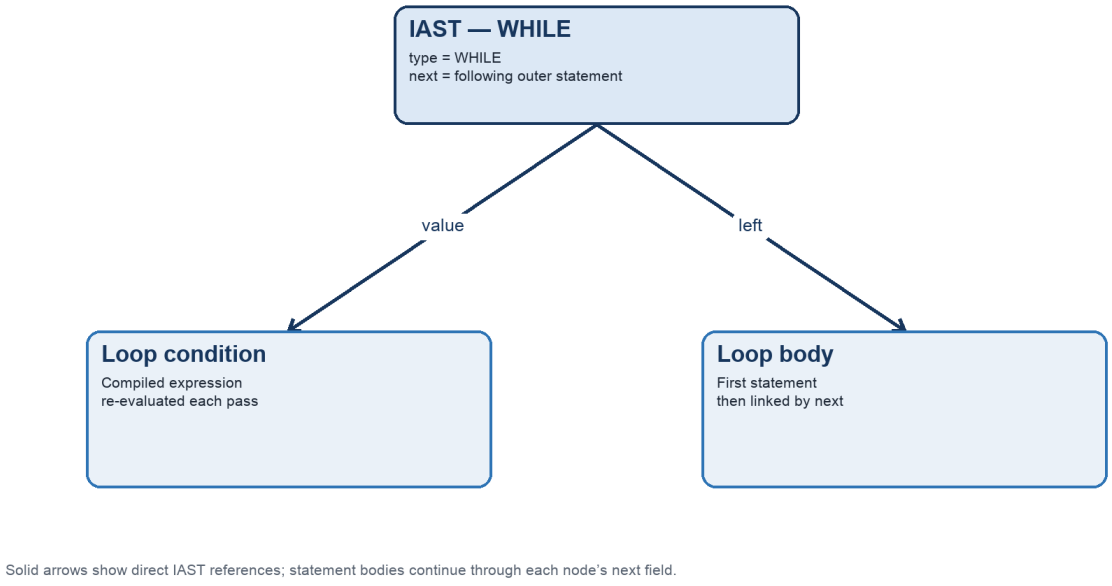

Figure 5. Compiler-produced IAST layout for WHILE.

Field contract

Field	Meaning
`type`	WHILE.
`value`	The loop condition expression.
`left`	First statement in the loop body.
`next`	The statement following the complete loop.

Implementation note. The body is a linked statement sequence. Break and return state are checked by the runtime while traversing this branch.

Source basis: `YASSCompiler.compileWhile()`

NODE 06 / COMPILED STRUCTURE

FUNCTION

Represents a function definition, including its name, parameters, body, visibility and permitted return types.

```
public function total(list $values) : number … end function
```

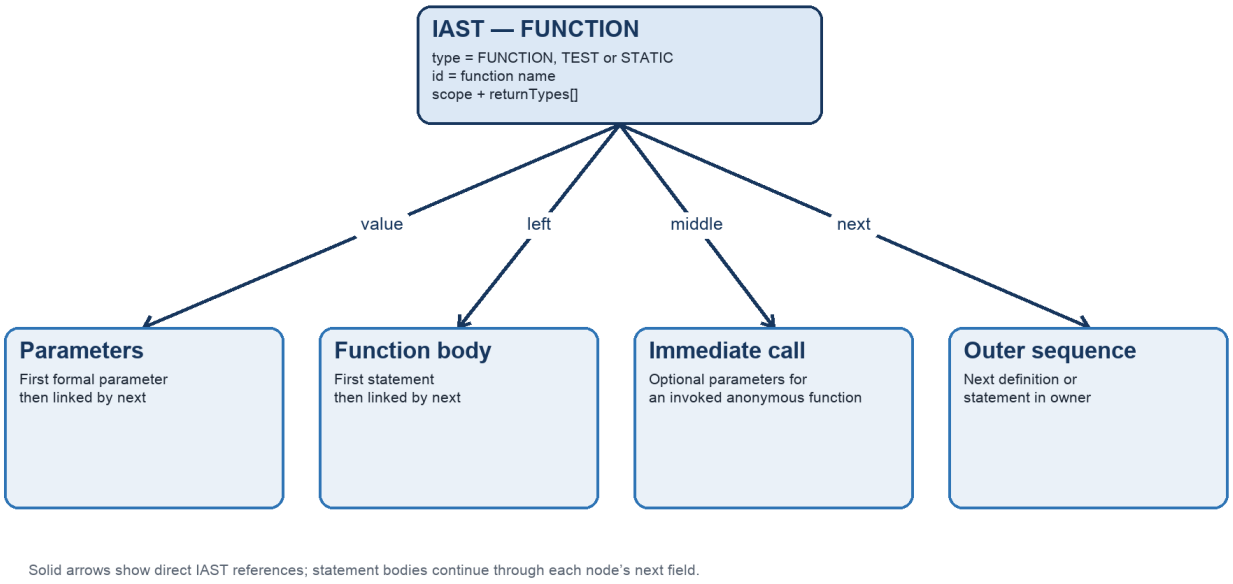

Figure 6. Compiler-produced IAST layout for FUNCTION.

Field contract

Field	Meaning
`type`	FUNCTION; TEST for test functions; STATIC for static definitions.
`id`	Function name, or an empty string for an anonymous function.
`value`	First formal-parameter node.
`left`	First statement in the function body.
`middle`	Optional immediate-invocation parameter list for anonymous functions.
`scope`	PUBLIC, PRIVATE, PROTECTED or GLOBAL.
`returnTypes`	Byte-code array of allowed return types.
`next`	Next definition or statement in the containing entity.

Implementation note. Parameters and body statements are separate linked lists. This prevents parameter metadata from sharing the executable body chain.

Source basis: `YASSCompiler.compileFunction()`

NODE 07 / COMPILED STRUCTURE

FOR

Represents a general counted loop with independent initialisation, condition, increment and body components.

```
for ($i = 0; $i < length($items); $i++) … end for
```

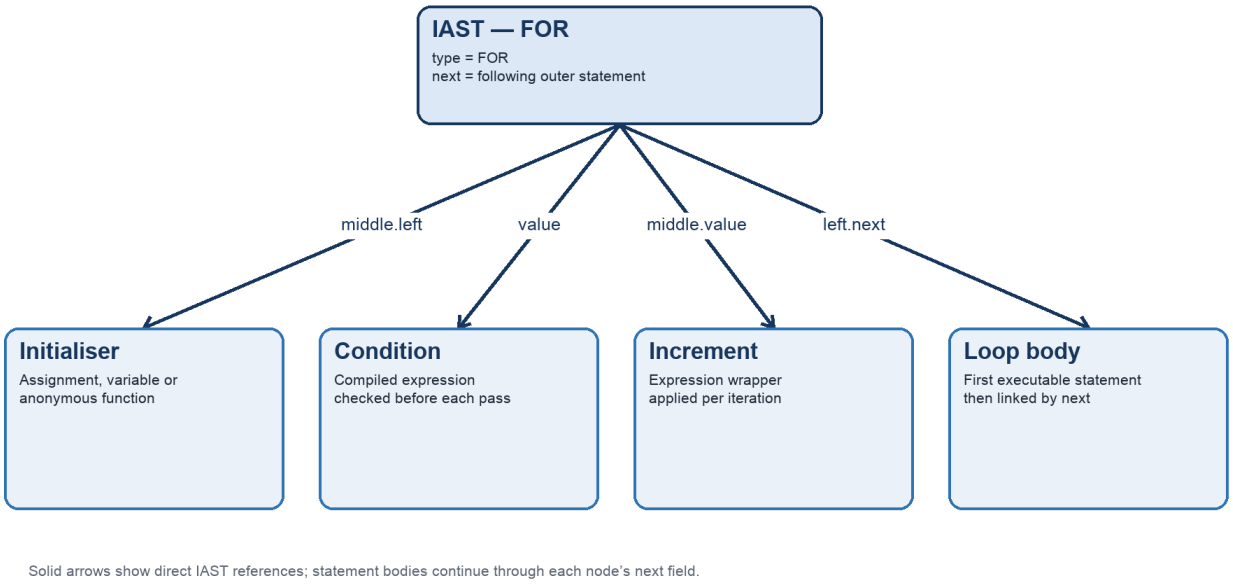

Figure 7. Compiler-produced IAST layout for FOR.

Field contract

Field	Meaning
`type`	FOR.
`middle`	A descriptor node: left stores the initialiser and value stores the increment expression.
`value`	The continuation condition evaluated before an iteration.
`left`	A retained body sentinel; the executable body begins at left.next.
`next`	The statement following the complete loop.

Implementation note. Unlike WHILE, the FOR body retains its construction sentinel. Runtime traversal therefore starts at left.next, while the three header components remain separate.

Source basis: `YASSCompiler.compileFor(); ZPERuntimeEnvironment.handleFor(IAST, …)`

NODE 08 / COMPILED STRUCTURE

FOR_TO

Represents the compact range-loop form. It reuses the FOR descriptor but replaces the condition and increment expressions with a terminal bound.

```
for ($index = 0 to 10) … end for
```

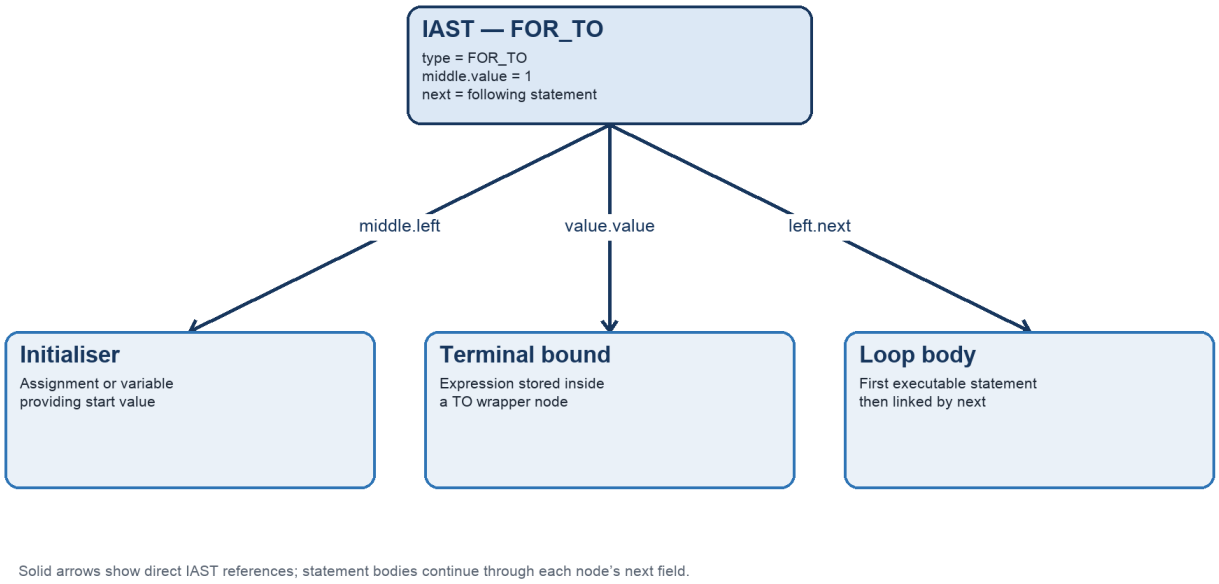

Figure 8. Compiler-produced IAST layout for FOR_TO.

Field contract

Field	Meaning
`type`	FOR_TO.
`middle.left`	The loop-variable initialiser.
`middle.value`	The numeric marker 1 written by the compiler for this compact form.
`value`	A TO node whose value field holds the compiled terminal-bound expression.
`left`	A retained body sentinel; executable traversal starts at left.next.
`next`	The statement following the complete loop.

Implementation note. Direction is not encoded in the AST. The runtime evaluates the start and bound, then selects a step of +1 or −1 from their ordering. The terminal bound is exclusive.

Source basis: `YASSCompiler.compileFor(); ZPERuntimeEnvironment.handleForTo(IAST, …)`

NODE 09 / COMPILED STRUCTURE

EACH

Represents iteration over a list, map, object or other iterable, with optional key/value binding and an optional stop condition.

```
for each ($value in $values) print($value) end for
```

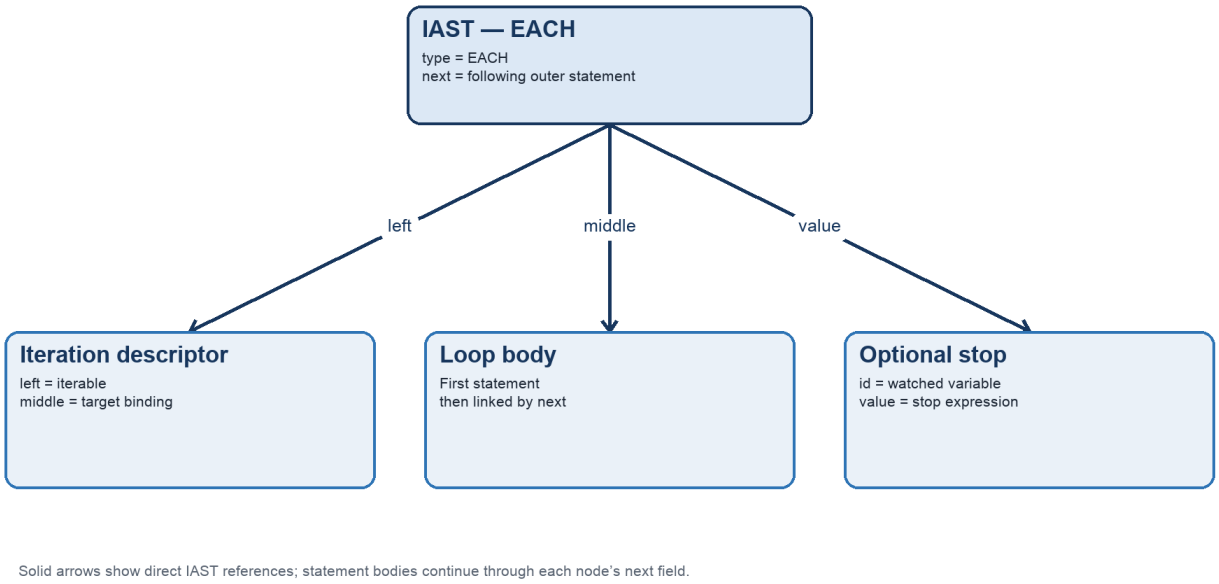

Figure 9. Compiler-produced IAST layout for EACH.

Field contract

Field	Meaning
`type`	EACH.
`left`	An assignment descriptor. Its left field is the iterable expression and its middle field is the binding target.
`left.middle`	A VAR target, or an ASSOCIATION holding key and value targets.
`middle`	First executable statement in the loop body.
`value`	Optional break-when descriptor containing the watched variable id and comparison value.
`next`	The statement following the complete loop.

Implementation note. The compiler normalises both “item in collection” and “collection as item” syntax into the same descriptor. Map/object iteration can bind two variables through ASSOCIATION.

Source basis: `YASSCompiler.compileForEach(); ZPERuntimeEnvironment.handleEach(IAST, …)`

NODE 10 / COMPILED STRUCTURE

UNTIL

Represents a pre-condition loop that continues while its condition is false.

```
loop until ($ready) … end loop
```

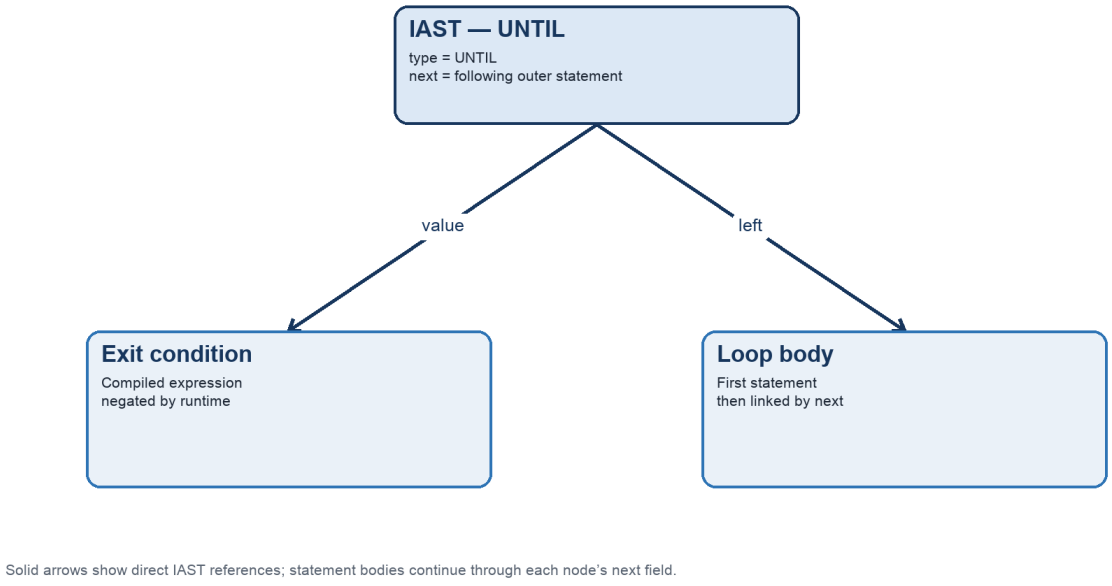

Figure 10. Compiler-produced IAST layout for UNTIL.

Field contract

Field	Meaning
`type`	UNTIL.
`value`	The condition expression; a true result terminates iteration.
`left`	First statement in the loop body.
`next`	The statement following the complete loop.

Implementation note. UNTIL has the same physical shape as WHILE. Its behaviour differs at evaluation: the runtime negates the condition before deciding whether to enter the body.

Source basis: `YASSCompiler.compileLoopUntil(); ZPERuntimeEnvironment.handleUntil(IAST, …)`

NODE 11 / COMPILED STRUCTURE

DO

Represents a post-condition loop. The body always runs once before the condition is evaluated.

```
do … loop while ($more)
```

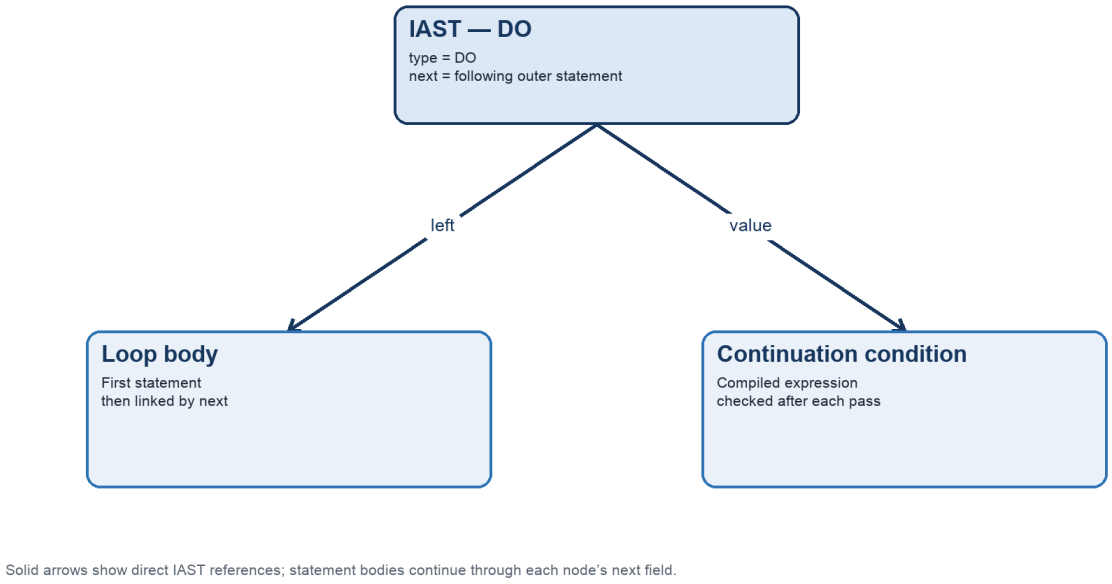

Figure 11. Compiler-produced IAST layout for DO.

Field contract

Field	Meaning
`type`	DO.
`left`	First statement in the loop body.
`value`	The continuation condition evaluated after body traversal.
`next`	The statement following the complete loop.

Implementation note. The AST does not need an explicit post-condition flag: the DO byte code selects runtime handling that traverses left before evaluating value.

Source basis: `YASSCompiler.compileDoWhile(); ZPERuntimeEnvironment.handleDo(IAST, …)`

NODE 12 / COMPILED STRUCTURE

WHEN

Represents multi-way selection over one variable. Case branches form a separate linked list, with the default body stored independently.

```
when ($choice) is 1 do … otherwise do … end when
```

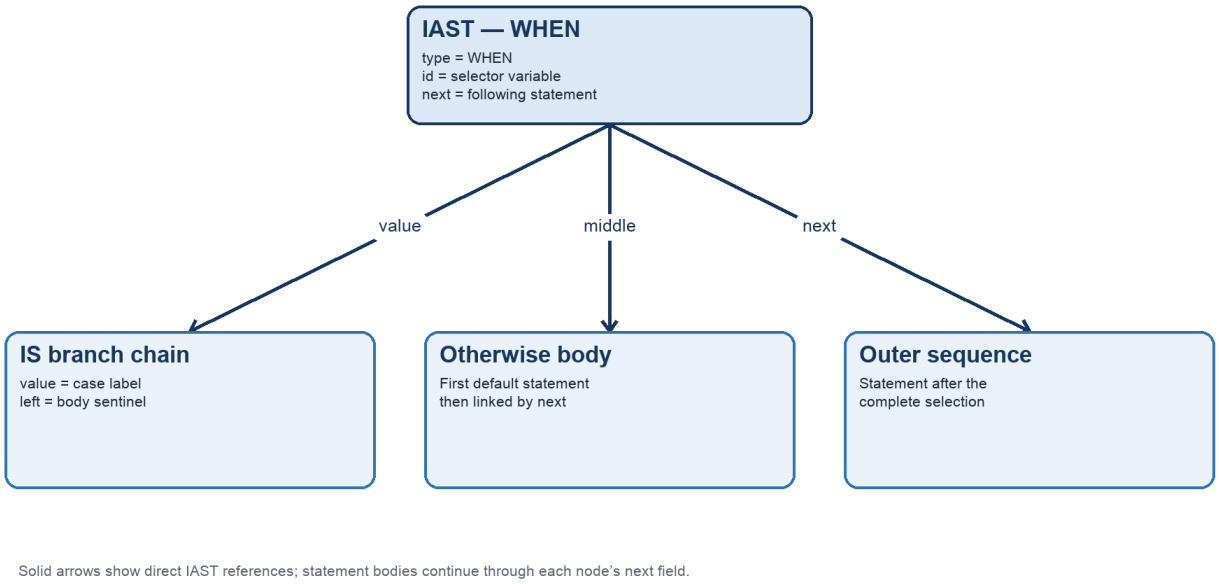

Figure 12. Compiler-produced IAST layout for WHEN.

Field contract

Field	Meaning
`type`	WHEN.
`id`	The selected variable id, including its $ prefix.
`value`	First IS node; subsequent cases are linked through each IS node's next field.
`IS.value`	The literal case label captured by the compiler.
`IS.left`	A retained body sentinel; executable case statements begin at IS.left.next.
`middle`	First statement of the OTHERWISE/default body, or null.
`next`	The statement following the complete selection.

Implementation note. WHEN and SWITCH compile through the same method and share this layout. Case labels are stored as values on IS nodes; runtime compares them with the substituted selector.

Source basis: `YASSCompiler.compileWhen(); ZPERuntimeEnvironment.handleWhen(IAST, …)`

NODE 13 / COMPILED STRUCTURE

MATCH

Represents an expression that selects and returns one result from a sequence of comparator/result arms, with an optional final fallback.

```
$result = match($value: 1 => "one"; 2 => "two"; else => "other";)
```

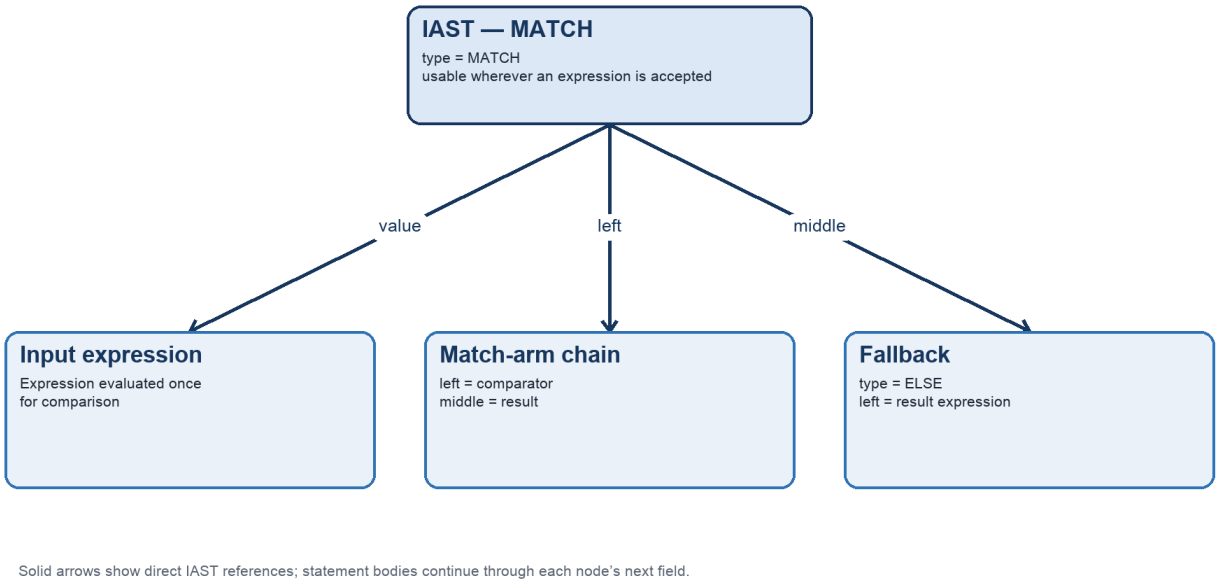

Figure 13. Compiler-produced IAST layout for MATCH.

Field contract

Field	Meaning
`type`	MATCH.
`value`	The input expression whose result is compared with every arm.
`left`	First arm descriptor; subsequent arms are linked through next.
`arm.left`	The comparator expression for this arm.
`arm.middle`	The result expression returned when the comparator matches.
`middle`	Optional ELSE node; its left field contains the fallback result.
`next`	Used by the surrounding expression or statement sequence.

Implementation note. Arm descriptors have no dedicated byte-code type: their meaning comes from their position in the MATCH node's left-linked chain. ELSE must be the final arm.

Source basis: `YASSCompiler.compileMatchExpression(); ZPERuntimeEnvironment.evaluateMatch(IAST)`

NODE 14 / COMPILED STRUCTURE

RETURN

Represents function termination with zero, one or several result expressions.

```
return $result, $status
```

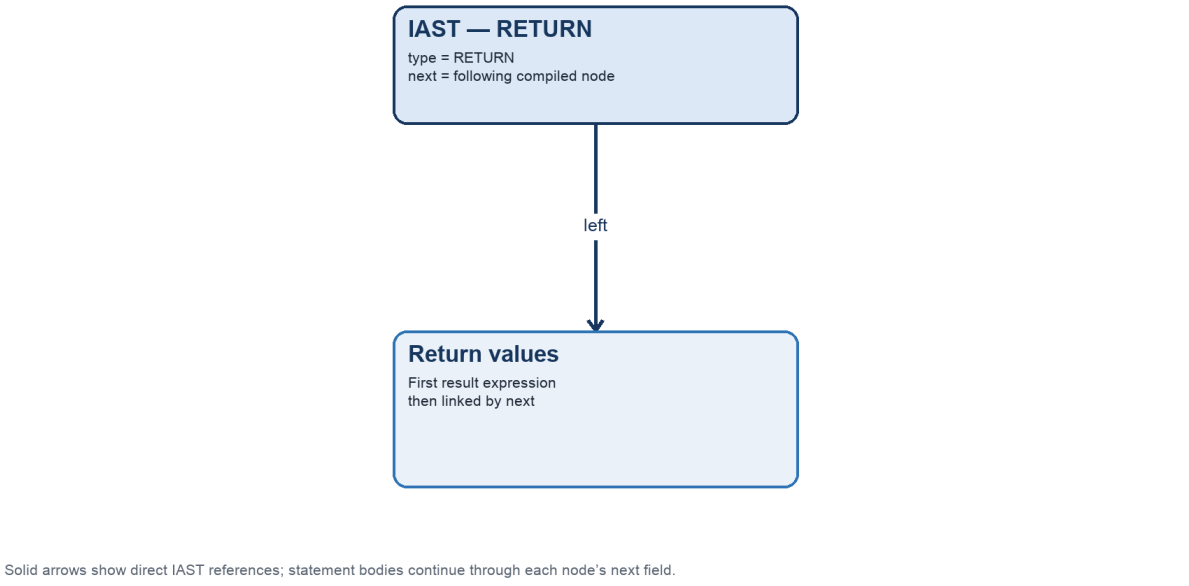

Figure 14. Compiler-produced IAST layout for RETURN.

Field contract

Field	Meaning
`type`	RETURN.
`left`	First returned expression, or null for a bare return.
`left.next`	Additional returned expressions in a multi-return statement.
`next`	The following compiled node, although runtime return state prevents normal traversal into it.

Implementation note. A single result and a multi-return use the same layout. The compiler creates no tuple wrapper; multiple expressions are represented directly as a next-linked chain.

Source basis: `YASSCompiler.compileReturn(); ZPEFunction traversal and return handling`

NODE 15 / COMPILED STRUCTURE

PRINT

Represents output of zero or more argument expressions through the runtime's configured print stream.

```
print("Total:", $total)
```

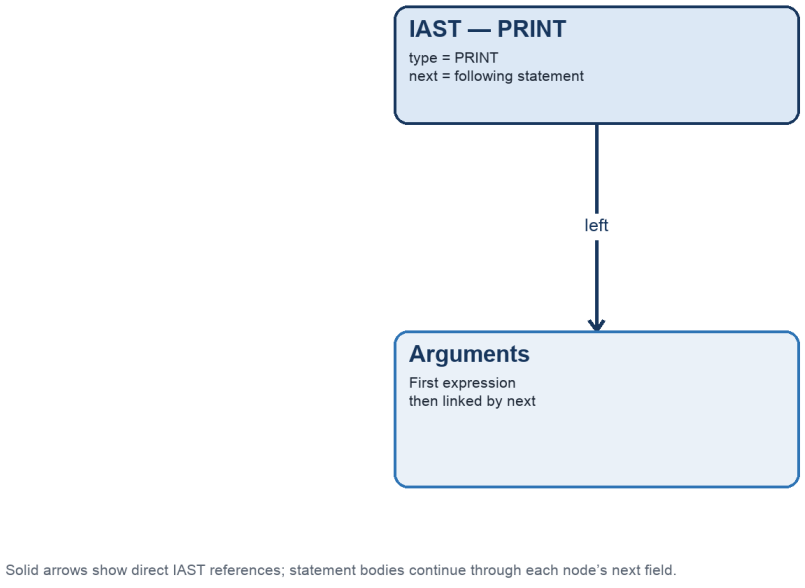

Figure 15. Compiler-produced IAST layout for PRINT.

Field contract

Field	Meaning
`type`	PRINT.
`left`	First argument expression; for an empty call this is an ARGUMENTS sentinel.
`left.next`	Subsequent argument expressions when one or more arguments are present.
`next`	The following statement in the containing body.

Implementation note. compileArguments normally removes its construction sentinel and returns the first expression. It preserves an empty ARGUMENTS node only when print() has no values.

Source basis: `YASSCompiler.compilePrint(); YASSCompiler.compileArguments(); ZPERuntimeEnvironment.handlePrint(IAST, …)`

NODE 16 / COMPILED STRUCTURE

LIST

Represents an ordered list literal, optionally with a replication expression that constructs a fixed-size array.

```
$values = [10, 20, 30]
```

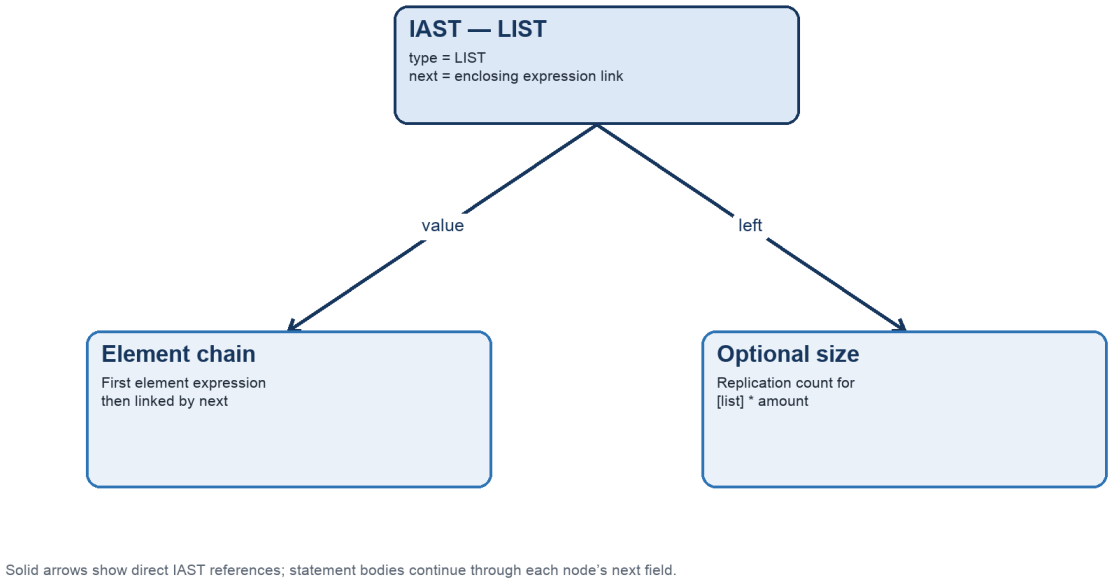

Figure 16. Compiler-produced IAST layout for LIST.

Field contract

Field	Meaning
`type`	LIST.
`value`	First element expression; each subsequent element is linked through next.
`left`	Optional repetition/count expression following the multiplication operator.
`next`	Used by the enclosing expression or argument list.

Implementation note. An empty literal has value = null. When left is present, runtime construction uses the first compiled element as the base value and creates a ZPEArray of the requested size.

Source basis: `YASSCompiler.compileList(); ZPERuntimeEnvironment.evaluateListLiteral(IAST, …)`

NODE 17 / COMPILED STRUCTURE

ASSOCIATION

Represents a map literal as an alternating key/value expression chain. The same root type also supports insertion-ordered map syntax.

```
$people = [77 => "Jack", 41 => "Emma"]
```

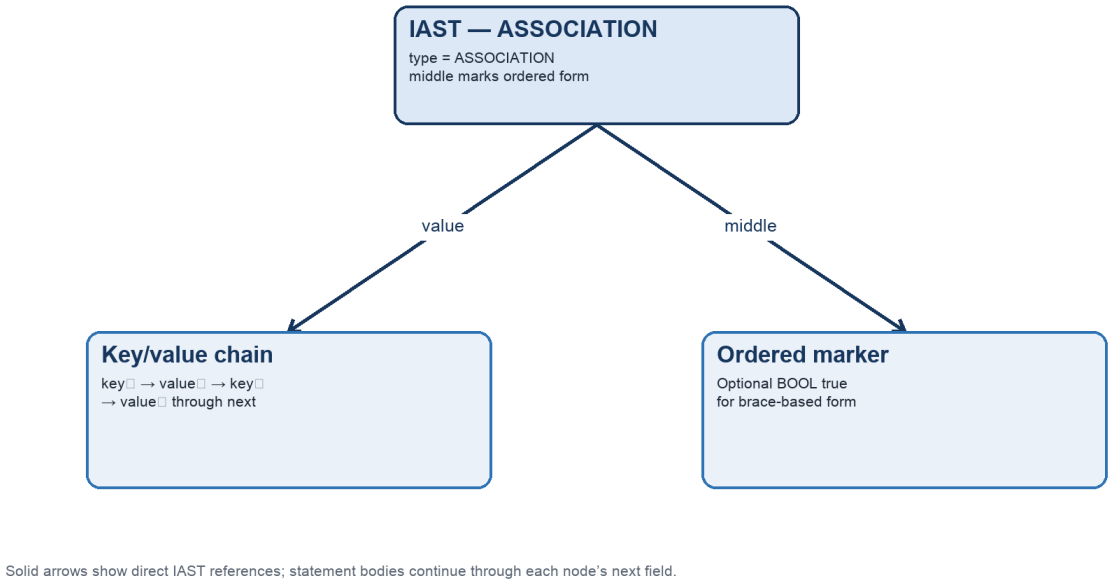

Figure 17. Compiler-produced IAST layout for ASSOCIATION.

Field contract

Field	Meaning
`type`	ASSOCIATION.
`value`	First key expression in an alternating key/value linked sequence.
`value.next`	The first value expression; its next points to the next key, and so on.
`middle`	A BOOL node containing true for the ordered-map form; otherwise null.
`next`	Used by the enclosing expression or argument list.

Implementation note. Pairs are positional rather than wrapped in pair nodes. The runtime advances across two linked nodes at a time, evaluating the key followed by its corresponding value.

Source basis: `YASSCompiler.compileList(); YASSCompiler.compileOrderedMap(); ZPERuntimeEnvironment.generateMap(IAST)`

NODE 18 / COMPILED STRUCTURE

OBJECT

Represents an anonymous object literal as a chain of named property descriptors.

```
$point = {x: 10, y: 20}
```

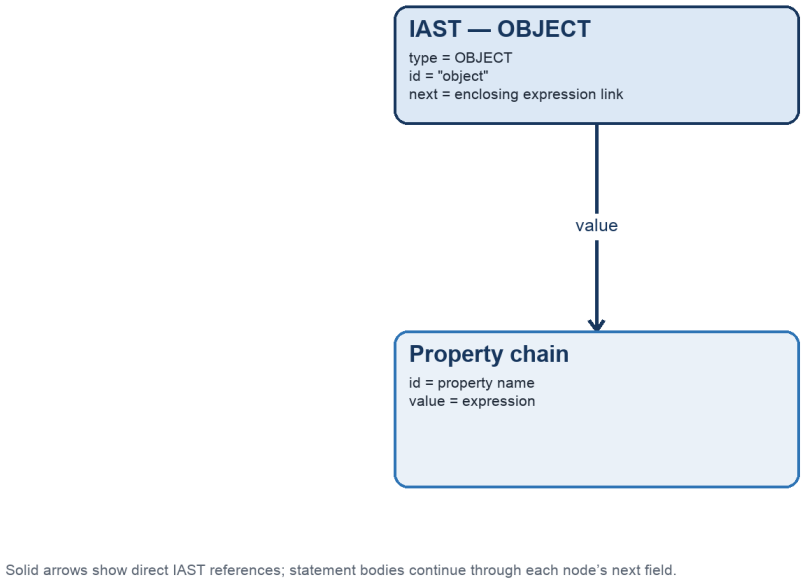

Figure 18. Compiler-produced IAST layout for OBJECT.

Field contract

Field	Meaning
`type`	OBJECT.
`id`	The literal identifier object.
`value`	First property descriptor; further descriptors are linked through next.
`property.id`	The property name, including $ when the source uses a bound variable name.
`property.value`	The compiled expression used to initialise the property.
`next`	Used by the enclosing expression or argument list.

Implementation note. Property descriptors have no independent byte-code type. Runtime evaluation walks the value chain and creates each property from the descriptor's id and value fields.

Source basis: `YASSCompiler.compileObject(); ZPERuntimeEnvironment.evaluateObjectLiteral(IAST, …)`

NODE 19 / COMPILED STRUCTURE

VAR

Represents a variable reference or assignment target, identified by its compiler-normalised name.

```
$customer
```

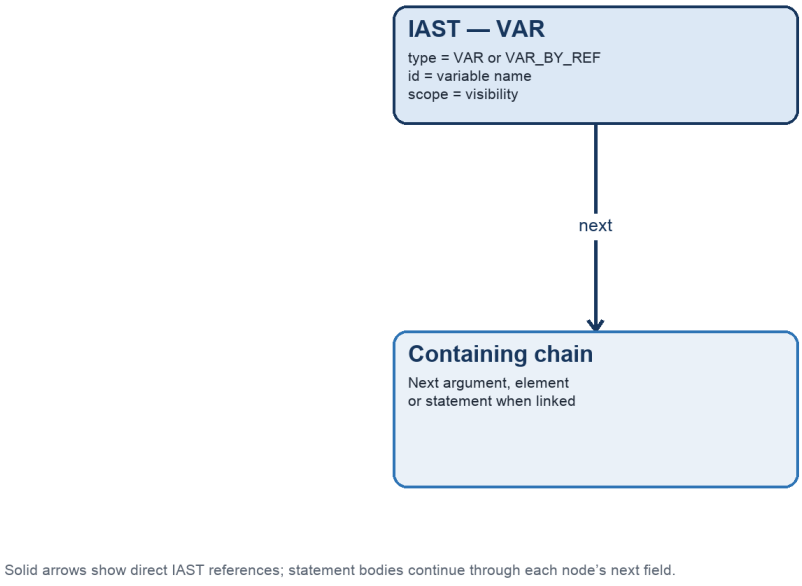

Figure 19. Compiler-produced IAST layout for VAR.

Field contract

Field	Meaning
`type`	VAR for ordinary access; VAR_BY_REF for a reference-preserving access.
`id`	The variable identifier. Bound variables retain the leading $ character.
`scope`	PUBLIC, PRIVATE or PROTECTED when visibility is relevant.
`next`	Optional link supplied by the containing argument, element or statement chain.

Implementation note. Indexing, calls and member access do not mutate the VAR node. The compiler wraps it inside INDEX_ACCESSOR, LAMBDA_CALL or ARROW_OPERATOR nodes.

Source basis: `YASSCompiler.compileVar(); ZPERuntimeEnvironment.evaluateVar(IAST, …)`

NODE 20 / COMPILED STRUCTURE

IDENTIFIER

Represents a named function or predefined-command invocation together with its argument chain.

```
calculate_total($values)
```

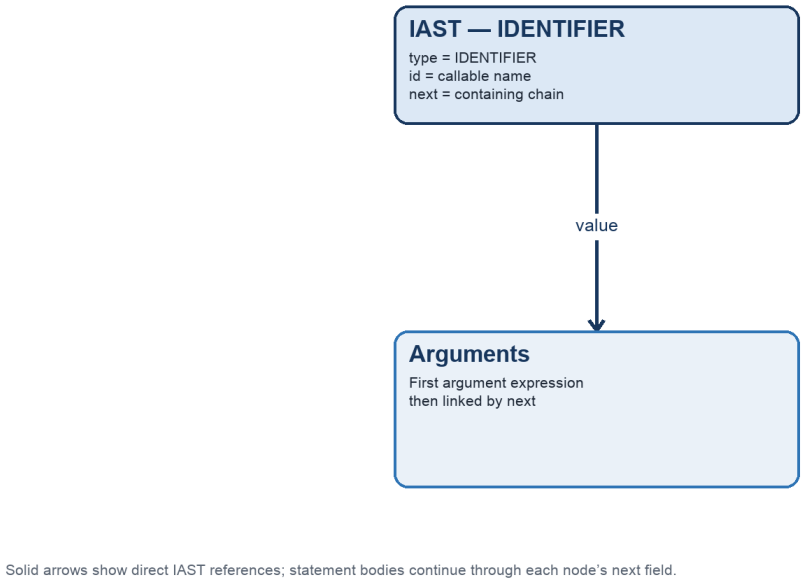

Figure 20. Compiler-produced IAST layout for IDENTIFIER.

Field contract

Field	Meaning
`type`	IDENTIFIER.
`id`	The source callable name, possibly replaced by an internal alias.
`value`	First compiled argument expression; an empty ARGUMENTS sentinel when the call has no arguments.
`next`	Optional link in the surrounding expression or argument sequence.

Implementation note. Static calls, indexing and object access wrap or retag this base layout. Ordinary resolution occurs at runtime against user functions and predefined commands.

Source basis: `YASSCompiler.compileIdentifier(); ZPERuntimeEnvironment.evaluateFunction(IAST)`

NODE 21 / COMPILED STRUCTURE

INDEX_ACCESSOR

Represents indexed or ranged access to a list, map, string, tuple or nested indexed result.

```
$matrix[$row][$column]
```

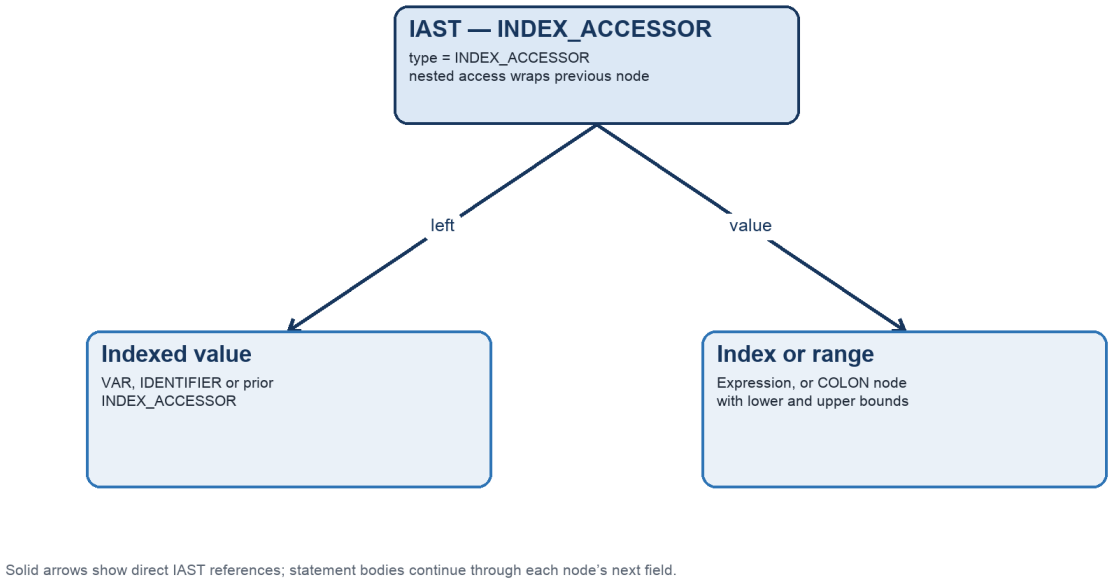

Figure 21. Compiler-produced IAST layout for INDEX_ACCESSOR.

Field contract

Field	Meaning
`type`	INDEX_ACCESSOR.
`left`	The value being indexed. For multidimensional access this is the previous accessor.
`value`	The index expression.
`value.left`	Lower-bound expression when value is a COLON range node.
`value.middle`	Upper-bound expression when value is a COLON range node.
`next`	Optional link in the enclosing expression or argument chain.

Implementation note. Successive brackets build outward: each new INDEX_ACCESSOR stores the previously compiled accessor in left, producing a naturally nested access path.

Source basis: `YASSCompiler.compileVar(); YASSCompiler.compileIdentifier(); ZPERuntimeEnvironment.evaluateIndexAccessor(IAST, …)`

NODE 22 / COMPILED STRUCTURE

ARROW_OPERATOR

Represents chained access to object properties or methods from a parent expression, including parent-dispatched access through super.

```
$queue->enqueue($item)     super->describe()
```

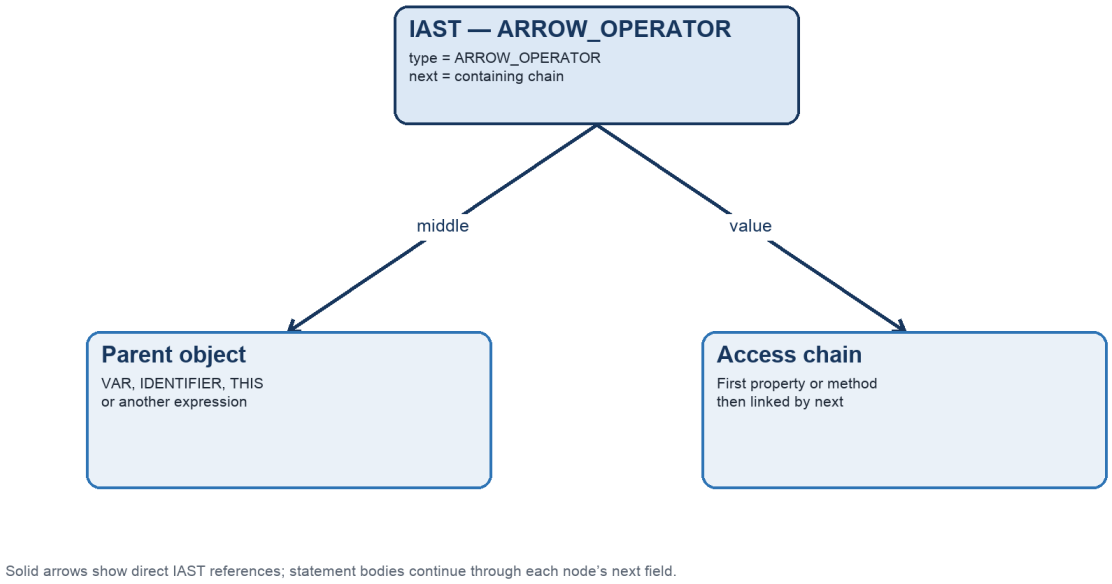

Figure 22. Compiler-produced IAST layout for ARROW_OPERATOR.

Field contract

Field	Meaning
`type`	ARROW_OPERATOR.
`middle`	The parent expression expected to evaluate to a ZPEObject. For super access, this is a THIS node with id set to super.
`value`	First property, method call or lambda-call descriptor in the access chain.
`value.next`	Subsequent chained accesses.
`next`	Optional link in the surrounding expression or statement sequence.

Implementation note. All successive arrow segments are normalised into one value-linked access chain. For super->method(), middle contains a THIS node whose id is super; runtime then resolves the first method against the parent structure of the current object rather than redispatching to the override.

Source basis: `YASSCompiler.compileVar(); YASSCompiler.compileIdentifier(); ZPERuntimeEnvironment.accessorEvaluate(IAST)`

NODE 23 / COMPILED STRUCTURE

DOT

Represents a chainable native-member operation or record-field access using dot syntax.

```
$text.to_uppercase()
```

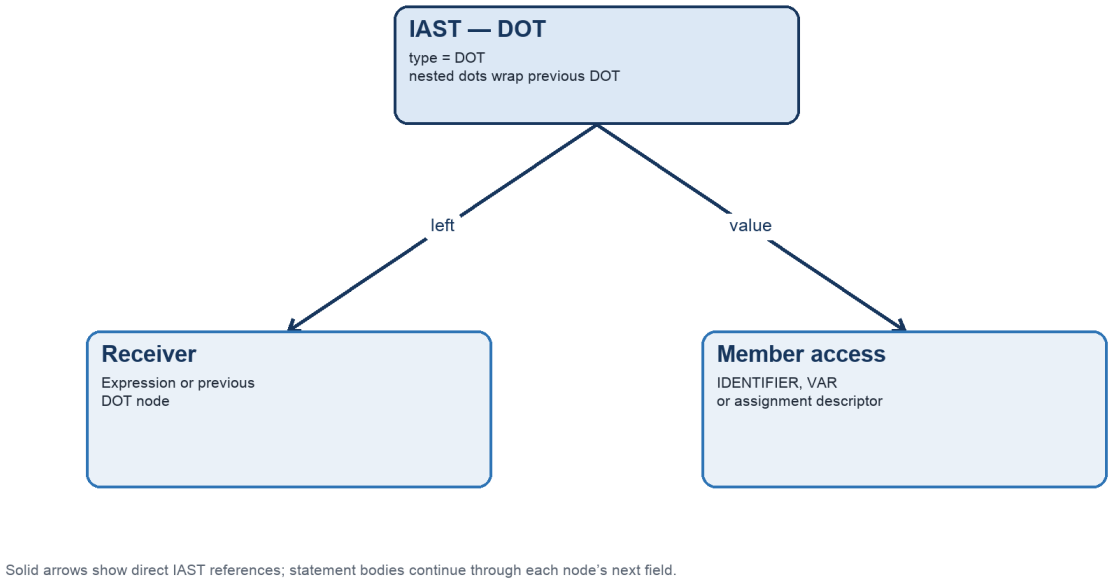

Figure 23. Compiler-produced IAST layout for DOT.

Field contract

Field	Meaning
`type`	DOT.
`left`	The receiver expression or previously compiled DOT chain.
`value`	A method-call IDENTIFIER, field VAR, or compiled field assignment.
`next`	Optional link in the enclosing expression or statement sequence.

Implementation note. Repeated dots are represented by nesting rather than a flat member list. Runtime evaluation passes the result of each DOT node into the outer receiver.

Source basis: `YASSCompiler.compileDotExpression(IAST); ZPERuntimeEnvironment.evaluateDot(IAST, …)`

NODE 24 / COMPILED STRUCTURE

NEW

Represents construction of a record, user structure, native object or module-qualified structure.

```
new stdAlgorithms::queue()
```

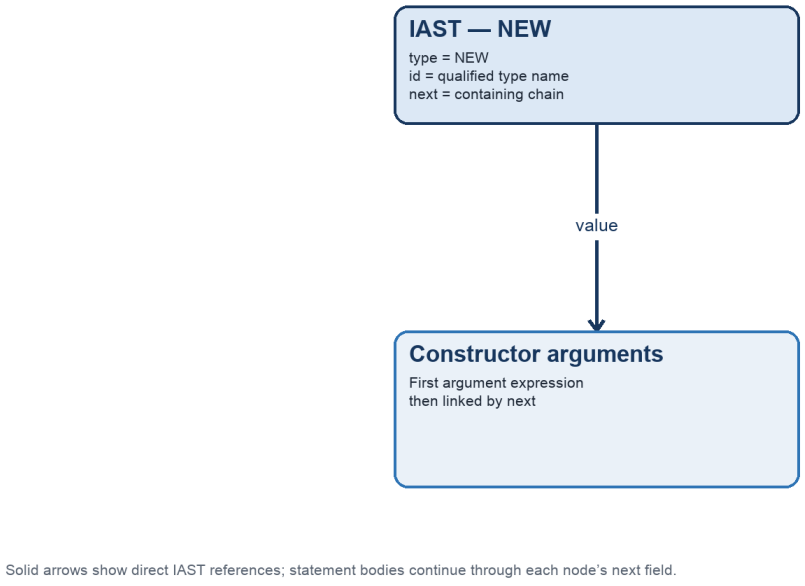

Figure 24. Compiler-produced IAST layout for NEW.

Field contract

Field	Meaning
`type`	NEW.
`id`	The structure name, including namespace paths or module qualification where supplied.
`value`	First constructor argument; an ARGUMENTS sentinel for an empty constructor call.
`next`	Optional link in the enclosing expression or argument sequence.

Implementation note. The AST does not encode which construction route will be used. Runtime lookup tries records, user structures, module structures, built-ins and imported structures.

Source basis: `YASSCompiler.compileNewStructure(); ZPERuntimeEnvironment.evaluateNew(IAST, …)`

NODE 25 / COMPILED STRUCTURE

TUPLE

Represents either a tuple type definition or population of a named tuple instance.

```
tuple coordinate(number, number) / coordinate(10, 20)
```

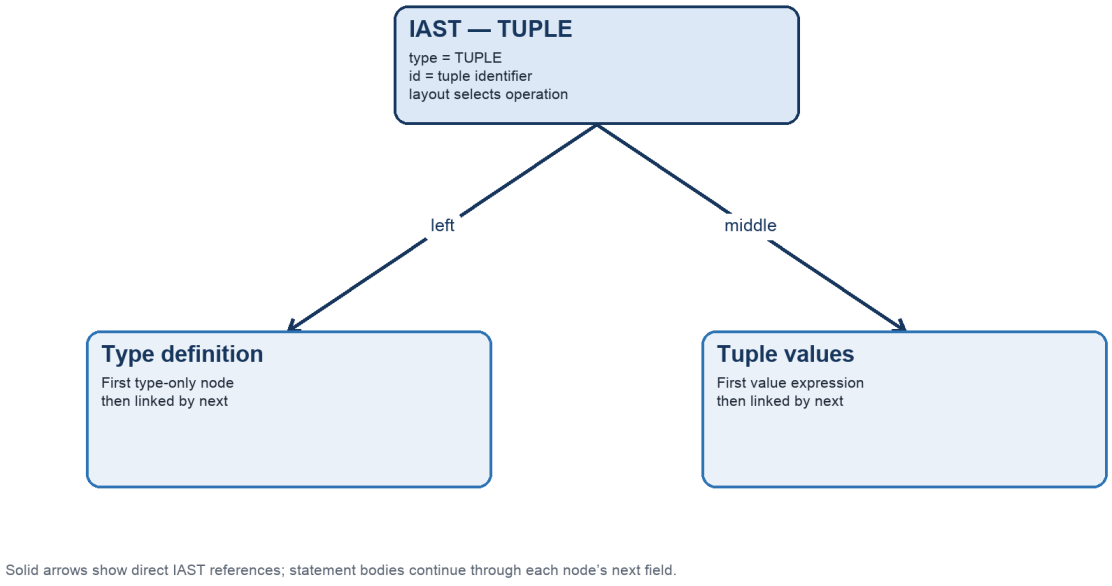

Figure 25. Compiler-produced IAST layout for TUPLE.

Field contract

Field	Meaning
`type`	TUPLE.
`id`	The tuple identifier.
`left`	First type node when defining a tuple shape; otherwise null.
`middle`	First value expression when populating an existing tuple; otherwise null.
`next`	Optional link in the containing definition or expression sequence.

Implementation note. The compiler uses mutually exclusive left and middle chains to distinguish definition from population without introducing separate tuple byte codes.

Source basis: `YASSCompiler.compileTuple(String); ZPERuntimeEnvironment.evaluateTuple(IAST, …)`

NODE 26 / COMPILED STRUCTURE

CAST

Represents explicit conversion of one evaluated value to a requested YASS type.

```
cast($count, string)
```

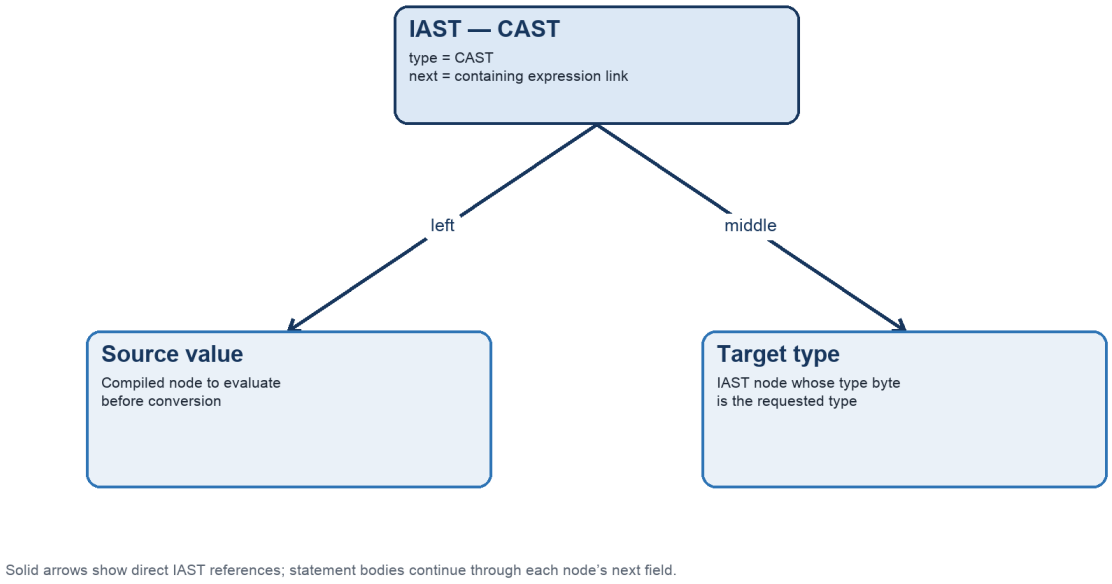

Figure 26. Compiler-produced IAST layout for CAST.

Field contract

Field	Meaning
`type`	CAST.
`left`	The source node to evaluate.
`middle`	A metadata IAST whose type field is the requested destination type.
`next`	Optional link in the enclosing expression or argument chain.

Implementation note. The target type is carried in middle.type rather than middle.value. Runtime evaluation passes that byte directly to the central type-casting helper.

Source basis: `YASSCompiler.compileCast(); ZPERuntimeEnvironment.evaluateExpression(IAST, …)`

NODE 27 / COMPILED STRUCTURE

NEGATION

Represents logical or numeric inversion of one compiled operand.

```
not($is_valid)
```

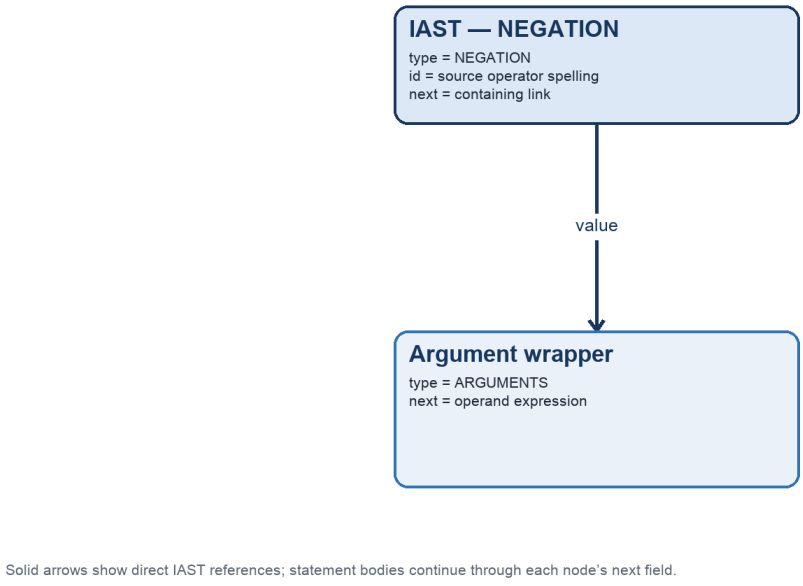

Figure 27. Compiler-produced IAST layout for NEGATION.

Field contract

Field	Meaning
`type`	NEGATION.
`id`	The source operator spelling, such as not or !.
`value`	An ARGUMENTS wrapper node.
`value.next`	The expression or node whose result is to be inverted.
`next`	Optional link in the surrounding expression or argument sequence.

Implementation note. Unlike most single-operand nodes, NEGATION retains an ARGUMENTS wrapper. Runtime argument generation evaluates the operand before inverting a number or boolean.

Source basis: `YASSCompiler.compileNegation(); ZPERuntimeEnvironment.evaluateNegation(IAST)`

NODE 28 / COMPILED STRUCTURE

NEGATIVE

Represents unary minus applied to a variable or numeric literal.

```
-$offset
```

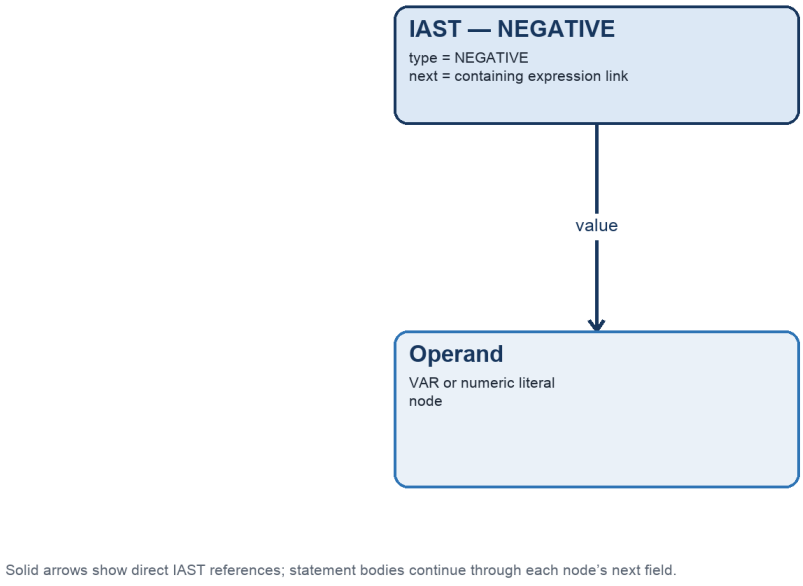

Figure 28. Compiler-produced IAST layout for NEGATIVE.

Field contract

Field	Meaning
`type`	NEGATIVE.
`value`	The variable node or numeric literal node to negate.
`next`	Optional link in the enclosing expression or argument chain.

Implementation note. NEGATIVE is distinct from NEGATION: it is produced specifically for unary minus. Literal text is converted to a numeric value during compilation.

Source basis: `YASSCompiler.compileNegative(); ZPERuntimeEnvironment.evaluateNegative(IAST, …)`

NODE 29 / COMPILED STRUCTURE

EXPRESSION

Wraps a non-logical expression tree and may carry an optimised compiled-expression sidecar.

```
$subtotal + ($tax * $rate)
```

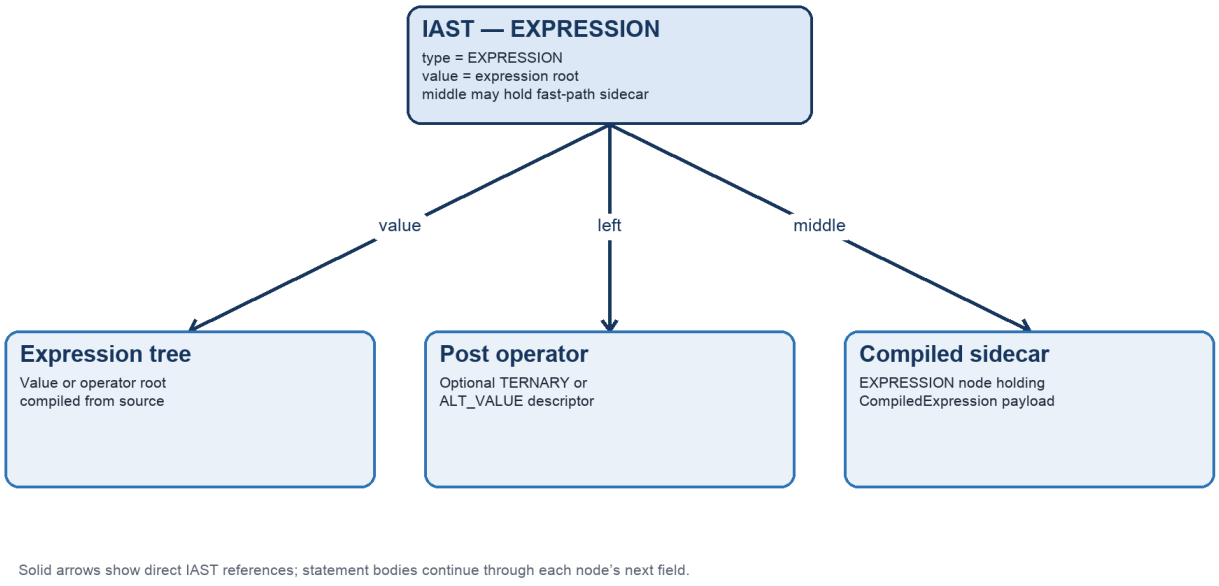

Figure 29. Compiler-produced IAST layout for EXPRESSION.

Field contract

Field	Meaning
`type`	EXPRESSION when the root is not classified as logical/comparative.
`value`	Root node of the ordinary expression tree; may be replaced by a folded literal.
`left`	Optional TERNARY or ALT_VALUE descriptor applied after the main result.
`middle`	Optional sidecar IAST whose value is a tokenised CompiledExpression.
`next`	Optional link in an enclosing argument, element or statement chain.

Implementation note. The ordinary tree remains the semantic representation. The sidecar is attached only when tokenisation succeeds and supplies a faster evaluation route without changing meaning.

Source basis: `YASSCompiler.compileExpression(boolean); YASSCompiler.attachCompiledExpression(IAST); LAMEX3`

NODE 30 / COMPILED STRUCTURE

LOGICAL_EXPRESSION

Wraps a comparison or logically joined condition used by branching and loop constructs.

```
$age >= 18 && $registered
```

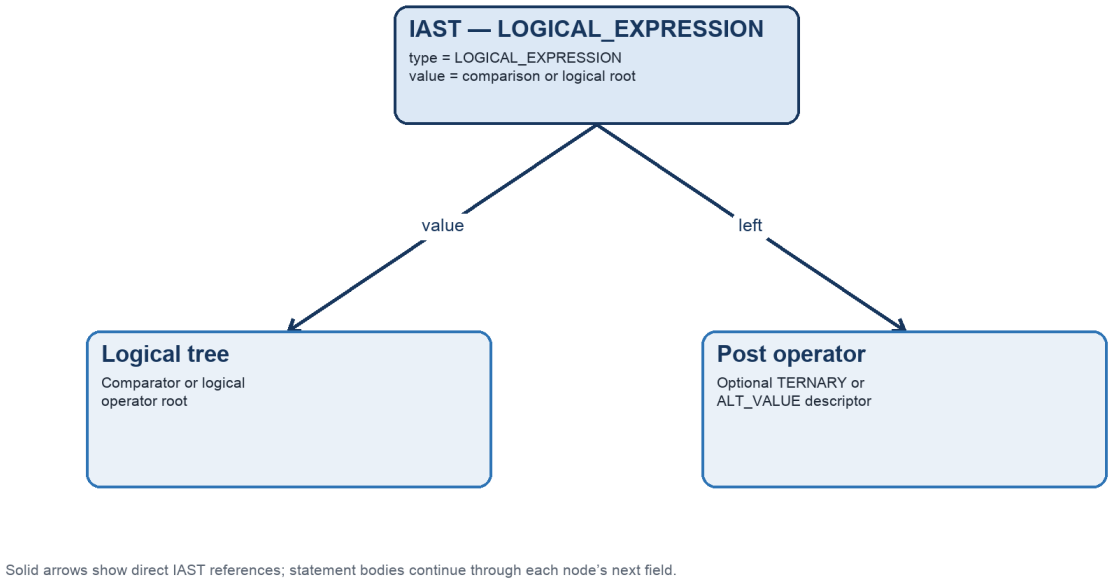

Figure 30. Compiler-produced IAST layout for LOGICAL_EXPRESSION.

Field contract

Field	Meaning
`type`	LOGICAL_EXPRESSION when comparison or logical operators are present.
`value`	Comparator node or logical-operator tree.
`value.left`	Left operand of a comparator/operator.
`value.middle`	Right operand of a comparator.
`value.next`	Right logical branch for AND/OR-style operator nodes.
`left`	Optional post-expression descriptor.

Implementation note. Logical joins store their first branch in the operator's left field and the recursively compiled remainder in next. Comparators instead use left and middle as operands.

Source basis: `YASSCompiler.compileLogicallyJoinedExpression(boolean, IAST);` `LAMEX3.evaluateExpressions(IAST)`

NODE 31 / COMPILED STRUCTURE

TERNARY

Describes the two result branches applied after evaluation of a parent expression.

```
$passed ? "Pass" : "Fail"
```

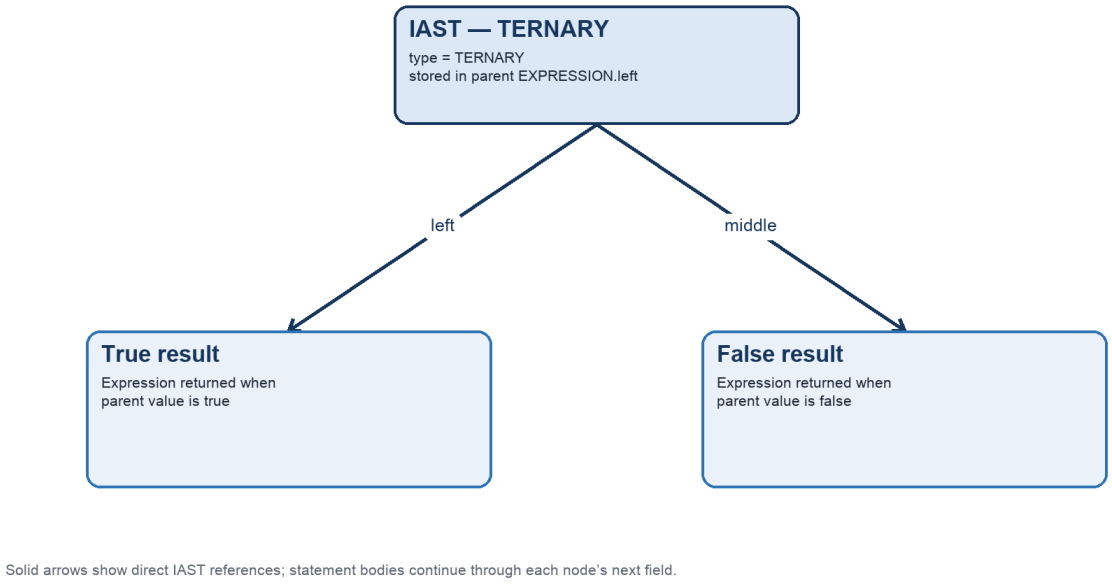

Figure 31. Compiler-produced IAST layout for TERNARY.

Field contract

Field	Meaning
`type`	TERNARY.
`left`	The true-branch result expression.
`middle`	The false-branch result expression.
`next`	Normally null; the descriptor is owned through its parent expression's left field.

Implementation note. The condition is not duplicated inside TERNARY. It is the main value of the parent EXPRESSION, and LAMEX3 selects one of this descriptor's two branches afterward.

Source basis: `YASSCompiler.compileExpression(boolean); LAMEX3.evaluateTernary(IAST, …)`

NODE 32 / COMPILED STRUCTURE

ALT_VALUE

Describes a fallback result used when the parent expression evaluates to null or undefined.

```
$possibly_undefined ?? "fallback"
```

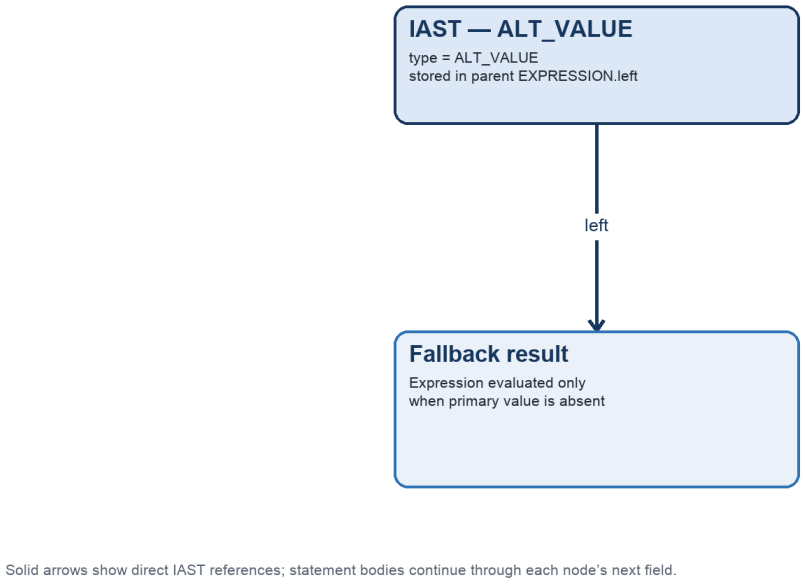

Figure 32. Compiler-produced IAST layout for ALT_VALUE.

Field contract

Field	Meaning
`type`	ALT_VALUE.
`left`	The fallback expression.
`next`	Normally null; ownership is through the parent expression's left field.

Implementation note. The primary value belongs to the parent EXPRESSION. Runtime evaluation returns it unchanged unless it is null or Undefined, in which case left is evaluated.

Source basis: `YASSCompiler.compileExpression(boolean); LAMEX3.evaluateAltValue(IAST, …)`

NODE 33 / COMPILED STRUCTURE

LAMBDA_CALL

Represents invocation of a function held in a variable rather than resolved by a fixed name.

```
$transform($value)
```

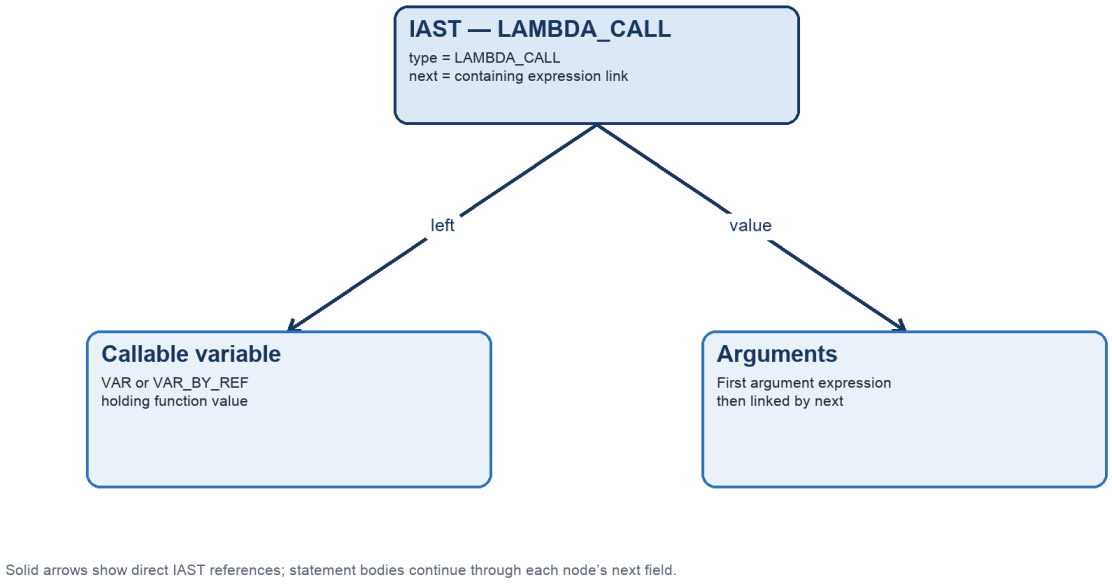

Figure 33. Compiler-produced IAST layout for LAMBDA_CALL.

Field contract

Field	Meaning
`type`	LAMBDA_CALL.
`left`	The variable node containing a FunctionReference or ZPEFunction.
`value`	First argument expression, or an empty ARGUMENTS sentinel.
`next`	Optional link in the surrounding expression or statement sequence.

Implementation note. This node is emitted when brackets follow a compiled variable. Runtime distinguishes a lightweight FunctionReference from a full ZPEFunction before invocation.

Source basis: `YASSCompiler.compileVar(); ZPERuntimeEnvironment.evaluateLambdaCall(IAST, …)`

NODE 34 / COMPILED STRUCTURE

SCOPE_RESOLUTION

Represents a member call made with ::, including a public static function resolved directly against an abstract structure.

```
PersonObject::get_people_list()
```

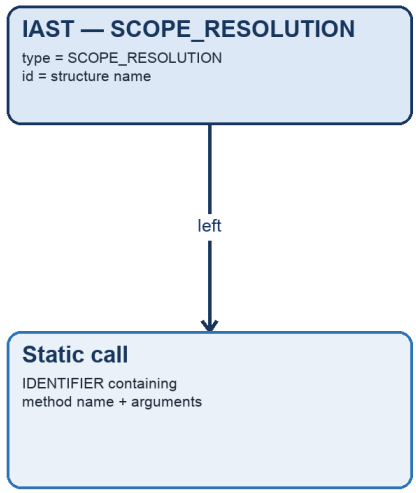

Figure 34. Compiler-produced IAST layout for SCOPE_RESOLUTION.

Field contract

Field	Meaning
`type`	SCOPE_RESOLUTION.
`id`	The structure identifier on the left of ::.
`left`	An IDENTIFIER node naming the static function; its value contains the compiled arguments.
`next`	Optional link in the enclosing expression or statement sequence.

Implementation note. The structure name remains in id and the called function is retained as the IDENTIFIER node in left. Runtime resolves the AbstractStructure, requires a public static function with that name, evaluates left.value as its argument list and invokes it without constructing an object instance.

Source basis: `YASSCompiler.compileIdentifier(); ZPERuntimeEnvironment.handleScopeResolution(IAST, …)`

NODE 35 / COMPILED STRUCTURE

TYPE

Represents a request for the runtime YASS type name of an evaluated value.

```
type($value)
```

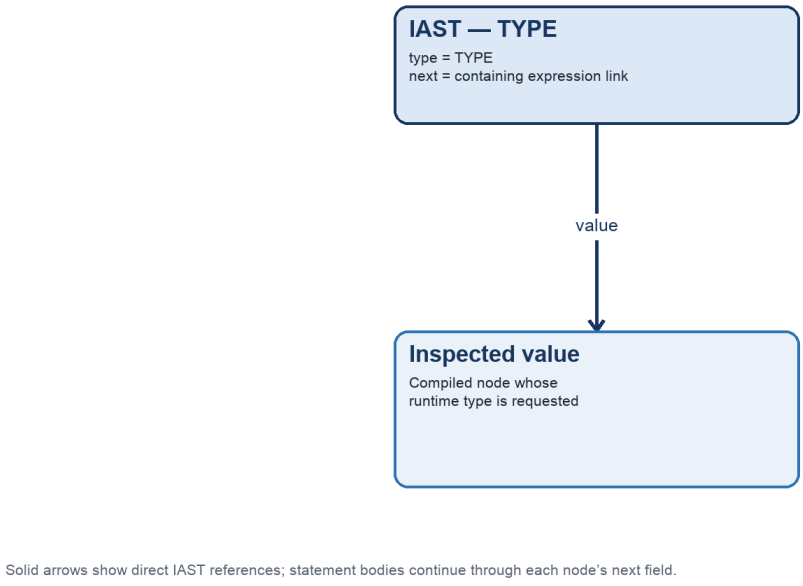

Figure 35. Compiler-produced IAST layout for TYPE.

Field contract

Field	Meaning
`type`	TYPE.
`value`	The node to evaluate and classify.
`next`	Optional link in the enclosing expression or argument chain.

Implementation note. Both type(...) and typeof syntax compile to this shape. The result is a YASS string such as STRING, OBJECT, RECORD, NUMBER or BOOLEAN.

Source basis: `YASSCompiler.compileType(); ZPERuntimeEnvironment.evaluateTypeName(IAST, …)`

NODE 36 / COMPILED STRUCTURE

EMPTY

Represents an emptiness test, unless the compiler can replace it with a constant boolean.

```
empty($items)
```

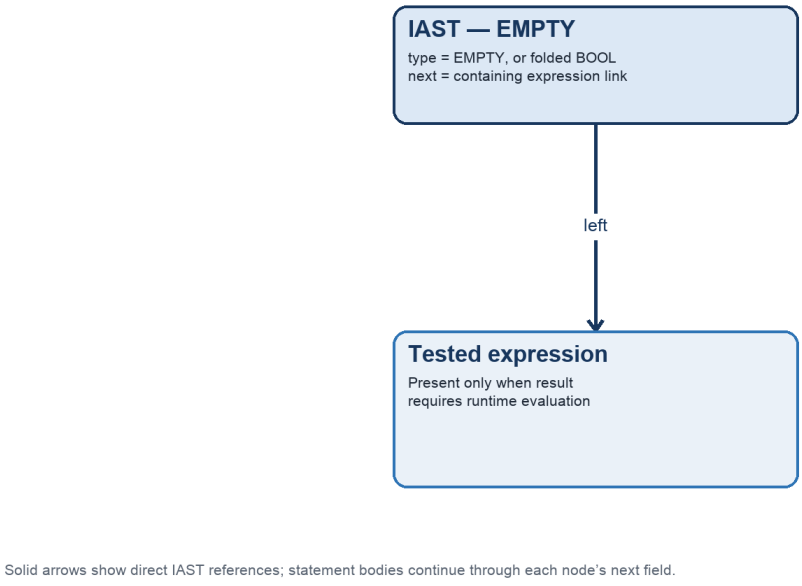

Figure 36. Compiler-produced IAST layout for EMPTY.

Field contract

Field	Meaning
`type`	EMPTY for a dynamic test; BOOL after successful constant folding.
`left`	The expression to test when evaluation must occur at runtime.
`value`	The folded boolean when the operand is statically known; otherwise null.
`next`	Optional link in the enclosing expression or argument chain.

Implementation note. This is an example of a compiler rewrite changing node type and shape. A statically known empty/non-empty literal discards left and stores the boolean directly in value.

Source basis: `YASSCompiler.compileEmpty(); ZPERuntimeEnvironment.evaluateEmpty(IAST)`

NODE 37 / COMPILED STRUCTURE

FROM

Represents inline range generation with an optional mapping function or arrow-expression predicate.

```
from 0 to 100 : ($value) => $value % 2 == 0
```

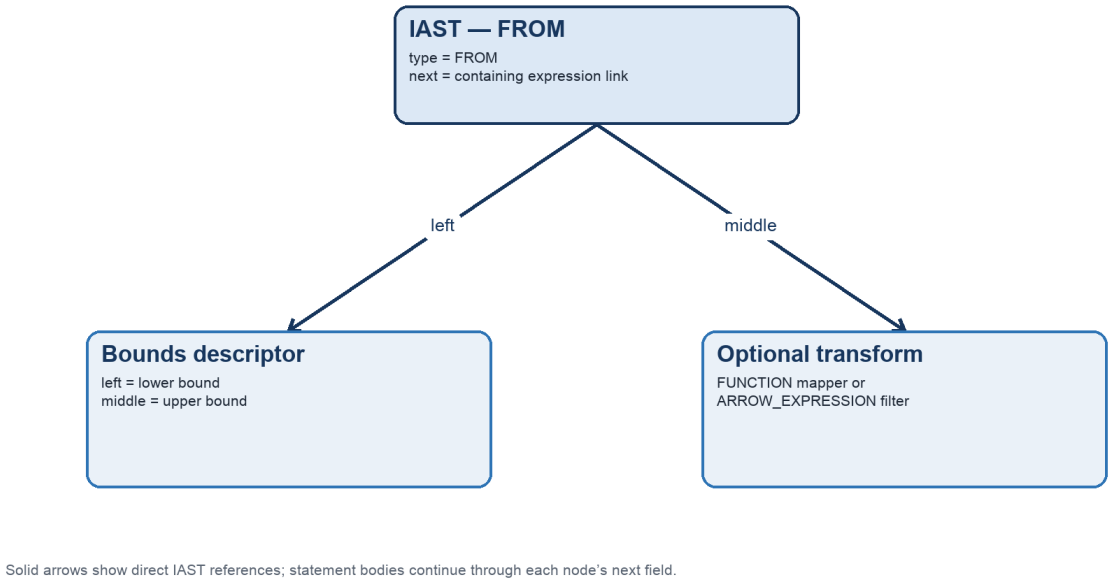

Figure 37. Compiler-produced IAST layout for FROM.

Field contract

Field	Meaning
`type`	FROM.
`left`	A bounds descriptor node.
`left.left`	The lower-bound expression.
`left.middle`	The exclusive upper-bound expression.
`middle`	Optional anonymous function mapper or arrow-expression predicate.
`next`	Optional link in the enclosing expression or argument chain.

Implementation note. The runtime distinguishes mapping from filtering by the middle node type. A FUNCTION produces one output per input; an ARROW_EXPRESSION retains inputs whose predicate is true.

Source basis: `YASSCompiler.compileInlineIteration(); ZPERuntimeEnvironment.evaluateFrom(IAST, …)`

NODE 38 / COMPILED STRUCTURE

TEMPLATE

Represents a compiled template fragment produced by the dedicated YASS template compiler.

```
template … embedded YASS content … template
```

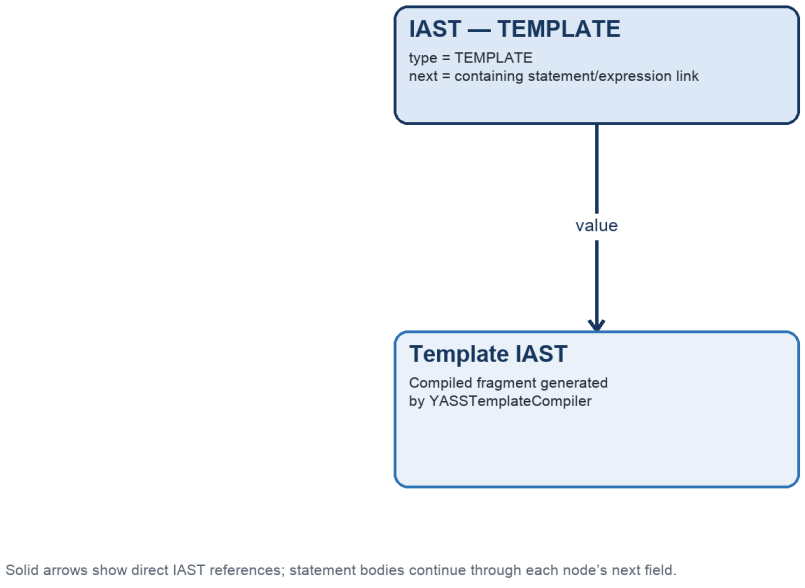

Figure 38. Compiler-produced IAST layout for TEMPLATE.

Field contract

Field	Meaning
`type`	TEMPLATE.
`value`	The root/sequence generated by YASSTemplateCompiler.
`next`	Optional link in the containing statement or expression sequence.

Implementation note. Template source is parsed by a specialised compiler after inheriting known globals from the main compiler. Runtime evaluation delegates the stored fragment to template handling.

Source basis: `YASSCompiler.compileTemplate(); YASSTemplateCompiler; ZPEFunction.evaluateTemplate(IAST, …)`

NODE 39 / COMPILED STRUCTURE

IS_SET

Represents a non-mutating test for whether a variable, indexed element or object member is defined.

```
is_set($options[$key])
```

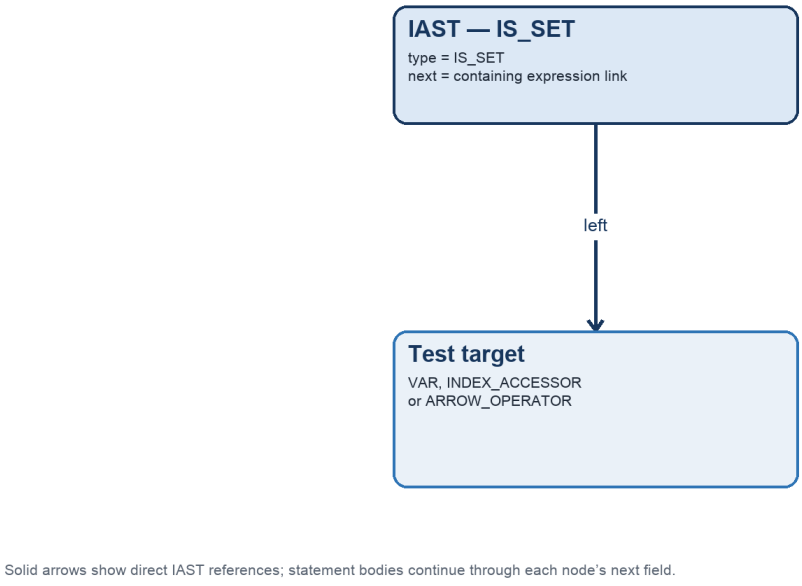

Figure 39. Compiler-produced IAST layout for IS_SET.

Field contract

Field	Meaning
`type`	IS_SET.
`left`	The variable or access path whose defined state is requested.
`next`	Optional link in the enclosing expression or argument chain.

Implementation note. Runtime handling specialises indexed targets so list/string bounds and map-key existence can be checked without first performing a failing access.

Source basis: `YASSCompiler.compileIsSet(); ZPERuntimeEnvironment.isSet(IAST)`

NODE 40 / COMPILED STRUCTURE

UNSET

Represents removal of a variable or indexed collection element.

```
unset($cache[$key])
```

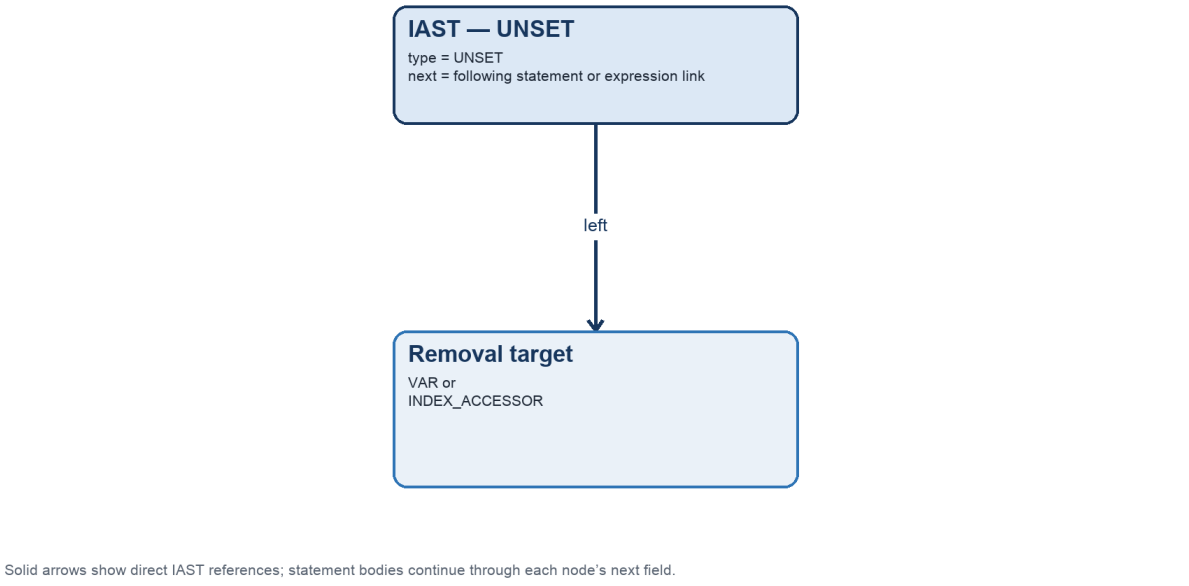

Figure 40. Compiler-produced IAST layout for UNSET.

Field contract

Field	Meaning
`type`	UNSET.
`left`	The variable or indexed element to remove.
`next`	The following statement or enclosing expression-chain link.

Implementation note. For a plain variable the runtime removes the property from the current function. For an index it mutates the existing list or map and returns a boolean result.

Source basis: `YASSCompiler.compileUnset(); ZPERuntimeEnvironment.unset(IAST)`

NODE 41 / COMPILED STRUCTURE

COUNT

Represents a request for the number of elements in a supported value, unless folded to a literal.

```
count($values)
```

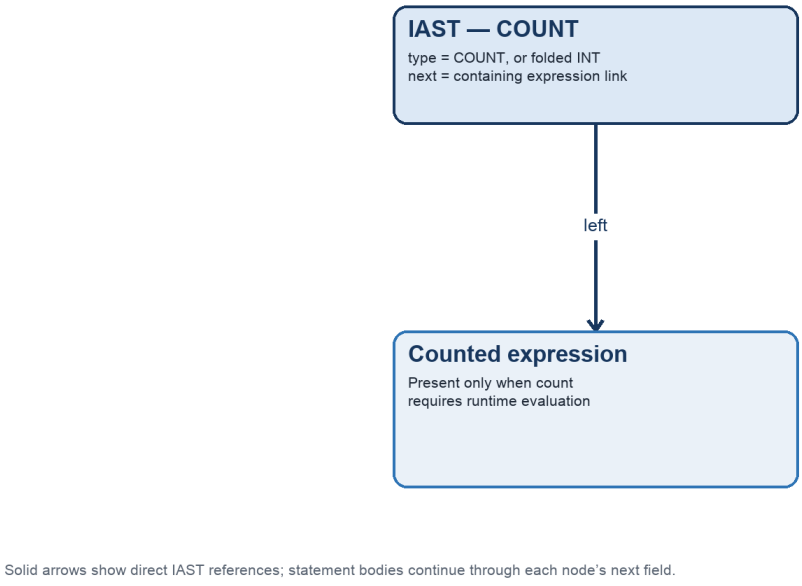

Figure 41. Compiler-produced IAST layout for COUNT.

Field contract

Field	Meaning
`type`	COUNT for a dynamic operation; INT after compile-time folding.
`left`	The expression whose elements are counted at runtime.
`value`	The folded long integer when the count is statically known.
`next`	Optional link in the enclosing expression or argument chain.

Implementation note. Like EMPTY, COUNT can change shape during compilation. A statically countable literal becomes an INT node and discards the original left expression.

Source basis: `YASSCompiler.compileCount(); ZPERuntimeEnvironment.countAction(IAST)`

NODE 42 / COMPILED STRUCTURE

ASSERT

Represents a test assertion together with source text and line metadata used for diagnostics.

```
assert($actual == $expected)
```

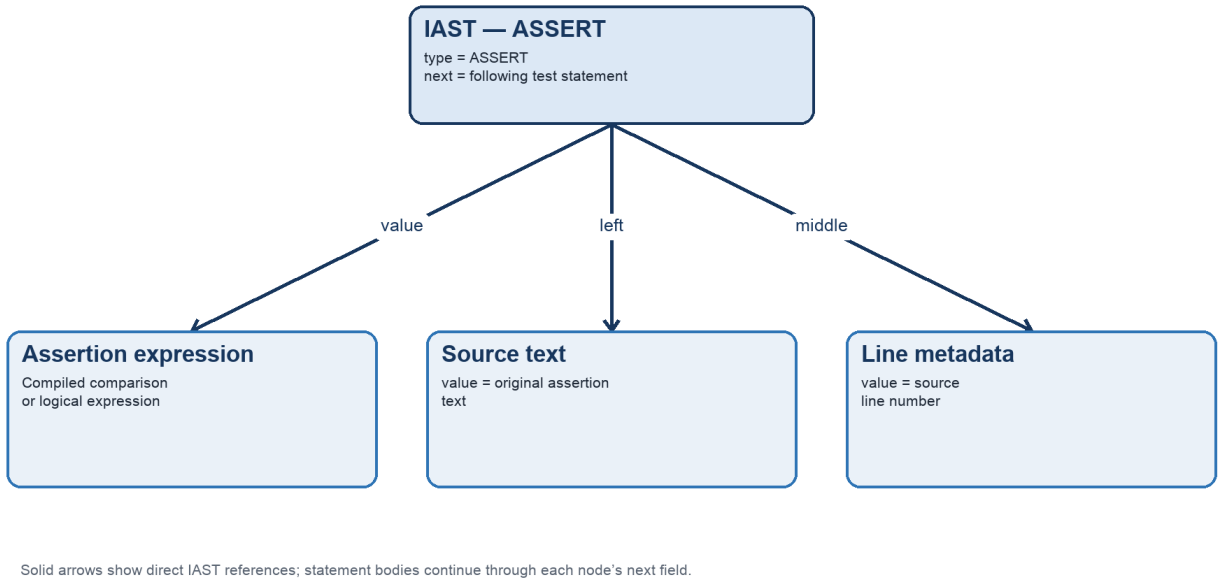

Figure 42. Compiler-produced IAST layout for ASSERT.

Field contract

Field	Meaning
`type`	ASSERT.
`value`	The compiled expression whose truth is asserted.
`left`	Metadata node containing the original source substring in value.
`middle`	Metadata node containing the source line number in value.
`next`	The following statement in the test body.

Implementation note. The extra metadata does not affect truth evaluation. It allows debugging/test mode to report which original assertion passed or failed and where it appeared.

Source basis: `YASSCompiler.compileAssertion(); ZPERuntimeEnvironment.handleAssert(IAST, …)`

NODE 43 / COMPILED STRUCTURE

BREAK

Represents an instruction to terminate traversal of the nearest active loop.

```
break
```

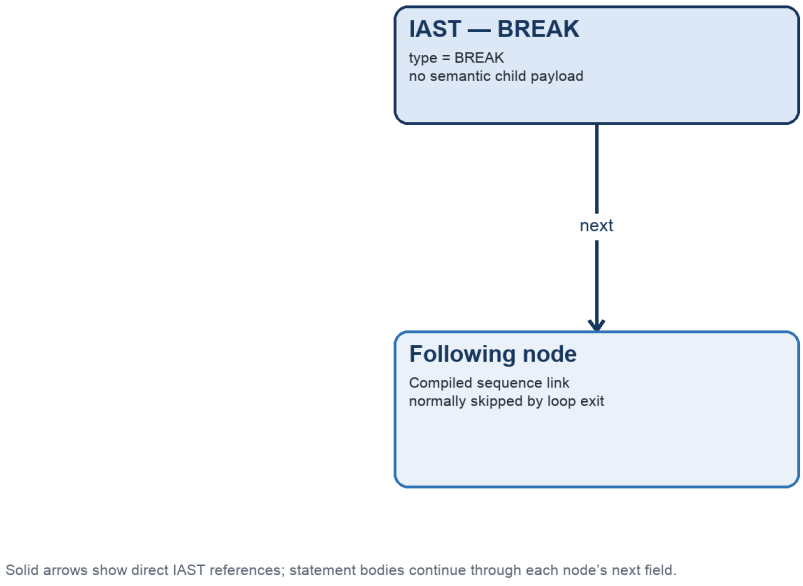

Figure 43. Compiler-produced IAST layout for BREAK.

Field contract

Field	Meaning
`type`	BREAK.
`next`	The following compiled statement; loop break state prevents ordinary traversal to it.

Implementation note. BREAK carries no target or depth in its own node. Runtime traversal sets break state on the current function, and the active loop consumes that state.

Source basis: `YASSCompiler.compileBreak(); ZPERuntimeEnvironment.constructTraverse(IAST, …)`

NODE 44 / COMPILED STRUCTURE

INCREMENT

Represents compound numeric addition of an assignable target and a right-hand expression.

```
$total += $value
```

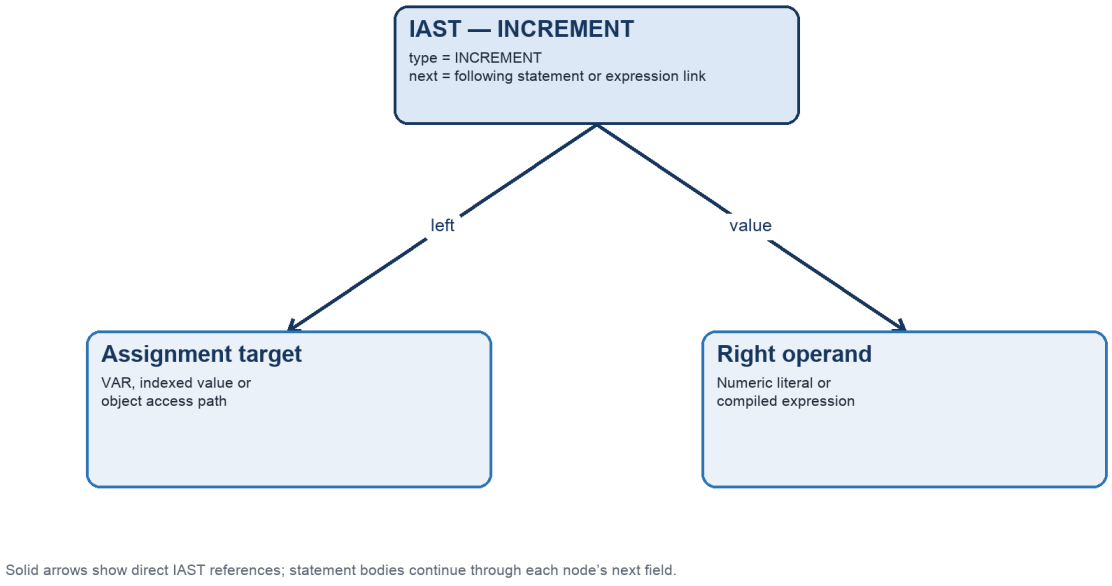

Figure 44. Compiler-produced IAST layout for INCREMENT.

Field contract

Field	Meaning
`type`	INCREMENT.
`left`	The assignable numeric target.
`value`	The expression whose numeric result is applied to the target.
`next`	The following statement or enclosing expression-chain link.

Implementation note. The compiler normalises this compound operator into a dedicated node instead of building a separate read, arithmetic expression and assignment. Runtime target resolution preserves variables, indexes and object properties, then applies addition.

Source basis: `YASSCompiler.compileNumericAssignment(IAST);`
`ZPERuntimeEnvironment.applyNumericAssignment(IAST, …)`

NODE 45 / COMPILED STRUCTURE

DECREMENT

Represents compound numeric subtraction of an assignable target and a right-hand expression.

```
$remaining -= 1
```

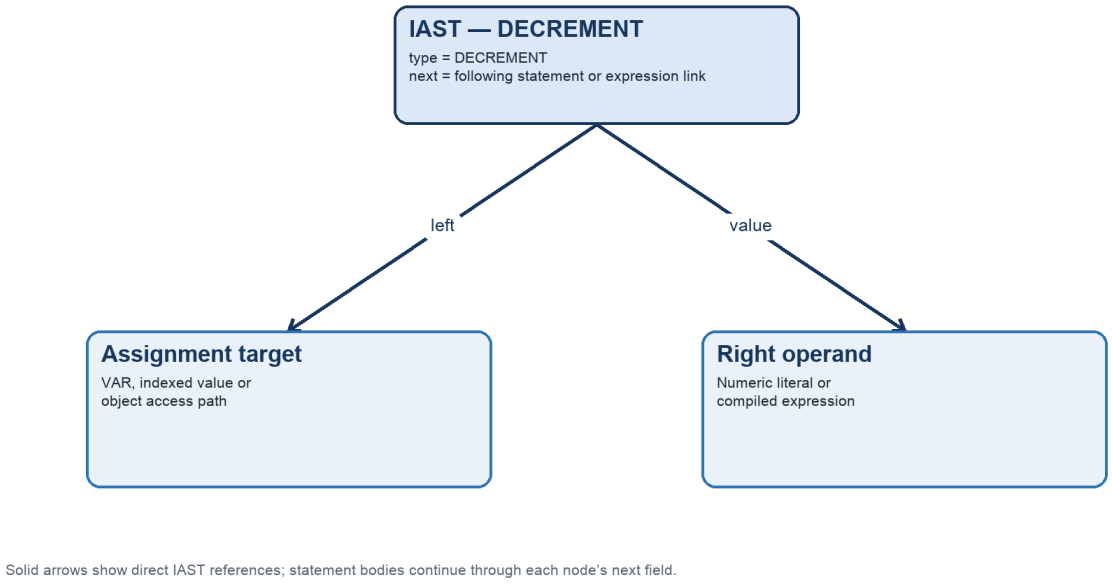

Figure 45. Compiler-produced IAST layout for DECREMENT.

Field contract

Field	Meaning
`type`	DECREMENT.
`left`	The assignable numeric target.
`value`	The expression whose numeric result is applied to the target.
`next`	The following statement or enclosing expression-chain link.

Implementation note. The compiler normalises this compound operator into a dedicated node instead of building a separate read, arithmetic expression and assignment. Runtime target resolution preserves variables, indexes and object properties, then applies subtraction.

Source basis: `YASSCompiler.compileNumericAssignment(IAST);`
`ZPERuntimeEnvironment.applyNumericAssignment(IAST, …)`

NODE 46 / COMPILED STRUCTURE

MULTI_ASSIGN

Represents compound numeric multiplication of an assignable target and a right-hand expression.

```
$scale *= 2
```

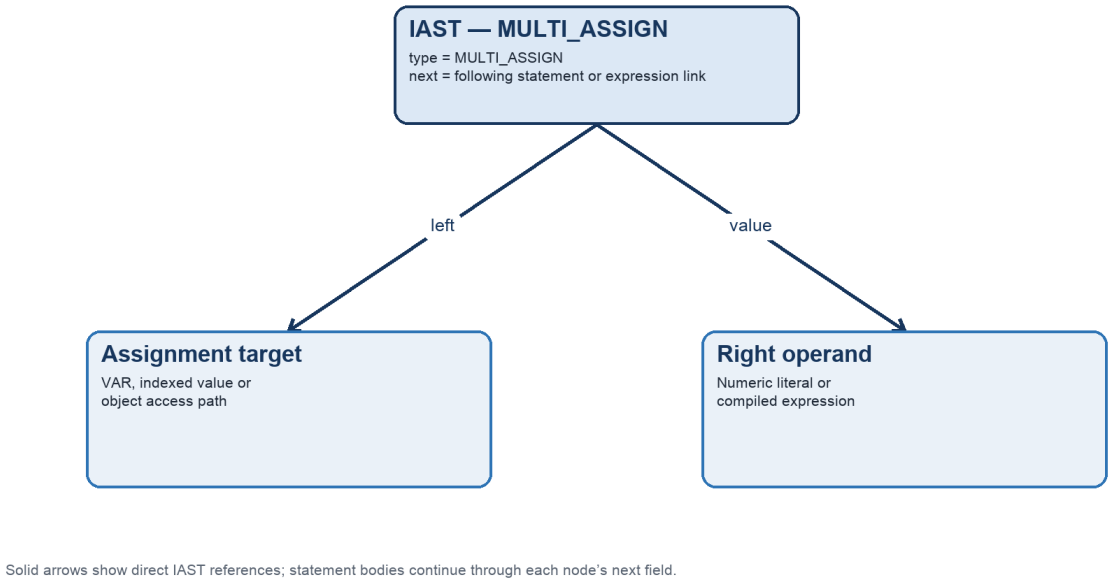

Figure 46. Compiler-produced IAST layout for MULTI_ASSIGN.

Field contract

Field	Meaning
`type`	MULTI_ASSIGN.
`left`	The assignable numeric target.
`value`	The expression whose numeric result is applied to the target.
`next`	The following statement or enclosing expression-chain link.

Implementation note. The compiler normalises this compound operator into a dedicated node instead of building a separate read, arithmetic expression and assignment. Runtime target resolution preserves variables, indexes and object properties, then applies multiplication.

Source basis: `YASSCompiler.compileNumericAssignment(IAST);`
`ZPERuntimeEnvironment.applyNumericAssignment(IAST, …)`

NODE 47 / COMPILED STRUCTURE

DIVIDE_ASSIGN

Represents compound numeric division of an assignable target and a right-hand expression.

```
$average /= $count
```

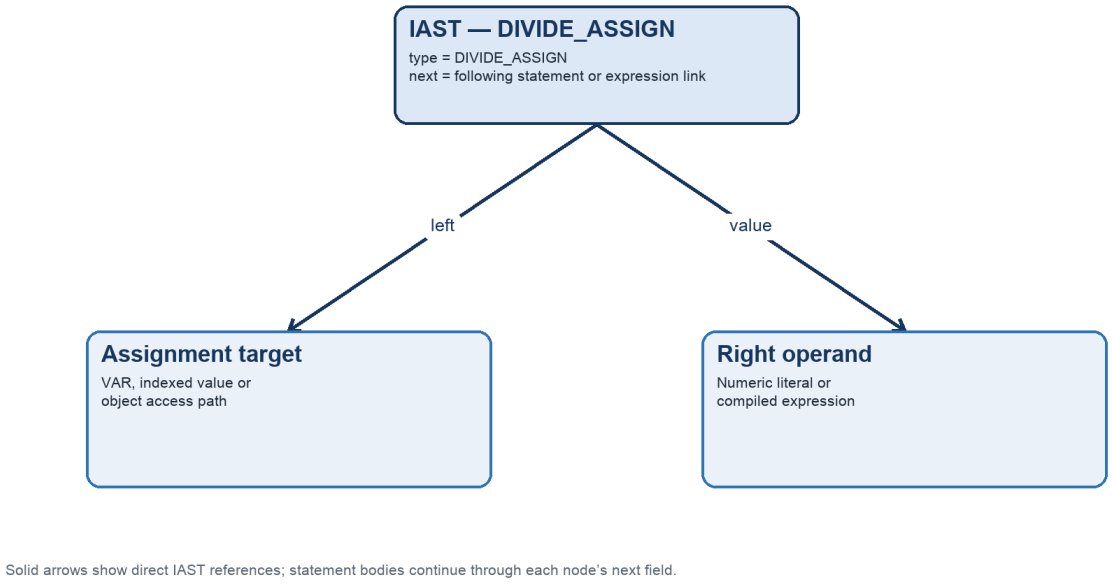

Figure 47. Compiler-produced IAST layout for DIVIDE_ASSIGN.

Field contract

Field	Meaning
`type`	DIVIDE_ASSIGN.
`left`	The assignable numeric target.
`value`	The expression whose numeric result is applied to the target.
`next`	The following statement or enclosing expression-chain link.

Implementation note. The compiler normalises this compound operator into a dedicated node instead of building a separate read, arithmetic expression and assignment. Runtime target resolution preserves variables, indexes and object properties, then applies division.

Source basis: `YASSCompiler.compileNumericAssignment(IAST);`
`ZPERuntimeEnvironment.applyNumericAssignment(IAST, …)`

NODE 48 / COMPILED STRUCTURE

POWER_ASSIGN

Represents compound numeric exponentiation of an assignable target and a right-hand expression.

```
$value ^= 2
```

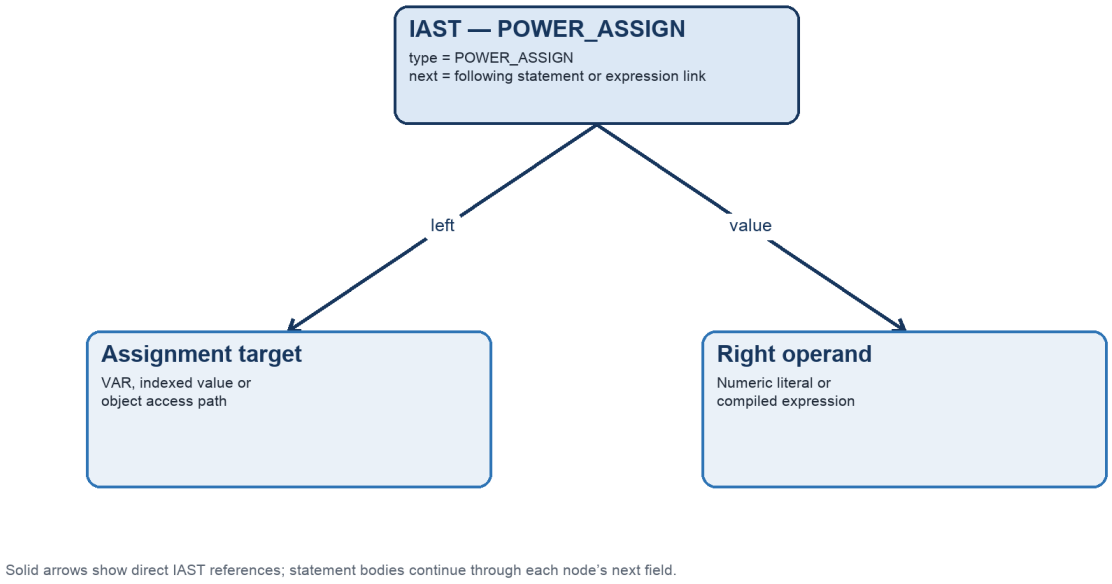

Figure 48. Compiler-produced IAST layout for POWER_ASSIGN.

Field contract

Field	Meaning
`type`	POWER_ASSIGN.
`left`	The assignable numeric target.
`value`	The expression whose numeric result is applied to the target.
`next`	The following statement or enclosing expression-chain link.

Implementation note. The compiler normalises this compound operator into a dedicated node instead of building a separate read, arithmetic expression and assignment. Runtime target resolution preserves variables, indexes and object properties, then applies Math.pow-based exponentiation.

Source basis: `YASSCompiler.compileNumericAssignment(IAST);`
`ZPERuntimeEnvironment.applyNumericAssignment(IAST, …)`

NODE 49 / COMPILED STRUCTURE

PRE_INCREMENT

Represents unit increment of an assignable target, returning the updated value.

```
++$index
```

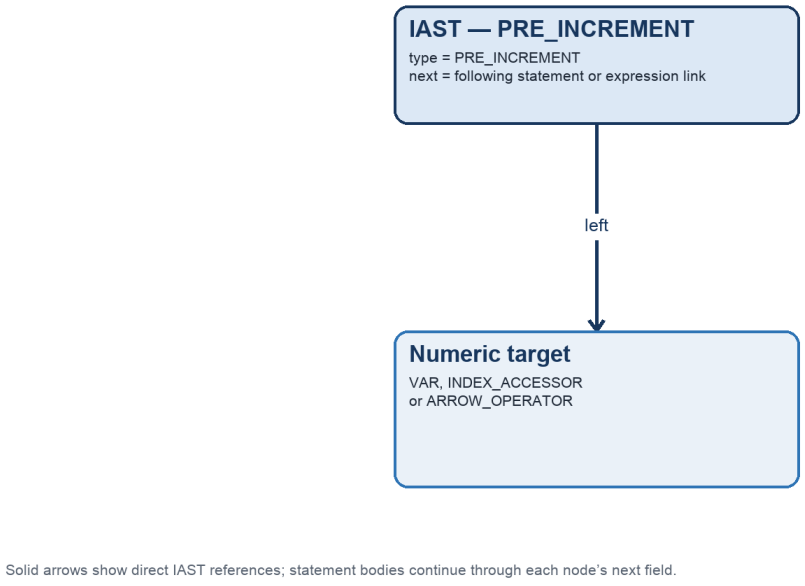

Figure 49. Compiler-produced IAST layout for PRE_INCREMENT.

Field contract

Field	Meaning
`type`	PRE_INCREMENT.
`left`	The assignable numeric target.
`next`	The following statement or enclosing expression-chain link.

Implementation note. The target is updated by one. Because the operator is before the target in source order, expression evaluation returns the updated value while preserving the same target-resolution machinery used by compound assignments.

Source basis: `YASSCompiler.compilePreIncrement(); ZPERuntimeEnvironment.applyUnitIncrement(IAST, …)`

NODE 50 / COMPILED STRUCTURE

POST_INCREMENT

Represents unit increment of an assignable target, returning the original value.

```
$index++
```

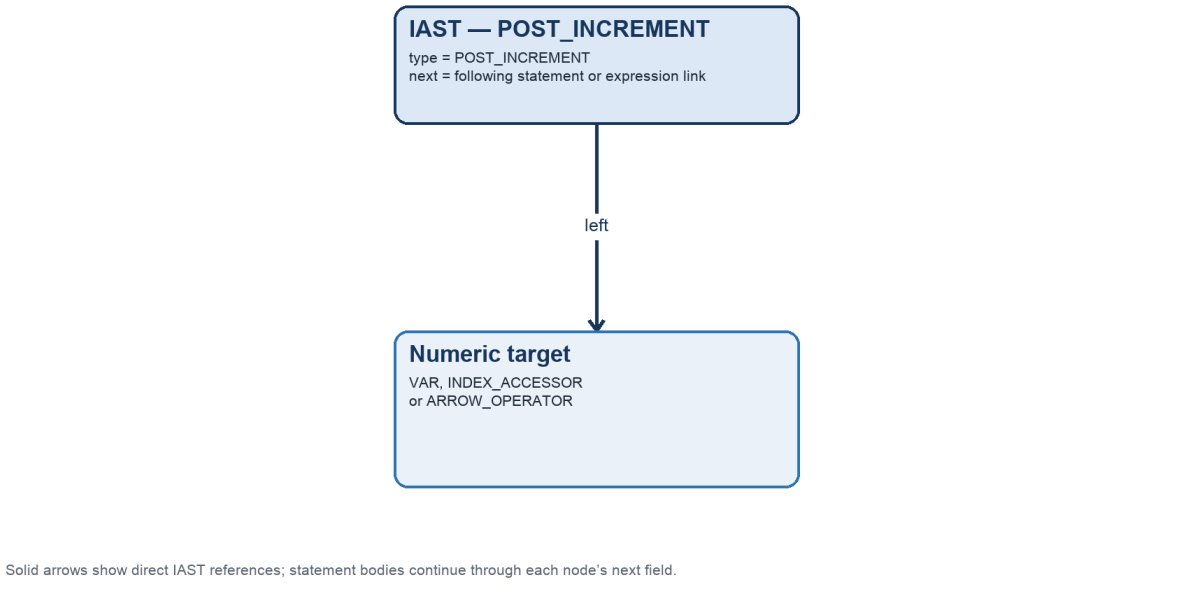

Figure 50. Compiler-produced IAST layout for POST_INCREMENT.

Field contract

Field	Meaning
`type`	POST_INCREMENT.
`left`	The assignable numeric target.
`next`	The following statement or enclosing expression-chain link.

Implementation note. The target is updated by one. Because the operator is after the target in source order, expression evaluation returns the original value while preserving the same target-resolution machinery used by compound assignments.

Source basis: `YASSCompiler.compilePostIncrement(); ZPERuntimeEnvironment.applyUnitIncrement(IAST, …)`

NODE 51 / COMPILED STRUCTURE

PRE_DECREMENT

Represents unit decrement of an assignable target, returning the updated value.

```
--$remaining
```

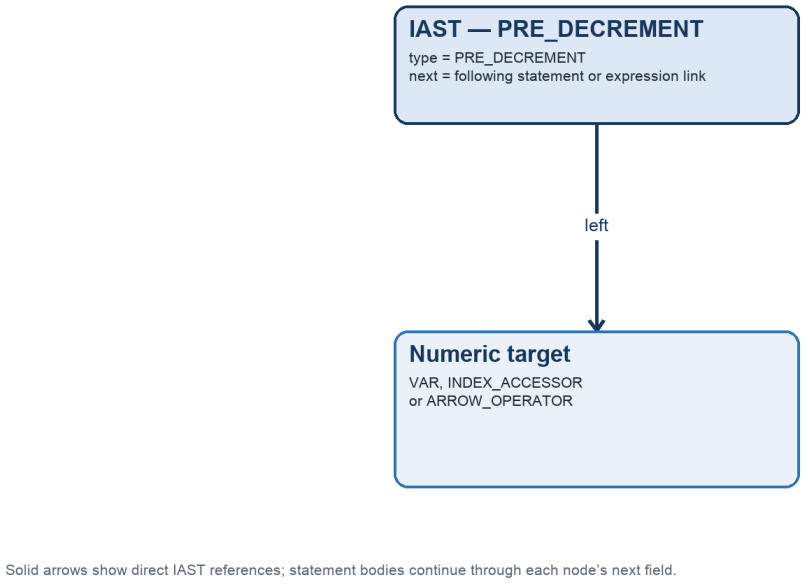

Figure 51. Compiler-produced IAST layout for PRE_DECREMENT.

Field contract

Field	Meaning
`type`	PRE_DECREMENT.
`left`	The assignable numeric target.
`next`	The following statement or enclosing expression-chain link.

Implementation note. The target is updated by one. Because the operator is before the target in source order, expression evaluation returns the updated value while preserving the same target-resolution machinery used by compound assignments.

Source basis: `YASSCompiler.compilePreDecrement(); ZPERuntimeEnvironment.applyUnitIncrement(IAST, …)`

NODE 52 / COMPILED STRUCTURE

POST_DECREMENT

Represents unit decrement of an assignable target, returning the original value.

```
$remaining--
```

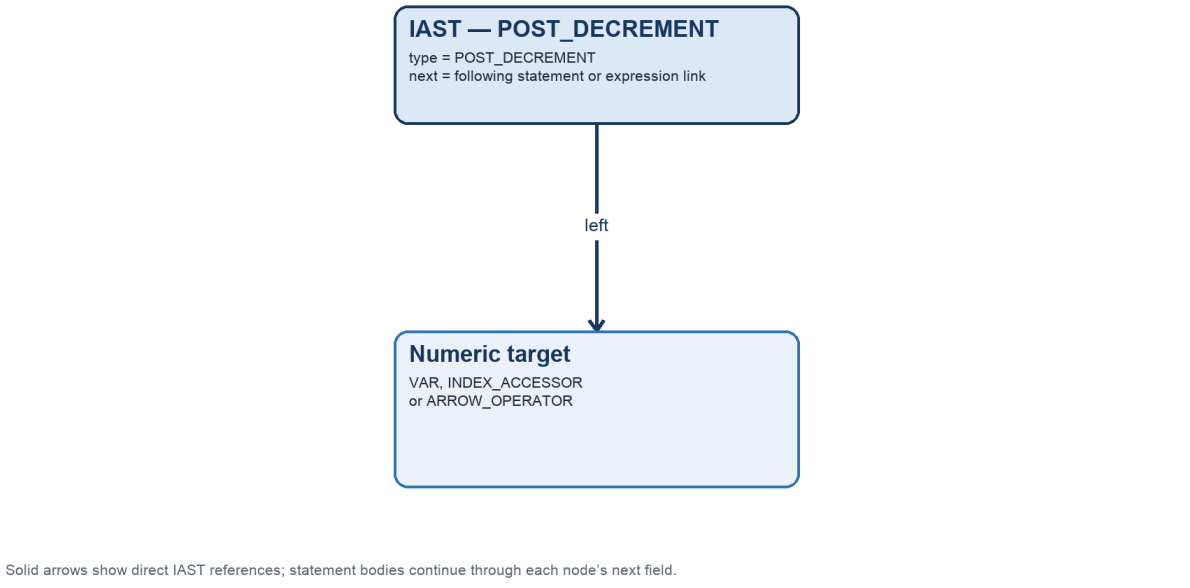

Figure 52. Compiler-produced IAST layout for POST_DECREMENT.

Field contract

Field	Meaning
`type`	POST_DECREMENT.
`left`	The assignable numeric target.
`next`	The following statement or enclosing expression-chain link.

Implementation note. The target is updated by one. Because the operator is after the target in source order, expression evaluation returns the original value while preserving the same target-resolution machinery used by compound assignments.

Source basis: `YASSCompiler.compilePostDecrement(); ZPERuntimeEnvironment.applyUnitIncrement(IAST, …)`

NODE 53 / COMPILED STRUCTURE

DECONSTRUCTING_ASSIGNMENT

Represents assignment of successive iterable elements to a sequence of target variables.

```
($x, $y) = [10, 20]
```

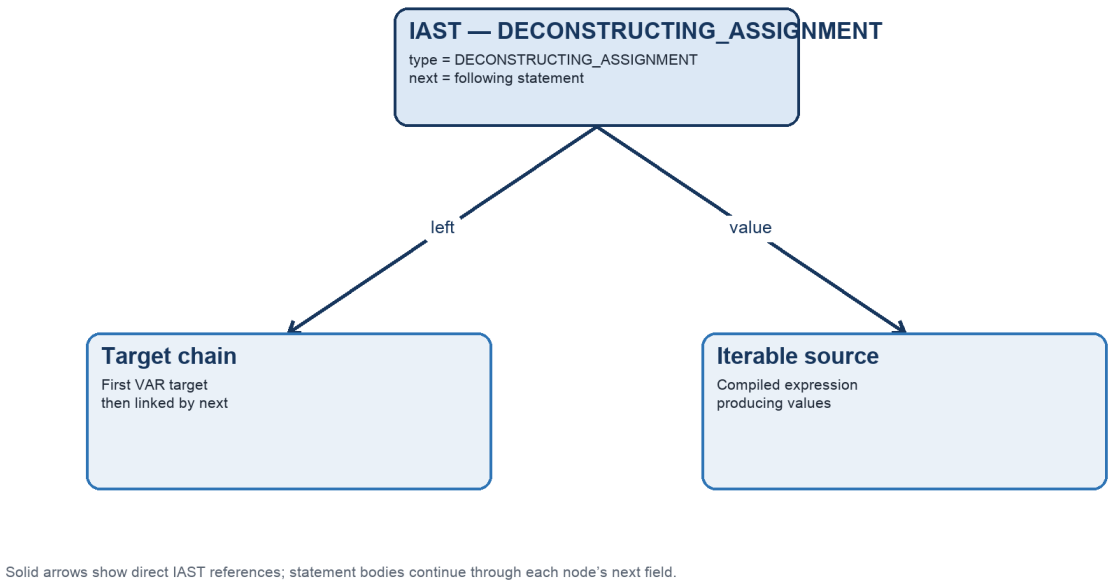

Figure 53. Compiler-produced IAST layout for DECONSTRUCTING_ASSIGNMENT.

Field contract

Field	Meaning
`type`	DECONSTRUCTING_ASSIGNMENT.
`left`	First target variable; additional targets are linked through next.
`value`	The expression expected to evaluate to an Iterable value.
`next`	The following statement in the containing body.

Implementation note. The runtime first materialises source elements, verifies there are enough values, then assigns them positionally. A map source produces one-entry maps for each target.

Source basis: `YASSCompiler.compileDestructuringAssignment();`
`ZPERuntimeEnvironment.handleDeconstructingAssignment(IAST, …)`

NODE 54 / COMPILED STRUCTURE

GLOBAL

Represents promotion of an existing local variable into the runtime's global variable map.

```
global($configuration)
```

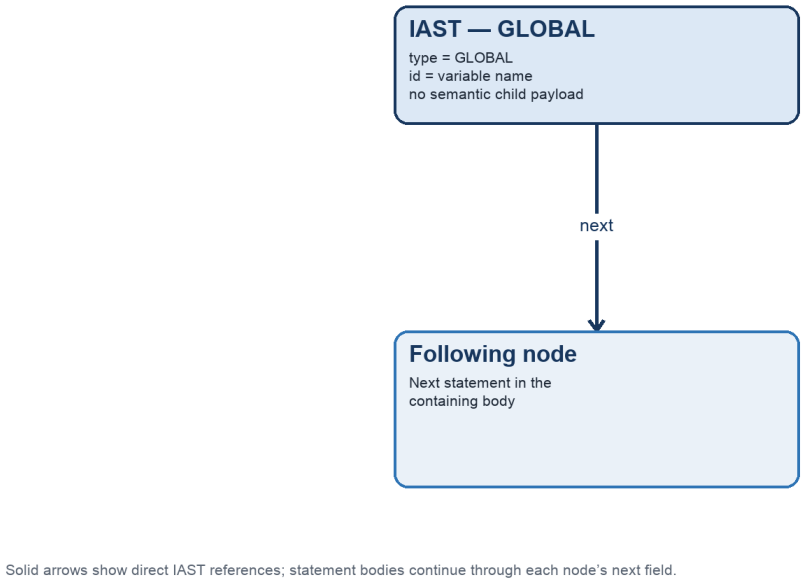

Figure 54. Compiler-produced IAST layout for GLOBAL.

Field contract

Field	Meaning
`type`	GLOBAL.
`id`	The variable identifier to expose globally.
`next`	The following statement in the containing body.

Implementation note. GLOBAL does not contain a value expression. Runtime handling looks up the already defined variable by id and shares its ZPEVariable wrapper with the global function.

Source basis: `YASSCompiler.compileGlobal(); ZPERuntimeEnvironment.handleGlobal(IAST, …)`

NODE 55 / COMPILED STRUCTURE

IMPORT

Represents loading of a named library, optionally under an alias.

```
import("storage") as store
```

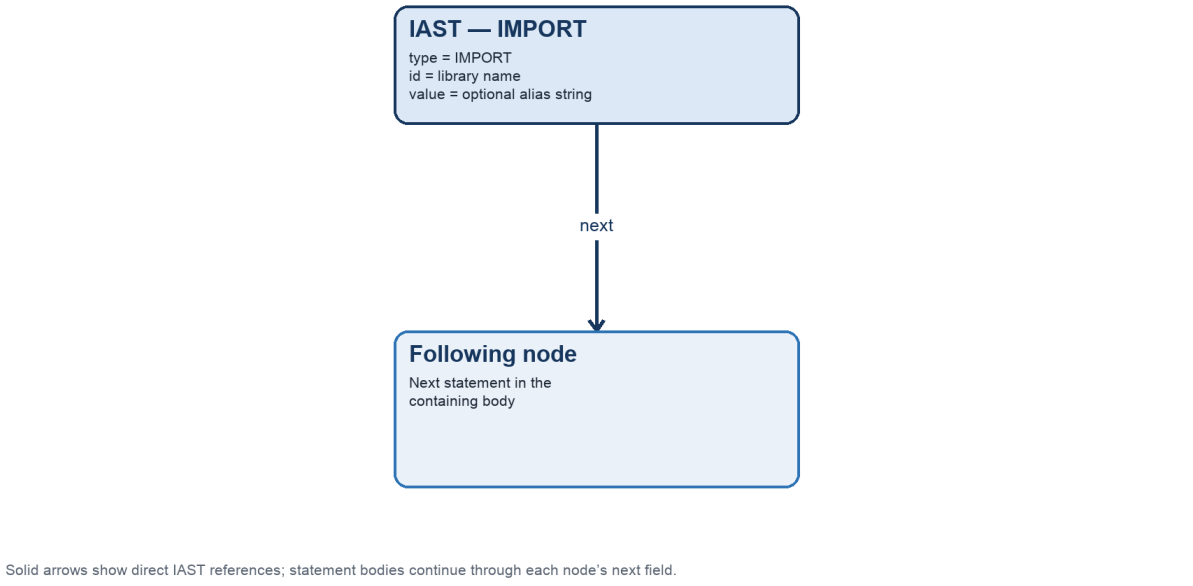

Figure 55. Compiler-produced IAST layout for IMPORT.

Field contract

Field	Meaning
`type`	IMPORT.
`id`	The requested library identifier or string.
`value`	Optional alias string following AS; otherwise null.
`next`	The following statement in the containing body.

Implementation note. The alias is a scalar payload, not a child node. Compilation also records the library in the executable's required-library collection for dependency metadata.

Source basis: `YASSCompiler.compileImport(); ZPERuntimeEnvironment.importLib(String, String)`

NODE 56 / COMPILED STRUCTURE

MODULE

Represents a named module that owns a sequence of definitions and executable nodes.

```
module stdAlgorithms … end module
```

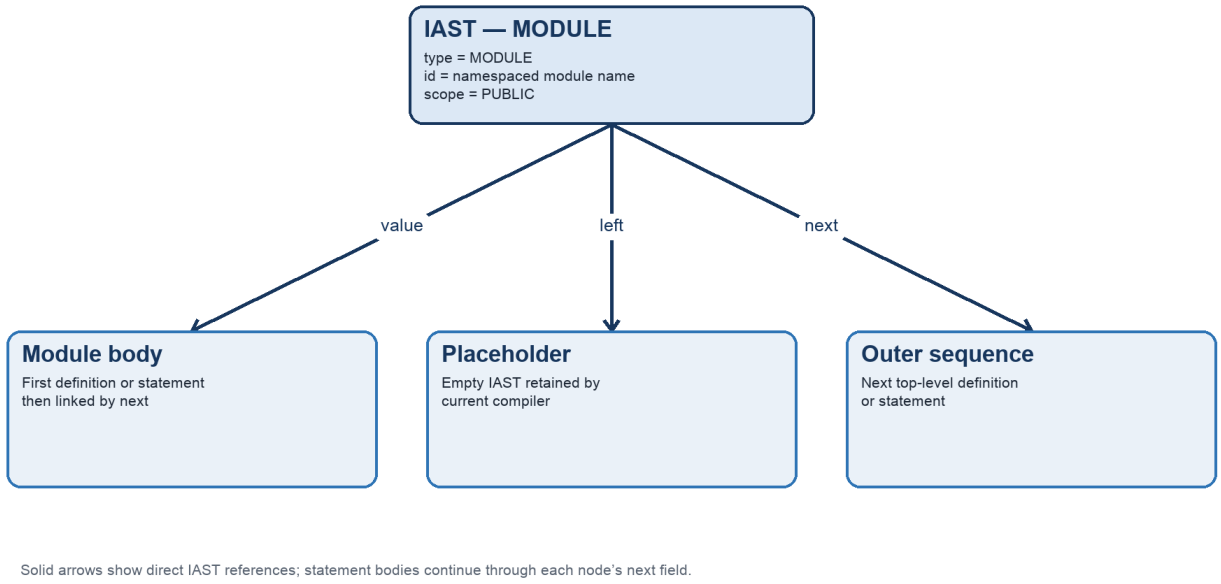

Figure 56. Compiler-produced IAST layout for MODULE.

Field contract

Field	Meaning
`type`	MODULE.
`id`	The module name, optionally prefixed by its namespace path.
`value`	First node in the module body.
`left`	An empty IAST placeholder retained by the current compiler.
`scope`	PUBLIC.
`next`	The next top-level definition or statement.

Implementation note. Module membership is expressed structurally: all owned definitions are reachable from value. Runtime construction creates a ZPEModule around that body.

Source basis: `YASSCompiler.compileModule(); ZPERuntimeEnvironment traversal and ZPEModule construction`

NODE 57 / COMPILED STRUCTURE

STRUCTURE

Represents a user-defined structure, its members, visibility and optional inheritance.

```
public structure queue inherits collection … end structure
```

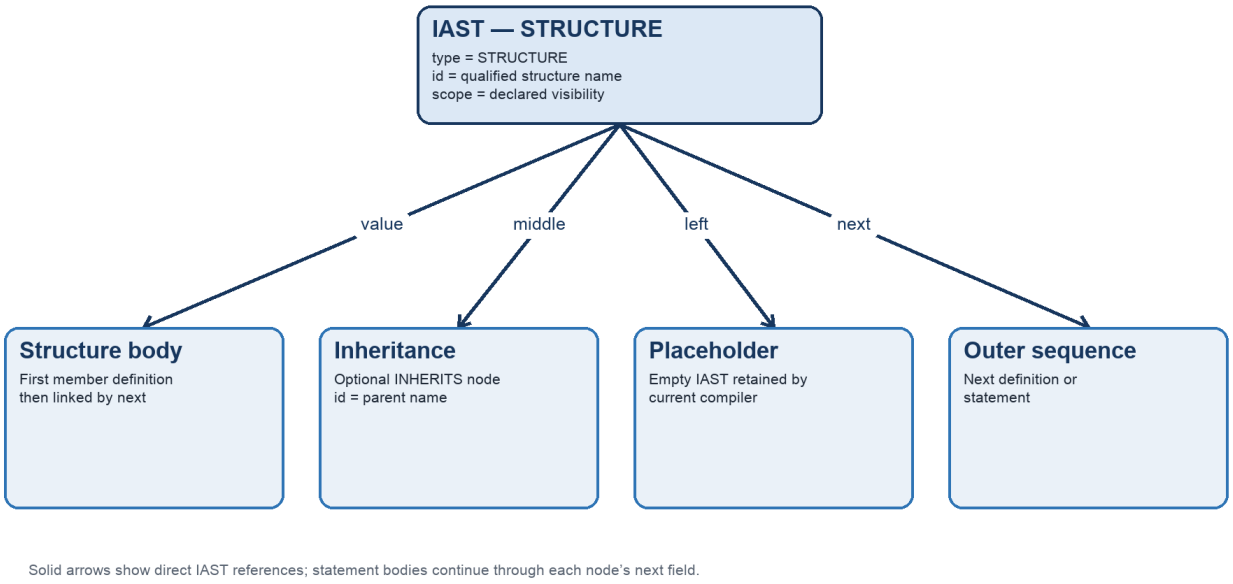

Figure 57. Compiler-produced IAST layout for STRUCTURE.

Field contract

Field	Meaning
`type`	STRUCTURE.
`id`	The structure name, optionally qualified by a namespace.
`value`	First member definition in the structure body.
`middle`	Optional INHERITS descriptor containing the parent structure id.
`left`	An empty IAST placeholder retained by the current compiler.
`scope`	Declared PUBLIC, PRIVATE or PROTECTED visibility.
`next`	The next definition or statement in the owner.

Implementation note. Inheritance metadata is kept separate from members. Runtime construction converts this definition into an AbstractStructure. Ordinary accessible members participate in object instantiation, whereas STATIC definitions remain owned by the structure and are not copied onto individual object instances.

Source basis: `YASSCompiler.compileStructure(); ZPERuntimeEnvironment.handleStructure(IAST, …)`

NODE 58 / COMPILED STRUCTURE

RECORD

Represents a fixed-field record definition with declared field types and optional defaults.

```
record pupil is {string name, number age}
```

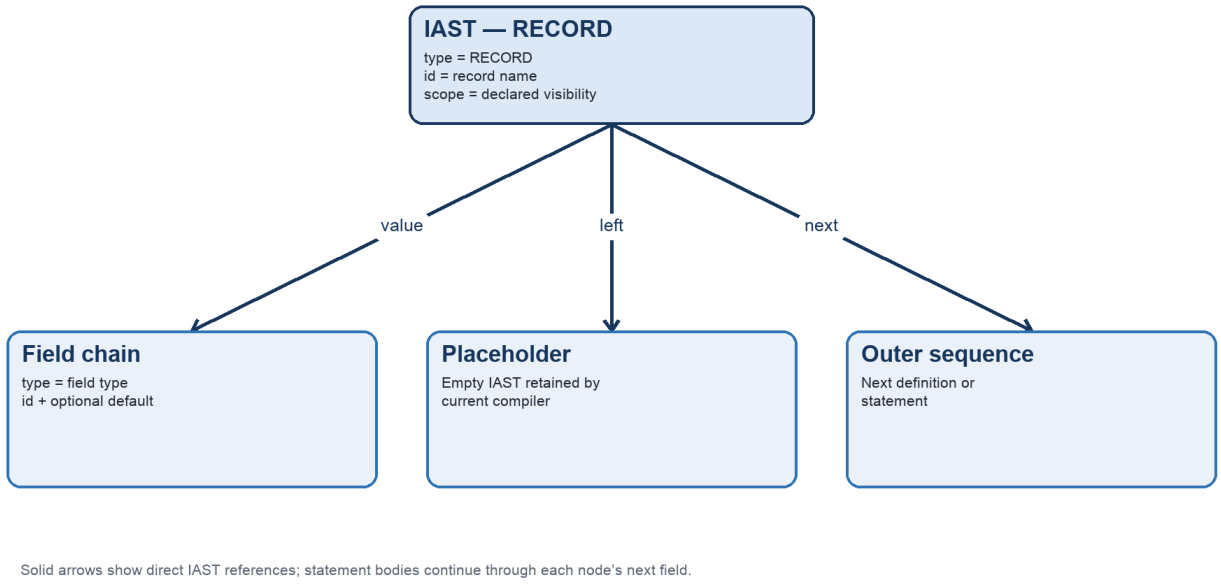

Figure 58. Compiler-produced IAST layout for RECORD.

Field contract

Field	Meaning
type	RECORD.
id	The record name.
value	First field descriptor; subsequent fields are linked through next.
field.type	The declared YASS field type.
field.id	The field name.
field.value	Optional compiled default-value expression.
left	An empty IAST placeholder retained by the current compiler.
scope	Declared PUBLIC, PRIVATE or PROTECTED visibility.
next	The next definition or statement in the owner.

Implementation note. Record fields are lightweight descriptors rather than ASSIGN nodes. Their type and id live directly on each chain node, with value populated only when a default is present.

Source basis: `YASSCompiler.compileRecord(); ZPERuntimeEnvironment.handleRecord(IAST, …)`

NODE 59 / COMPILED STRUCTURE

TRY

Represents protected traversal, an optional catch path and the variable used to expose an exception.

```
try … catch ($exception) … end try
```

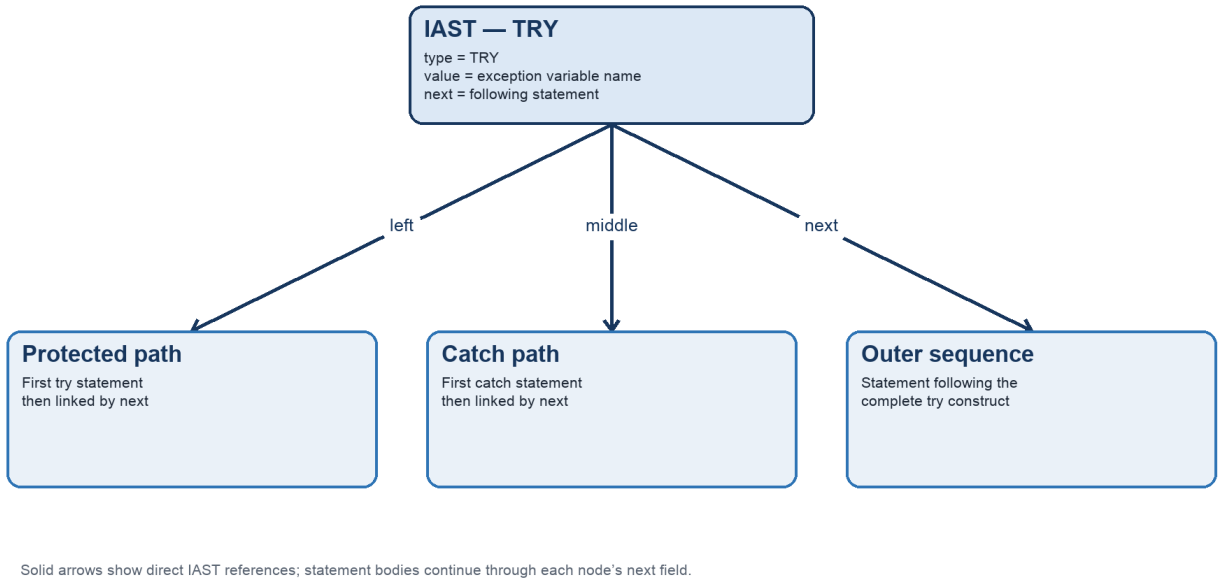

Figure 59. Compiler-produced IAST layout for TRY.

Field contract

Field	Meaning
`type`	TRY.
`left`	First statement in the protected path.
`middle`	First statement in the catch path, or null.
`value`	Catch-variable identifier string, or null when no name is declared.
`next`	The statement following the complete construct.

Implementation note. When an accepted runtime exception is caught, an ExceptionObject is temporarily bound under value before middle is traversed. Previous variable state is restored afterward.

Source basis: `YASSCompiler.compileTry(); ZPERuntimeEnvironment.handleTry(IAST, …)`

NODE 60 / COMPILED STRUCTURE

BLOCK

Represents a lexically isolated statement sequence executed through a dedicated block scope.

```
block … end block
```

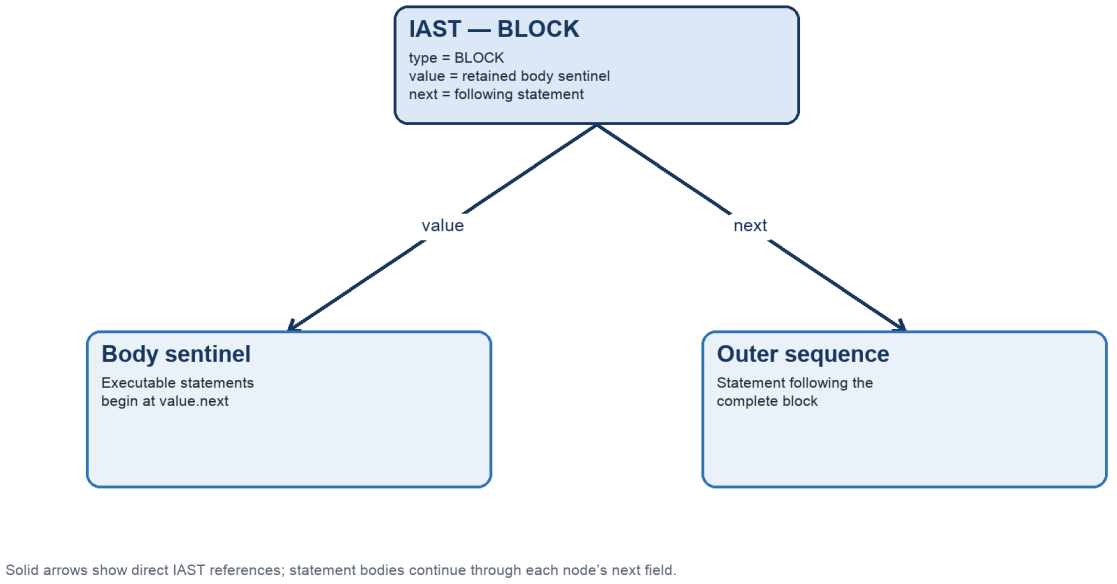

Figure 60. Compiler-produced IAST layout for BLOCK.

Field contract

Field	Meaning
`type`	BLOCK.
`value`	A retained body-construction sentinel whose next field begins the block body.
`next`	The statement following the complete block.

Implementation note. Unlike many body-owning constructs, BLOCK deliberately stores the sentinel rather than value.next. ZPEBlockScope receives that wrapper and manages isolated execution state.

Source basis: `YASSCompiler.compileBlock(); ZPERuntimeEnvironment.handleBlock(IAST, …); ZPEBlockScope`

NODE 61 / COMPILED STRUCTURE

ARROW_EXPRESSION

Represents a compact one- or two-parameter expression used as a predicate or callable value.

```
($value, $index) => $value > $index
```

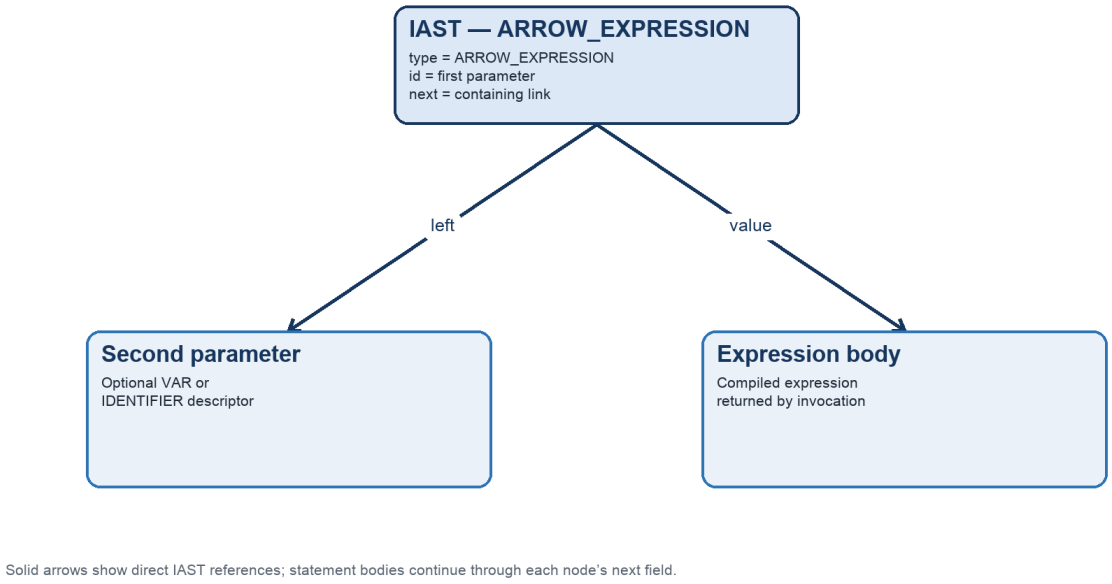

Figure 61. Compiler-produced IAST layout for ARROW_EXPRESSION.

Field contract

Field	Meaning
`type`	ARROW_EXPRESSION.
`id`	The first parameter identifier.
`left`	Optional descriptor for the second parameter.
`value`	The compiled expression body.
`next`	Optional link in the containing expression or argument chain.

Implementation note. Parameter names are registered in a temporary CompilerFunction while value is compiled, giving the body normal local-variable semantics without a full FUNCTION node.

Source basis: `YASSCompiler.compileArrowExpression(); ZPEArrowExpression`

NODE 62 / COMPILED STRUCTURE

TYPED_PARAMETER

Represents one explicitly typed formal parameter using separate variable and type descriptors.

```
function calculate(number $value) … end function
```

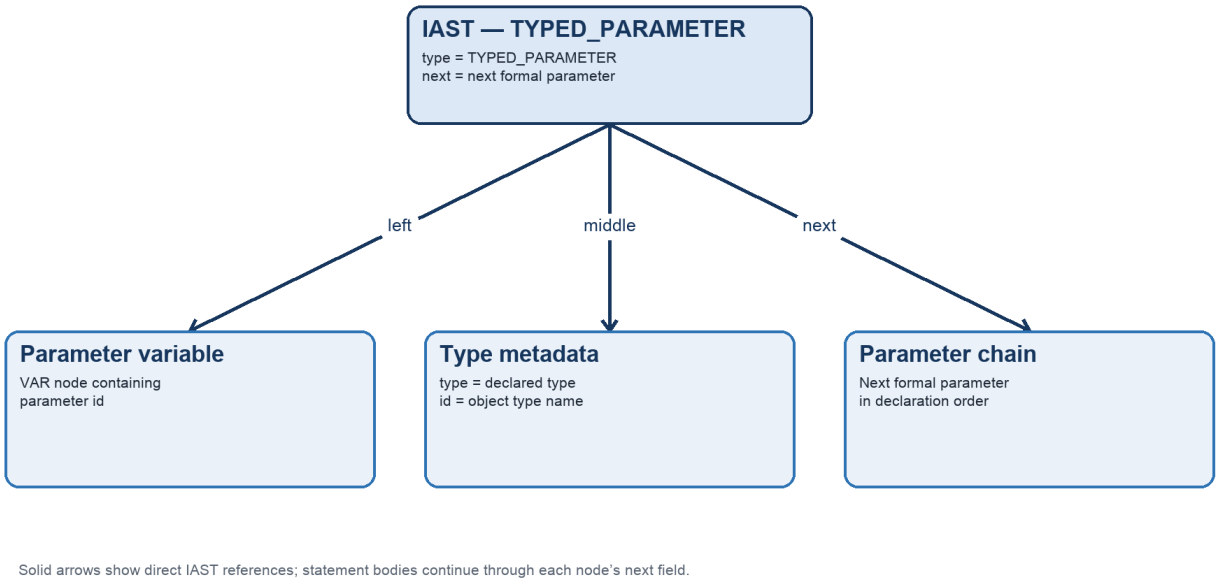

Figure 62. Compiler-produced IAST layout for TYPED_PARAMETER.

Field contract

Field	Meaning
`type`	TYPED_PARAMETER.
`left`	The parameter VAR node.
`middle`	Metadata IAST whose type byte is the declared parameter type.
`middle.id`	Optional specific object-type name.
`next`	The next formal parameter.

Implementation note. The type is not stored on the variable itself. Function construction reads the id from left and the accepted byte code from middle.type.

Source basis: `YASSCompiler.compileVar(); YASSCompiler.compileParameters(); ZPERuntimeEnvironment.astToFunction(IAST)`

NODE 63 / COMPILED STRUCTURE

INFINITE_PARAMETERS

Marks the final formal parameter as variadic so it can receive an arbitrary number of arguments.

```
function combine($values...) … end function
```

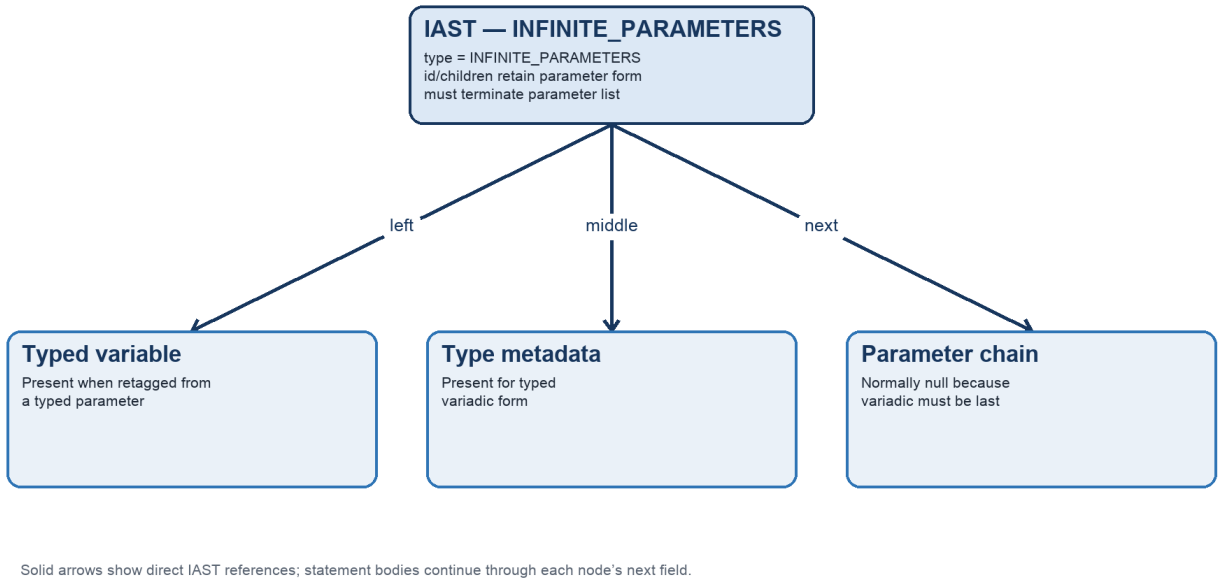

Figure 63. Compiler-produced IAST layout for INFINITE_PARAMETERS.

Field contract

Field	Meaning
`type`	INFINITE_PARAMETERS.
`id`	Parameter id for the ordinary untyped form.
`left`	Retained variable descriptor when originating from a typed parameter.
`middle`	Retained type descriptor when originating from a typed parameter.
`next`	Normally null because this parameter must be last.

Implementation note. The compiler retags the already compiled parameter node. Runtime function creation uses this byte code to enable acceptInfiniteParams and collect surplus arguments.

Source basis: `YASSCompiler.compileParameters(); ZPERuntimeEnvironment function-definition handling`

NODE 64 / COMPILED STRUCTURE

PARAMETERS

Acts as the empty-list sentinel for a formal parameter list.

```
function task() … end function
```

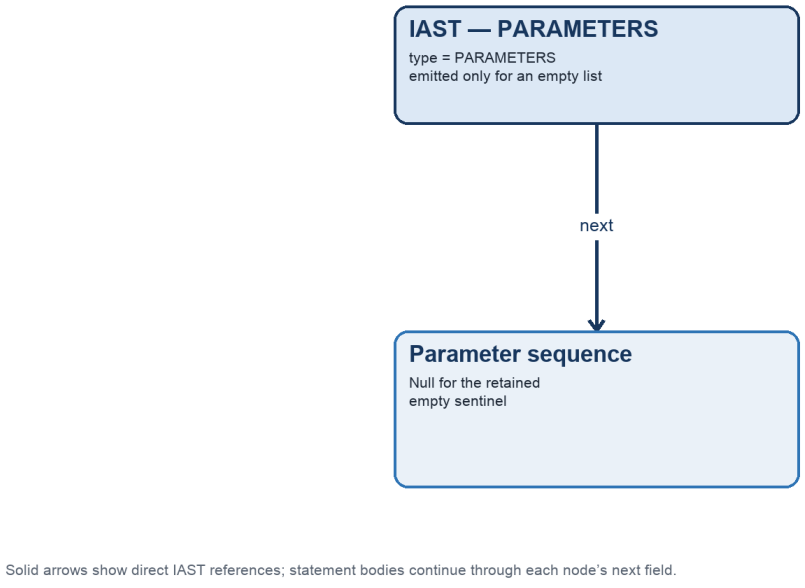

Figure 64. Compiler-produced IAST layout for PARAMETERS.

Field contract

Field	Meaning
`type`	PARAMETERS.
`next`	Null in the retained empty-list form.

Implementation note. For a non-empty declaration, compileParameters returns the first actual parameter and discards its PARAMETERS construction sentinel. The byte code therefore survives only for an empty list.

Source basis: `YASSCompiler.compileParameters()`

NODE 65 / COMPILED STRUCTURE

ARGUMENTS

Acts as the empty-list sentinel for a call argument list.

```
perform_task()
```

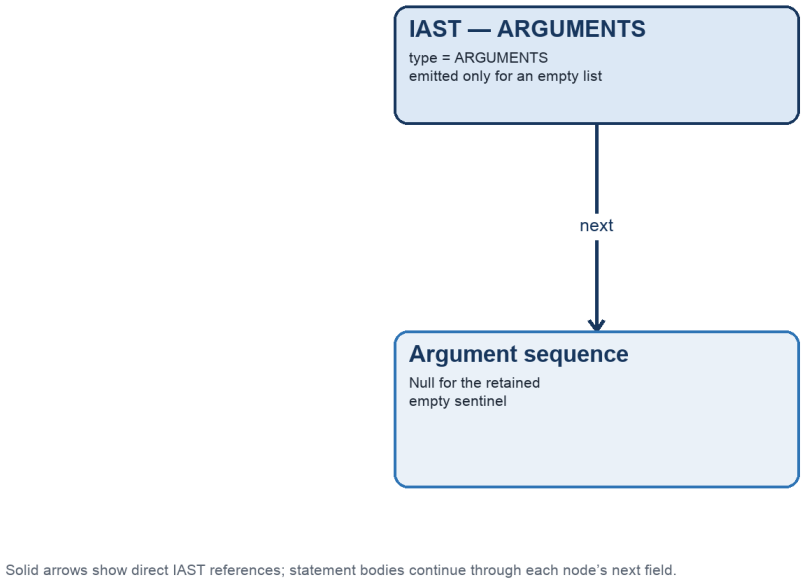

Figure 65. Compiler-produced IAST layout for ARGUMENTS.

Field contract

Field	Meaning
`type`	ARGUMENTS.
`next`	Null in the retained empty-list form.

Implementation note. For a non-empty call, compileArguments returns the first expression and discards the sentinel. Runtime argument generation treats this empty node as zero arguments.

Source basis: `YASSCompiler.compileArguments(); ZPERuntimeEnvironment.generateArguments(IAST)`

NODE 66 / COMPILED STRUCTURE

THIS

Represents this or the special super access root inside a structure instance method.

```
this->property    super->describe()
```

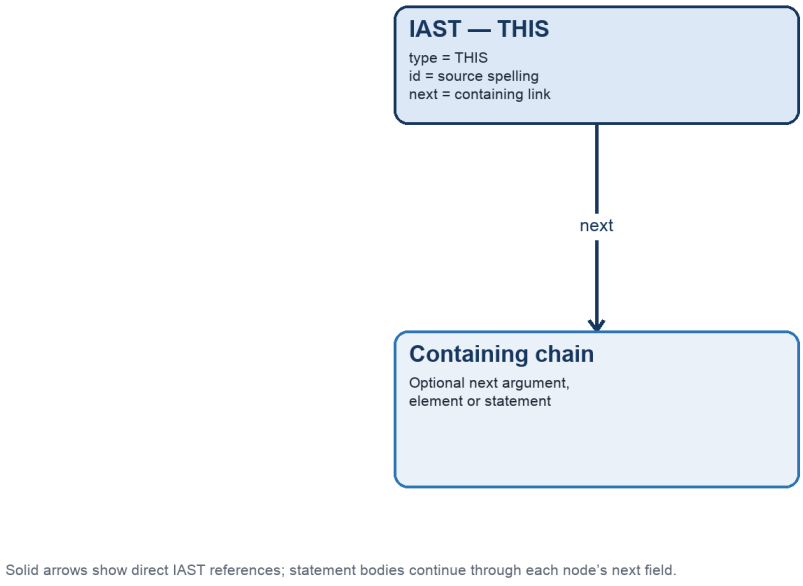

Figure 66. Compiler-produced IAST layout for THIS.

Field contract

Field	Meaning
`type`	THIS.
`id`	The source identifier: this, self or super.
`next`	Optional link in the enclosing sequence.

Implementation note. Member-access compilation places this node in ARROW_OPERATOR.middle. An id of this resolves the current function's parent object normally; an id of super uses that same object but begins method lookup at its inherited parent structure.

Source basis: `YASSCompiler.compileVar(); ZPERuntimeEnvironment member-access handling`

NODE 67 / COMPILED STRUCTURE

BREAKPOINT

Represents a debugger suspension point without any semantic operand payload.

```
breakpoint
```

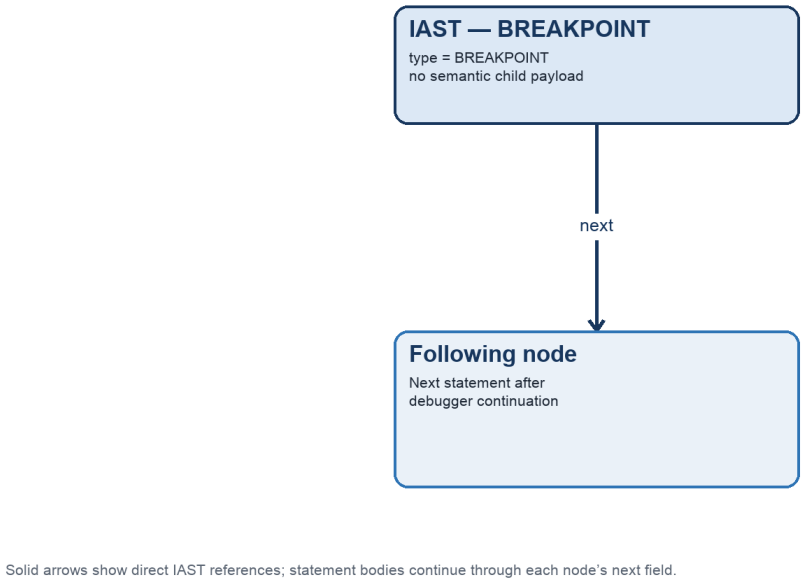

Figure 67. Compiler-produced IAST layout for BREAKPOINT.

Field contract

Field	Meaning
`type`	BREAKPOINT.
`next`	The following statement in the containing body.

Implementation note. Ordinary execution passes through this node. When debugging is enabled, function traversal raises a BreakPointHalt before continuing to next.

Source basis: `YASSCompiler.compileBreakpoint(); ZPEFunction.traverse(IAST)`

NODE 68 / COMPILED STRUCTURE

GOTO

Transfers statement traversal to the compiled node associated with a named label.

```
goto target_label
```

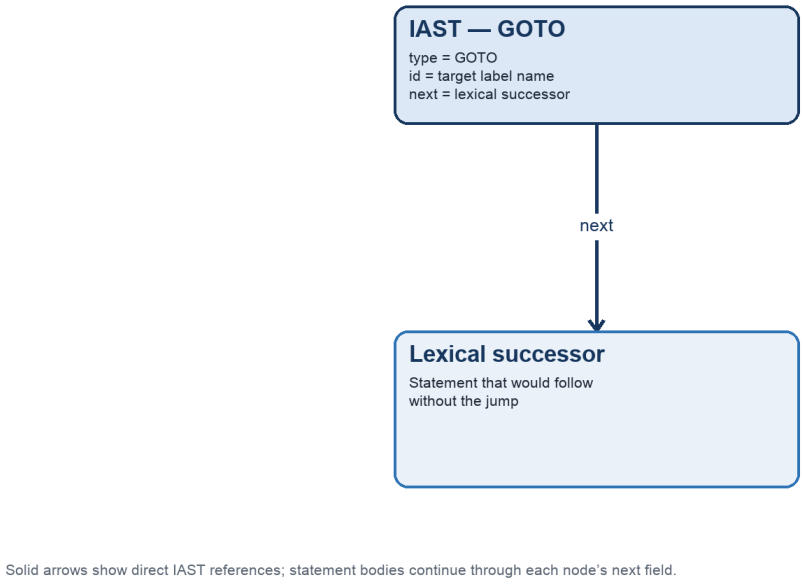

Figure 68. Compiler-produced IAST layout for GOTO.

Field contract

Field	Meaning
`type`	GOTO.
`id`	Name of the label to which execution should jump.
`next`	The next statement in lexical order; bypassed when the jump succeeds.

Implementation note. Labels are retained in the program label map rather than emitted as separate LABEL nodes. During traversal, id is resolved through that map and the current instruction pointer is replaced with the target node.

Source basis: `YASSCompiler.compileGoTo(); YASSCompiler label registration; ZPEFunction.traverse(IAST)`

NODE 69 / COMPILED STRUCTURE

TEST

Represents a test function using the normal function layout with an additional test designation.

```
test function validates_result() … end function
```

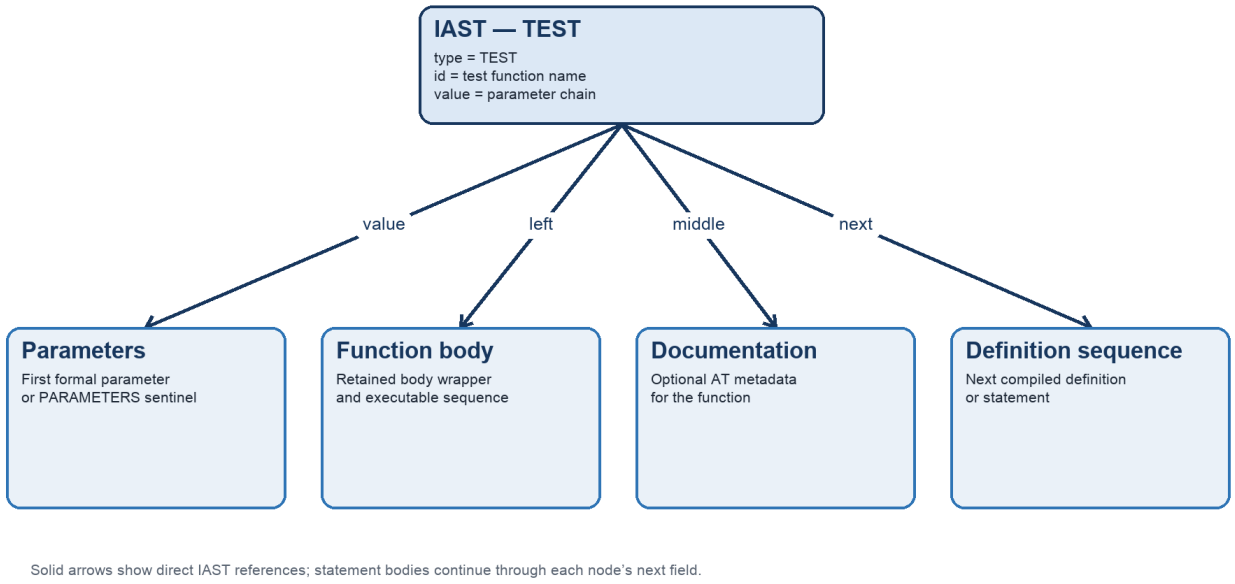

Figure 69. Compiler-produced IAST layout for TEST.

Field contract

Field	Meaning
`type`	TEST.
`id`	The declared test-function name.
`value`	The formal parameter representation.
`left`	The compiled function body representation.
`middle`	Optional documentation metadata.
`returnTypes`	Optional accepted return-type metadata.
`scope`	Compiled access level.
`next`	The next node in the containing sequence.

Implementation note. Compilation begins with the standard FUNCTION contract and retags the node as TEST. Runtime definition handling constructs a ZPEFunction and marks it as a test function, allowing assertions to report test-specific results while debugging.

Source basis: `YASSCompiler.compileFunction(); ZPERuntimeEnvironment.defineFunction(IAST, …)`

NODE 70 / COMPILED STRUCTURE

LET

Represents a declaration-time assignment that follows the assignment layout but requires a new binding.

```
let $result be calculate()
```

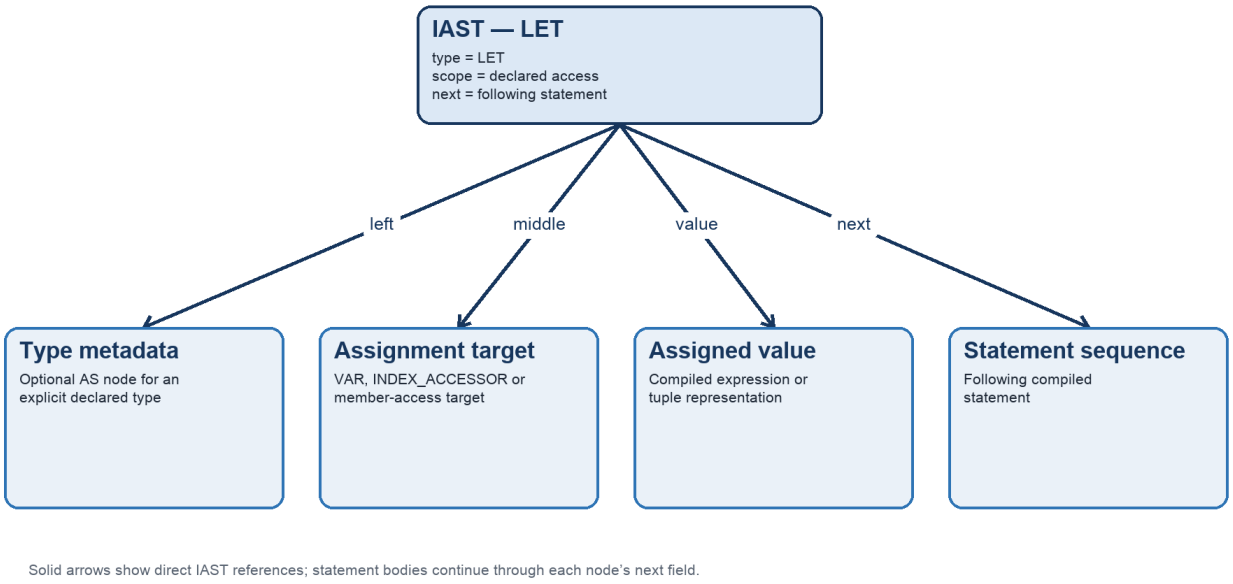

Figure 70. Compiler-produced IAST layout for LET.

Field contract

Field	Meaning
`type`	LET.
`scope`	Access level attached to the declared variable.
`left`	Optional AS metadata describing the declared type.
`middle`	The variable or compound assignment target.
`value`	The expression whose result is assigned.
`next`	The following statement.

Implementation note. LET is produced by the assignment compiler when the source declaration uses let. The runtime evaluates it through the same assignment machinery as ASSIGN, but passes a declaration flag so the target is treated as a newly introduced binding.

Source basis: `YASSCompiler.compileVar(); YASSCompiler.compileAssign(IAST, …); ZPERuntimeEnvironment.constructTraverse(IAST, …)`

NODE 71 / COMPILED STRUCTURE

VAR_BY_REF

Represents a variable access whose runtime value must be retained by reference rather than copied.

```
&$value
```

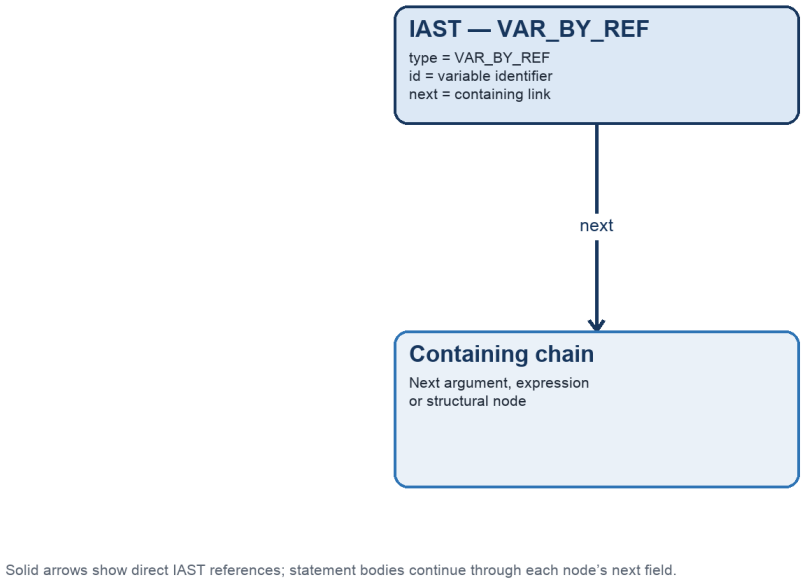

Figure 71. Compiler-produced IAST layout for VAR_BY_REF.

Field contract

Field	Meaning
`type`	VAR_BY_REF.
`id`	The referenced variable identifier, including its variable prefix where applicable.
`scope`	Access metadata inherited from variable compilation.
`next`	Optional continuation in the containing chain.

Implementation note. Ordinary VAR evaluation may copy a ZPECoreType. VAR_BY_REF instead resolves the stored value directly from the active scope, preserving identity for argument passing and assignments that require reference semantics.

Source basis: `YASSCompiler.compileVar(); ZPERuntimeEnvironment.evaluateVarByReference(IAST, …)`

NODE 72 / COMPILED STRUCTURE

OPEN_HEREDOC

Represents a heredoc value as an ordered chain of string fragments and embedded expressions.

```
<<<TEXT … TEXT
```

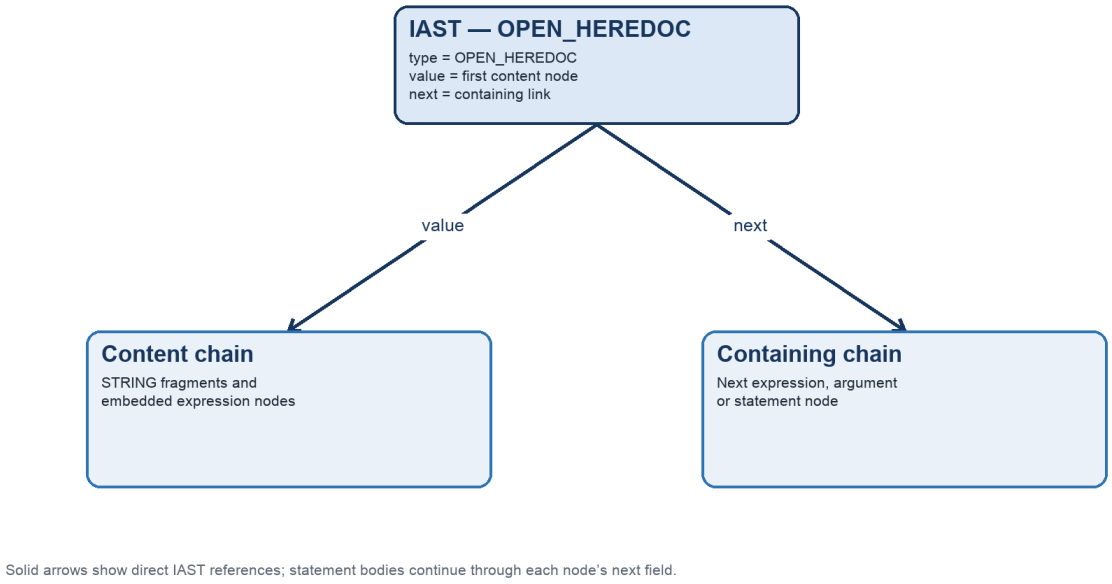

Figure 72. Compiler-produced IAST layout for OPEN_HEREDOC.

Field contract

Field	Meaning
`type`	OPEN_HEREDOC.
`value`	The first node in the ordered heredoc-content chain.
`next`	Optional continuation in the enclosing structure.

Implementation note. The simple inline-document form stores a STRING node in value. More expressive heredoc compilation may link several fragments through next. Runtime evaluation traverses the value chain and concatenates literal and evaluated content in order.

Source basis: `YASSCompiler.compileInlineDoc(); ZPERuntimeEnvironment.evaluateHeredoc(IAST)`

NODE 73 / COMPILED STRUCTURE

STRING

Represents a literal string value embedded directly in the compiled expression tree.

```
"Hello, world"
```

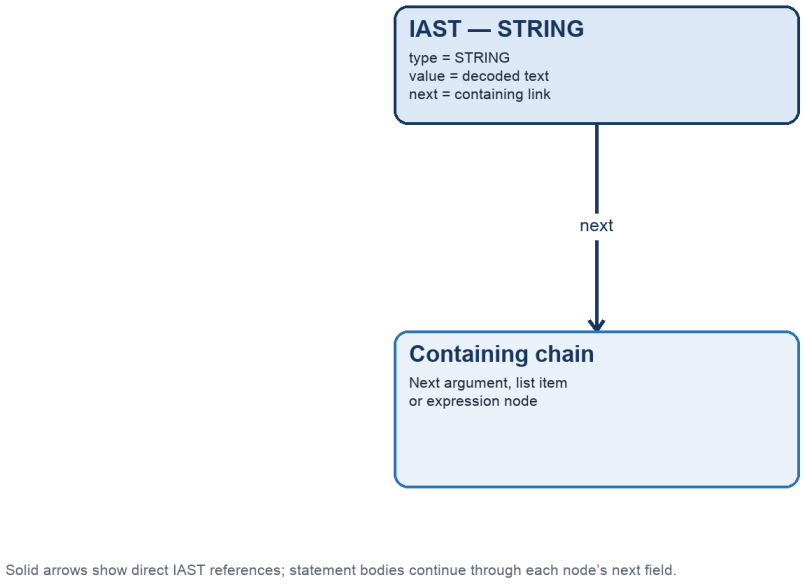

Figure 73. Compiler-produced IAST layout for STRING.

Field contract

Field	Meaning
`type`	STRING.
`value`	The compiled Java String content of the literal.
`next`	Optional continuation in the enclosing chain.

Implementation note. The compiler stores literal content in value after lexical string handling. Runtime evaluation constructs a ZPEString from that value. STRING nodes also appear as literal fragments inside heredoc and template-related structures.

Source basis: `YASSCompiler.compileNode(); YASSCompiler.compileUnescapedString(); ZPERuntimeEnvironment.evaluateExpression(IAST, …)`

NODE 74 / COMPILED STRUCTURE

INT

Represents an integral numeric literal stored in compiled numeric form.

```
42
```

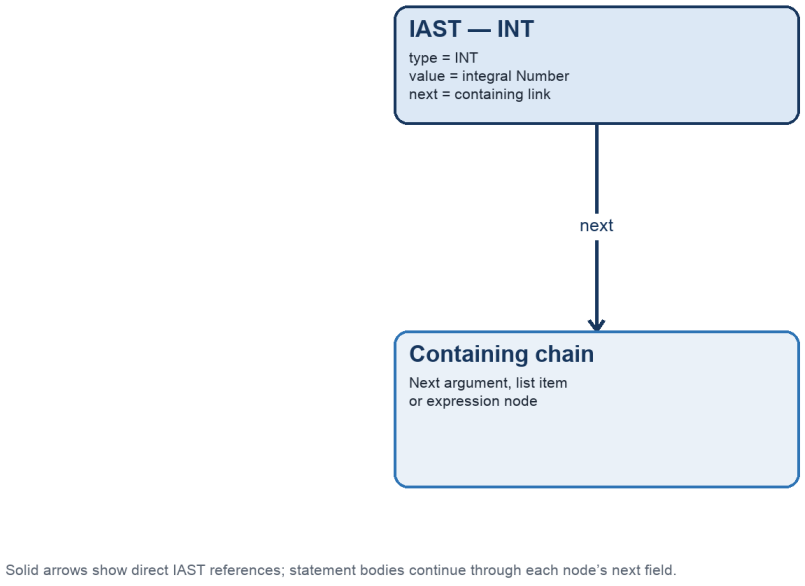

Figure 74. Compiler-produced IAST layout for INT.

Field contract

Field	Meaning
`type`	INT.
`value`	A Number containing the parsed integral value.
`next`	Optional continuation in the enclosing chain.

Implementation note. The source spelling is converted during compilation rather than reparsed on every execution. Runtime evaluation wraps the stored Number in a ZPENumber.

Source basis: `YASSCompiler.compileNode(); ZPERuntimeEnvironment.evaluateExpression(IAST, …)`

NODE 75 / COMPILED STRUCTURE

DOUBLE

Represents a floating-point numeric literal stored in compiled numeric form.

```
3.14159
```

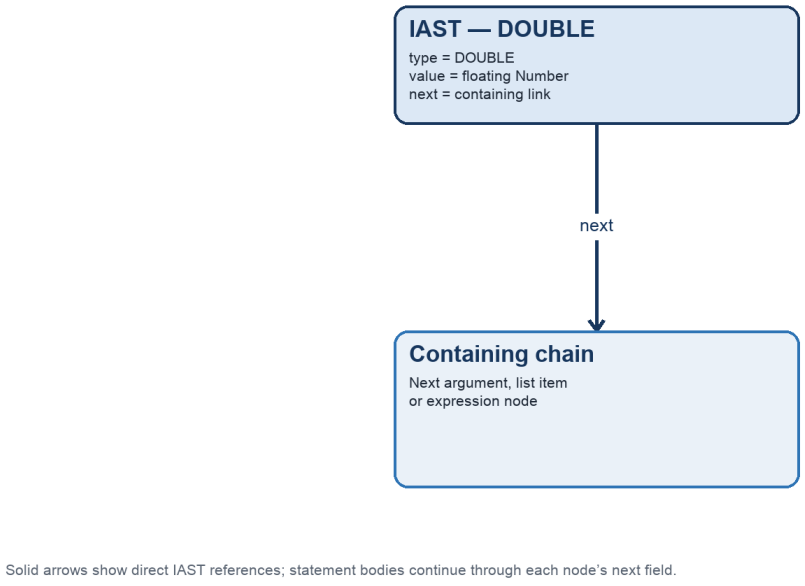

Figure 75. Compiler-produced IAST layout for DOUBLE.

Field contract

Field	Meaning
`type`	DOUBLE.
`value`	A Number containing the parsed floating-point value.
`next`	Optional continuation in the enclosing chain.

Implementation note. The compiler converts the textual literal to a numeric value before placing it in the IAST. Runtime evaluation therefore constructs a ZPENumber directly from value.

Source basis: `YASSCompiler.compileNode(); ZPERuntimeEnvironment.evaluateExpression(IAST, …)`

NODE 76 / COMPILED STRUCTURE

BOOL

Represents a Boolean literal used directly within a compiled expression.

```
true
```

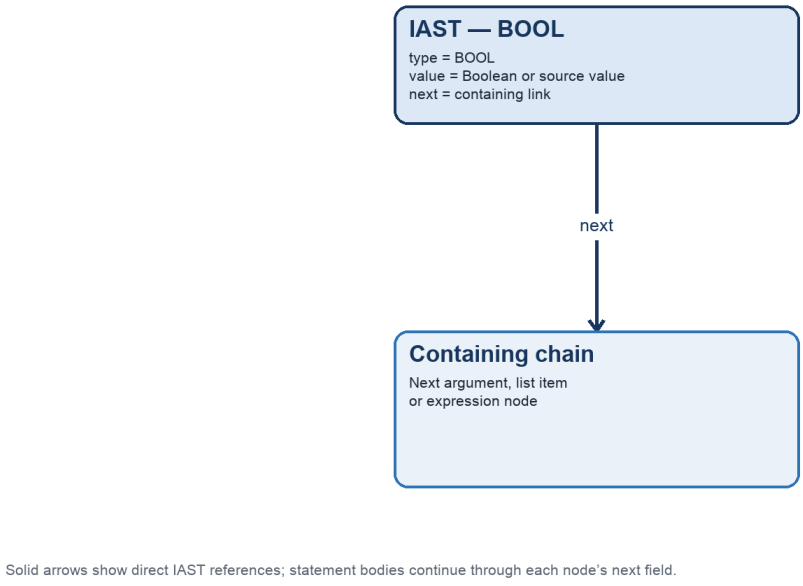

Figure 76. Compiler-produced IAST layout for BOOL.

Field contract

Field	Meaning
`type`	BOOL.
`value`	The literal Boolean value or its compiler-retained source representation.
`next`	Optional continuation in the enclosing chain.

Implementation note. Runtime evaluation returns the shared ZPEBoolean true or false value when value is already a Java Boolean. A retained source representation is converted through the runtime Boolean conversion helper.

Source basis: `YASSCompiler.compileNode(); ZPERuntimeEnvironment.evaluateExpression(IAST, …)`

NODE 77 / COMPILED STRUCTURE

NULL

Represents the explicit absence of a runtime value within a compiled expression.

```
null
```

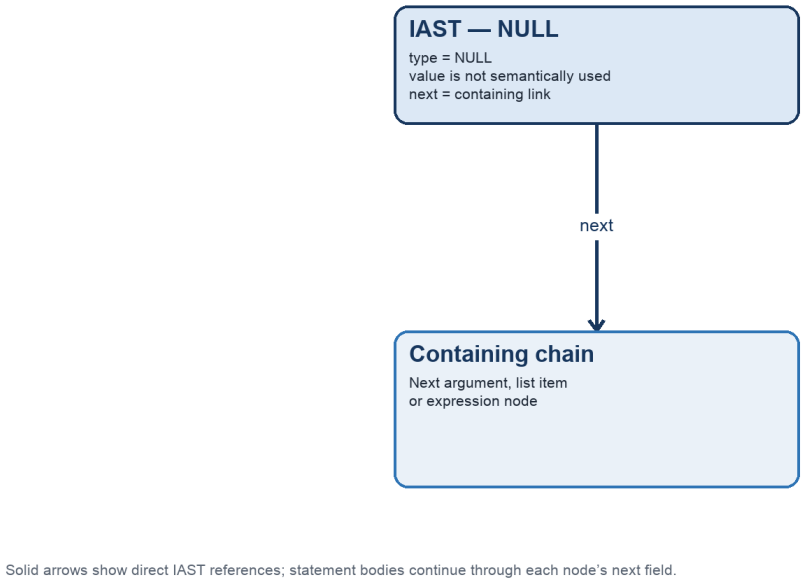

Figure 77. Compiler-produced IAST layout for NULL.

Field contract

Field	Meaning
`type`	NULL.
`value`	Not required for runtime evaluation.
`next`	Optional continuation in the enclosing chain.

Implementation note. Unlike a datatype marker, NULL is an executable expression node. Evaluation returns Java null, allowing assignments, arguments, comparisons and collection elements to carry an explicit absence of value.

Source basis: `YASSCompiler.compileNode(); ZPERuntimeEnvironment.evaluateExpression(IAST, …)`

NODE 78 / COMPILED STRUCTURE

FUNCTION_TYPE

Represents a named function as a first-class value without invoking it.

```
$callback = named_function
```

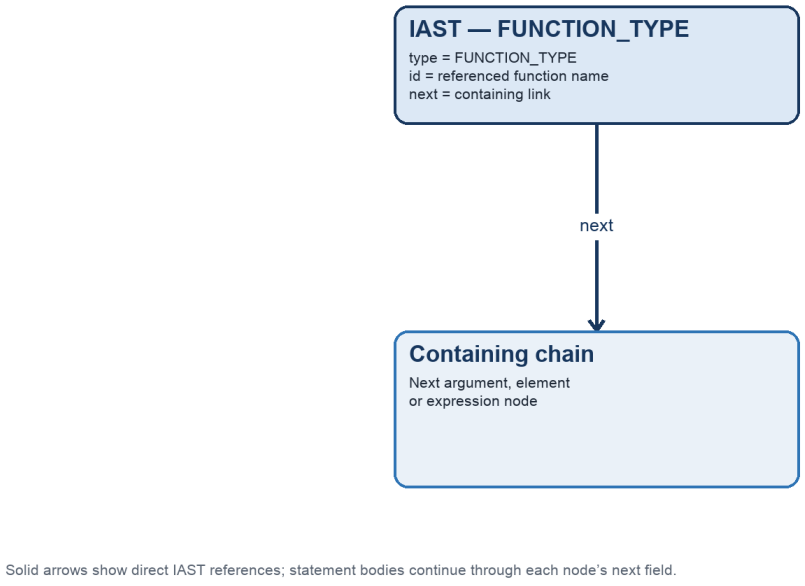

Figure 78. Compiler-produced IAST layout for FUNCTION_TYPE.

Field contract

Field	Meaning
`type`	FUNCTION_TYPE.
`id`	The name of the user-defined function or predefined command.
`next`	Optional continuation in the enclosing expression or argument sequence.

Implementation note. FUNCTION_TYPE also exists as a datatype byte, but this contract concerns the actual IAST emitted by compileFunctionDefinition. At runtime, the node is resolved to a FunctionReference while leaving the referenced function unexecuted.

Source basis: `YASSCompiler.compileFunctionDefinition();`
`ZPERuntimeEnvironment.resolveFunctionReference(IAST, …)`

NODE 79 / COMPILED STRUCTURE

AS

Carries declared-type metadata for an assignment without becoming part of the assigned expression.

```
declare $total as number = 0
```

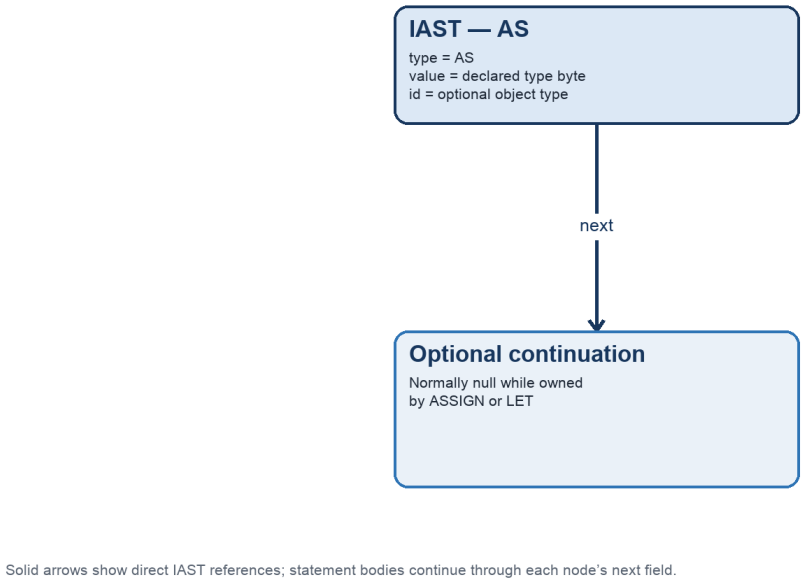

Figure 79. Compiler-produced IAST layout for AS.

Field contract

Field	Meaning
`type`	AS.
`value`	The byte code of the declared value type.
`id`	Optional specific object-type name for an object declaration.

Implementation note. The compiler stores this node in ASSIGN.left or LET.left. The target remains in middle and the assigned expression remains in value, allowing type metadata to be read independently of the executable right-hand side.

Source basis: `YASSCompiler.compileAssign(IAST, …); ZPERuntimeEnvironment assignment handling`

NODE 80 / COMPILED STRUCTURE

AT

Stores compiled documentation metadata attached to a definition or documented node.

```
@description "Returns the calculated total"
```

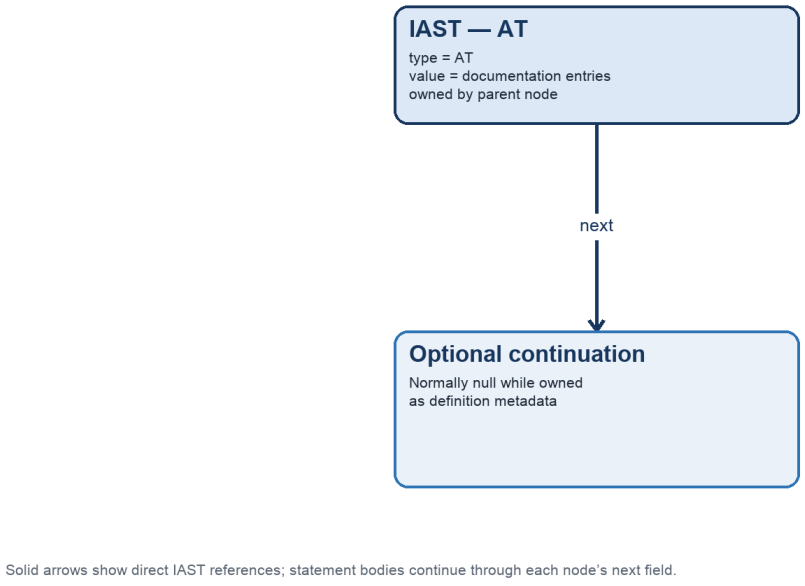

Figure 80. Compiler-produced IAST layout for AT.

Field contract

Field	Meaning
`type`	AT.
`value`	An array of name/value documentation entries.
`next`	Normally null because the node is owned through a parent metadata field.

Implementation note. Documentation annotations do not execute as statements. The compiler gathers them and places the AT node in the documented node's middle field when that field is available, preserving documentation alongside the compiled definition.

Source basis: `YASSCompiler.compileDoc(); definition documentation handling`

NODE 81 / COMPILED STRUCTURE

COLON

Represents the pair of boundary expressions used by an indexed range or slice.

```
$values[$start:$end]
```

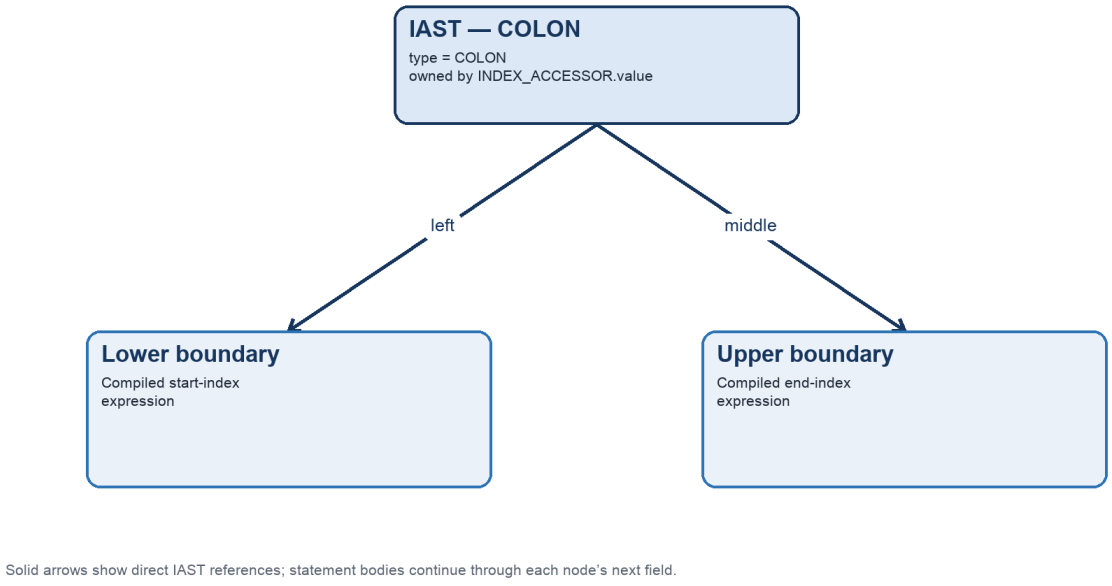

Figure 81. Compiler-produced IAST layout for COLON.

Field contract

Field	Meaning
`type`	COLON.
`left`	The lower or starting boundary expression.
`middle`	The upper or ending boundary expression.
`next`	Normally null; ownership is through INDEX_ACCESSOR.value.

Implementation note. COLON is not retained merely as punctuation. The compiler creates a dedicated node and places it in the accessor's value field. Runtime index evaluation recognises that node and constructs the range descriptor from its two children.

Source basis: `YASSCompiler.compileVar(); ZPERuntimeEnvironment.evaluateIndexAccessor(IAST, …)`

NODE 82 / COMPILED STRUCTURE

CONCAT

Represents concatenation as a dedicated binary expression node.

```
$first & $second
```

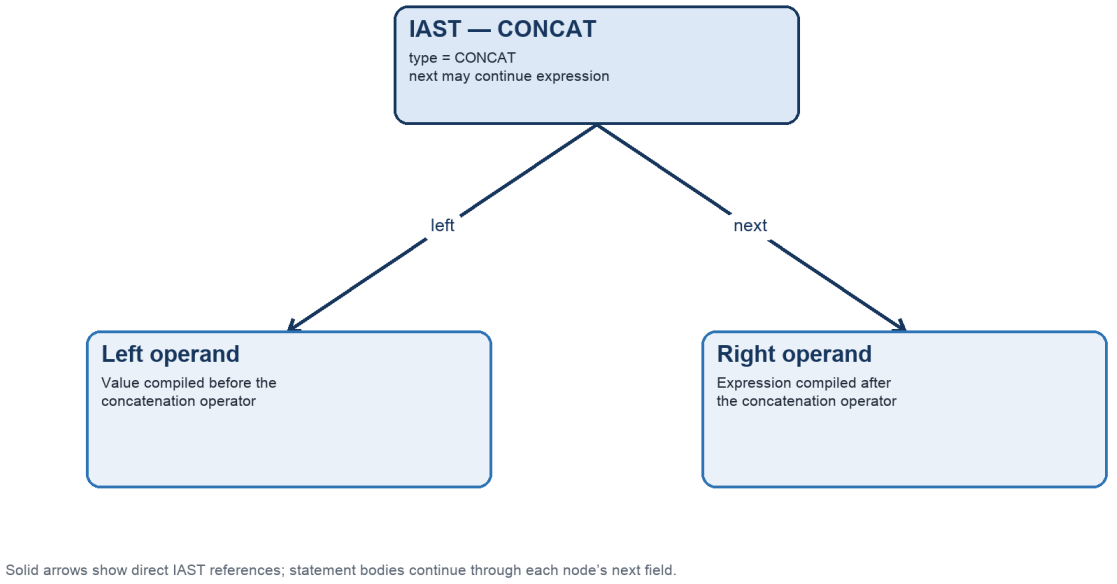

Figure 82. Compiler-produced IAST layout for CONCAT.

Field contract

Field	Meaning
`type`	CONCAT.
`left`	The left-hand operand.
`next`	The right-hand expression.

Implementation note. Unlike comparator nodes, which use left and middle, CONCAT stores its recursively compiled right side in next. This permits a concatenation chain to be represented using the same sequence-capable link used elsewhere in IAST.

Source basis: `YASSCompiler.compileInnerExpression(boolean); LAMEX3 expression evaluation`

NODE 83 / COMPILED STRUCTURE

INHERITS

Identifies the parent structure named by a compiled structure declaration.

```
structure Child inherits Parent … end structure
```

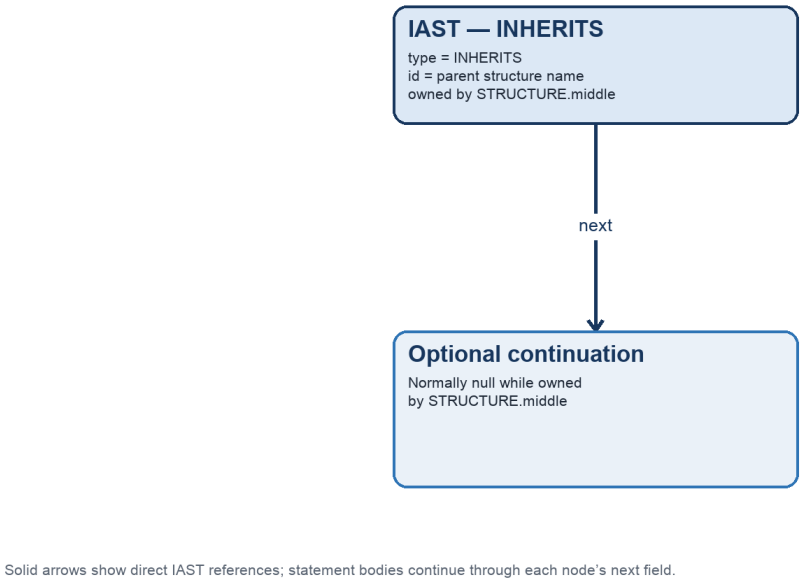

Figure 83. Compiler-produced IAST layout for INHERITS.

Field contract

Field	Meaning
`type`	INHERITS.
`id`	The identifier of the structure being inherited.
`next`	Normally null; ownership is through the containing STRUCTURE node.

Implementation note. The compiler places this node in STRUCTURE.middle. Runtime registers the parent relationship, uses it for inherited member lookup and supplies the parent-dispatch starting point required by super. The IAST layout itself is unchanged by the inheritance overhaul.

Source basis: `YASSCompiler.compileStructure(); ZPERuntimeEnvironment structure-definition handling`

NODE 84 / COMPILED STRUCTURE

IS

Represents one selectable branch within a compiled WHEN or SWITCH structure.

```
when ($value) is 1 do … end when
```

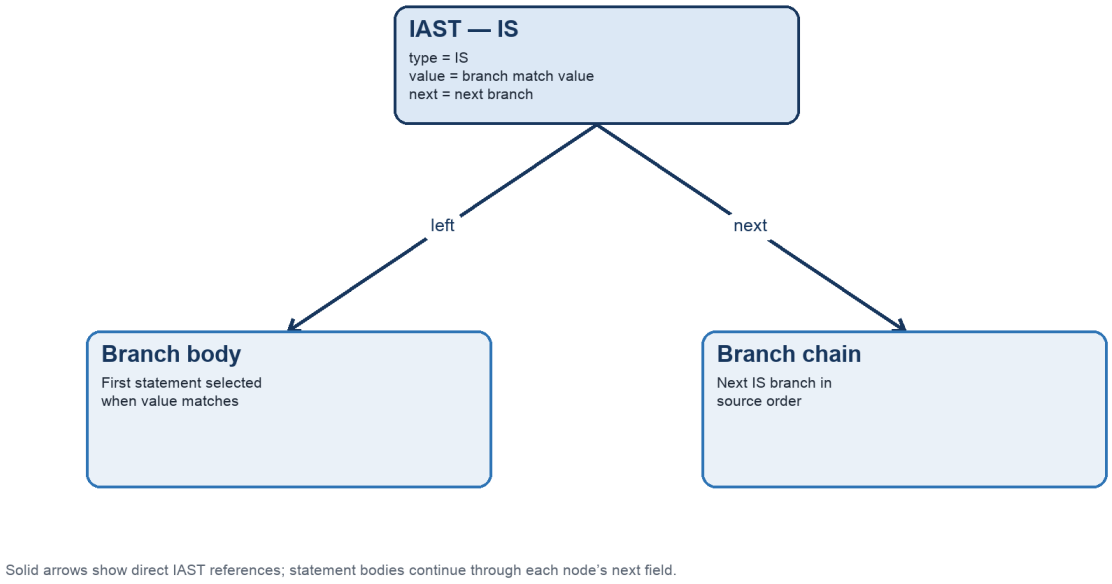

Figure 84. Compiler-produced IAST layout for IS.

Field contract

Field	Meaning
`type`	IS.
`value`	The compiled or retained value identifying this branch.
`left`	The first statement in the branch body.
`next`	The next selectable branch.

Implementation note. Both when/is and switch/case source forms compile to IS branch nodes. Runtime WHEN handling walks this chain, compares each value with the controlling value and traverses the first matching node's left branch.

Source basis: `YASSCompiler.compileWhen(); ZPERuntimeEnvironment.handleWhen(IAST, …)`

NODE 85 / COMPILED STRUCTURE

STATIC

Represents a static function definition owned by its declaring structure.

A static method remains on its structure, is not copied onto instances and, when public, is called through StructureName::method().

```
public static function get_people_list() … end function
```

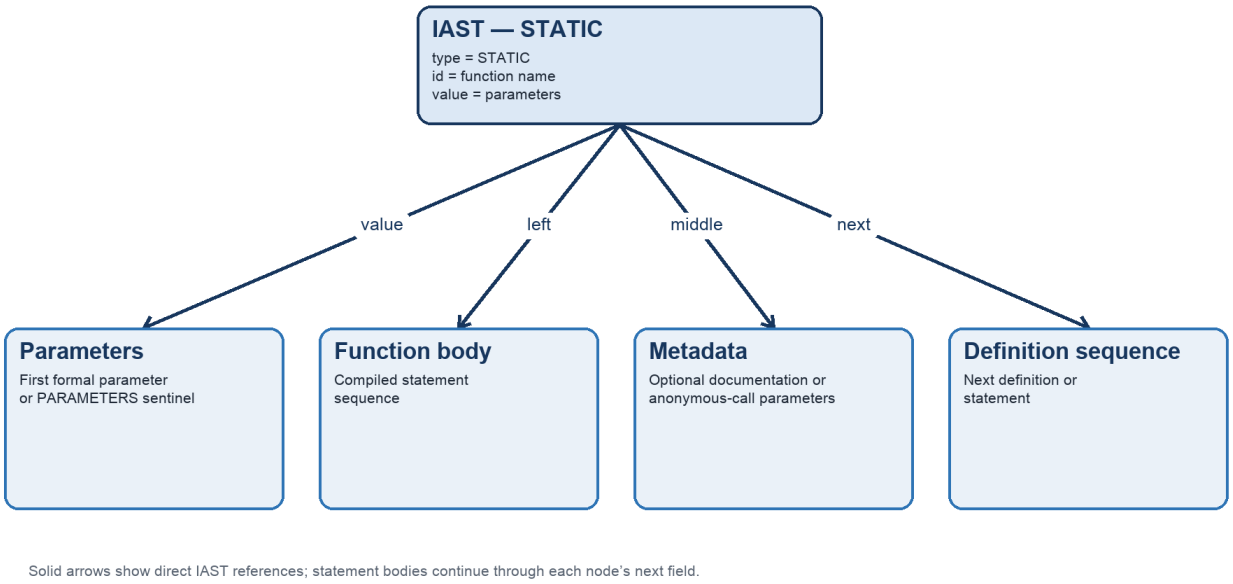

Figure 85. Compiler-produced IAST layout for STATIC.

Field contract

Field	Meaning
`type`	STATIC.
`id`	The declared function name.
`value`	The formal parameter representation.
`left`	The compiled function body.
`middle`	Optional metadata used by the function form.
`returnTypes`	Optional accepted return types.
`scope`	Compiled access level.
`next`	The next node in the containing sequence.

Implementation note. The compiler retags the standard FUNCTION contract as STATIC. AbstractStructure stores it apart from instance members, and scope resolution invokes it without constructing an object. Its implementation was made available in ZPE 1.14.9 (Spurriergate, September 2026).

Source basis: `YASSCompiler.compileFunction(); ZPERuntimeEnvironment.defineFunction(IAST, …)`

NODE 86 / COMPILED STRUCTURE

TO

Wraps the terminal expression used by the specialised FOR_TO loop representation.

```
for ($index = 0 to $limit) … end for
```

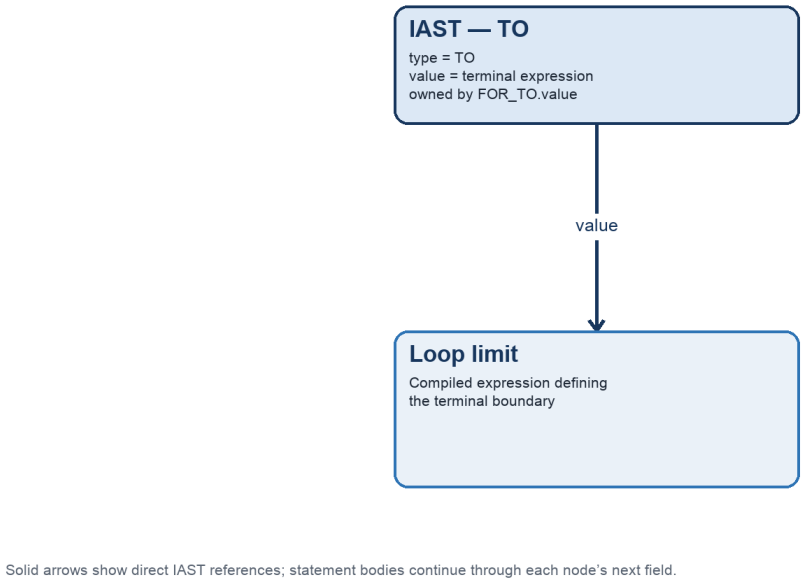

Figure 86. Compiler-produced IAST layout for TO.

Field contract

Field	Meaning
`type`	TO.
`value`	The compiled terminal-boundary expression.
`next`	Normally null; ownership is through FOR_TO.value.

Implementation note. TO survives compilation as a nested node rather than only separating source tokens. The FOR_TO runtime handler unwraps its value field, evaluates the boundary and uses that result to determine direction and termination.

Source basis: `YASSCompiler.compileFor(); YASSCompiler optimisation handling; ZPERuntimeEnvironment.handleForTo(IAST, …)`

NODE 87 / COMPILED STRUCTURE

UNDEFINED

Represents the explicit YASS undefined value within a compiled expression.

```
undefined
```

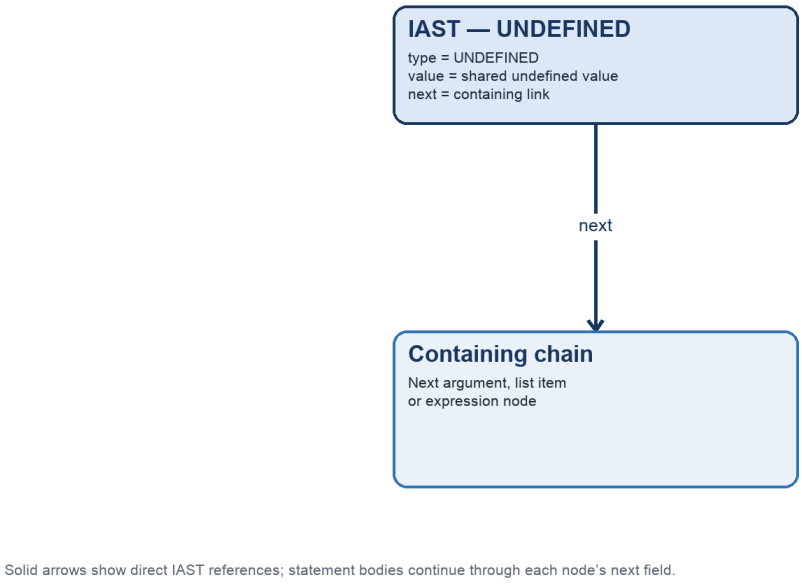

Figure 87. Compiler-produced IAST layout for UNDEFINED.

Field contract

Field	Meaning
`type`	UNDEFINED.
`value`	The shared ZPE undefined value retained by the compiler.
`next`	Optional continuation in the enclosing chain.

Implementation note. UNDEFINED is distinct from NULL: it represents a value that is not defined rather than an explicitly empty value. The compiler stores the shared ZPE.UNDEFINED object in value so expression and fallback handling can preserve that distinction.

Source basis: `YASSCompiler.compileInnerExpression(boolean); LAMEX3 and ZPE undefined-value handling`

NODE 88 / COMPILED STRUCTURE

PLUS

Represents addition as a binary node within a compiled arithmetic expression.

```
$left + $right
```

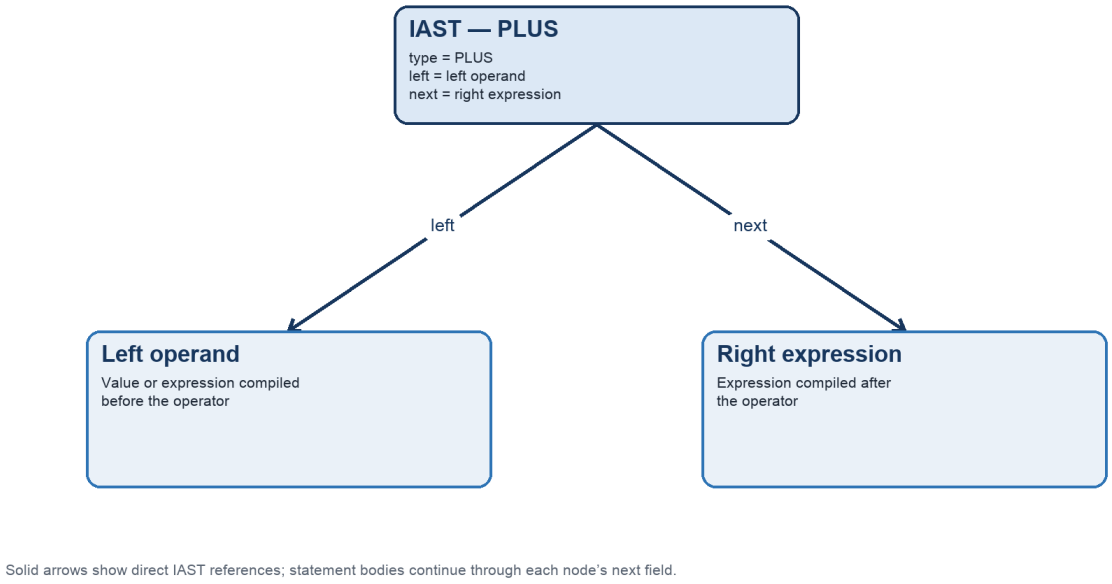

Figure 88. Compiler-produced IAST layout for PLUS.

Field contract

Field	Meaning
`type`	PLUS.
`left`	The operand appearing before the operator.
`next`	The operand or recursively compiled expression appearing after the operator.

Implementation note. The compiler assigns this type from the current parser symbol after recognising a mathematical operator. The right side is stored in next rather than middle, allowing the expression compiler to retain the remaining operator chain.

Source basis: `YASSCompiler.compileInnerExpression(boolean); LAMEX3 arithmetic-expression evaluation`

NODE 89 / COMPILED STRUCTURE

MINUS

Represents subtraction as a binary node within a compiled arithmetic expression.

```
$left - $right
```

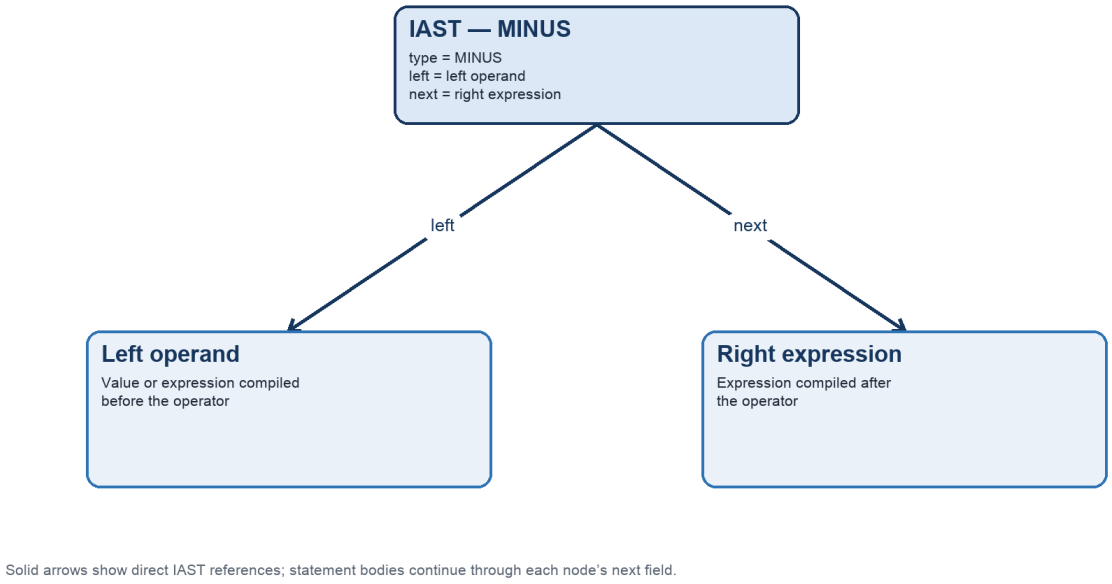

Figure 89. Compiler-produced IAST layout for MINUS.

Field contract

Field	Meaning
`type`	MINUS.
`left`	The operand appearing before the operator.
`next`	The operand or recursively compiled expression appearing after the operator.

Implementation note. The compiler assigns this type from the current parser symbol after recognising a mathematical operator. The right side is stored in next rather than middle, allowing the expression compiler to retain the remaining operator chain.

Source basis: `YASSCompiler.compileInnerExpression(boolean); LAMEX3 arithmetic-expression evaluation`

NODE 90 / COMPILED STRUCTURE

MULT

Represents multiplication as a binary node within a compiled arithmetic expression.

```
$left * $right
```

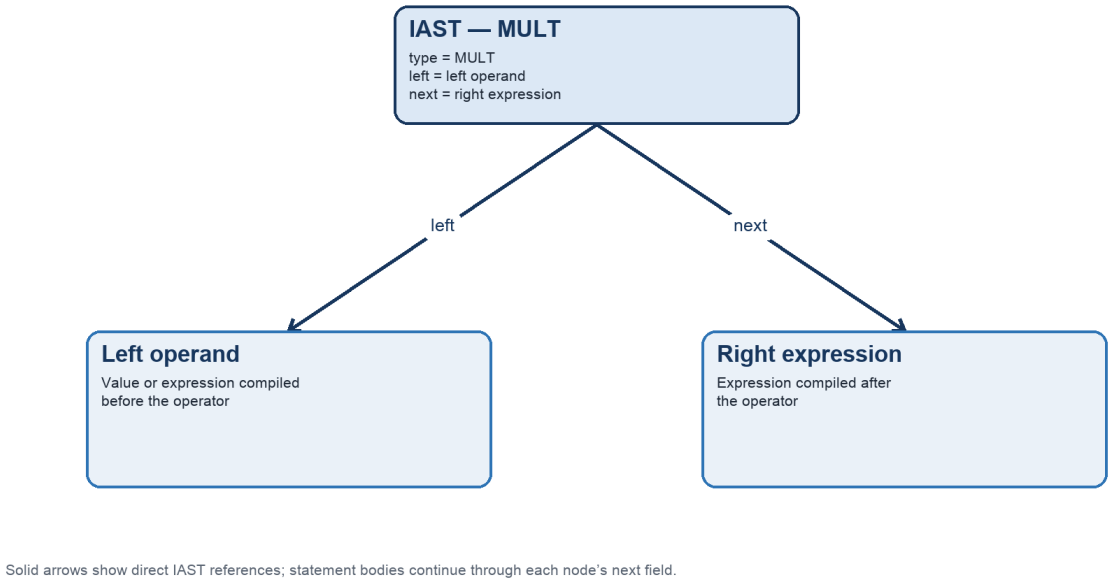

Figure 90. Compiler-produced IAST layout for MULT.

Field contract

Field	Meaning
`type`	MULT.
`left`	The operand appearing before the operator.
`next`	The operand or recursively compiled expression appearing after the operator.

Implementation note. The compiler assigns this type from the current parser symbol after recognising a mathematical operator. The right side is stored in next rather than middle, allowing the expression compiler to retain the remaining operator chain.

Source basis: `YASSCompiler.compileInnerExpression(boolean); LAMEX3 arithmetic-expression evaluation`

NODE 91 / COMPILED STRUCTURE

DIVIDE

Represents division as a binary node within a compiled arithmetic expression.

```
$left / $right
```

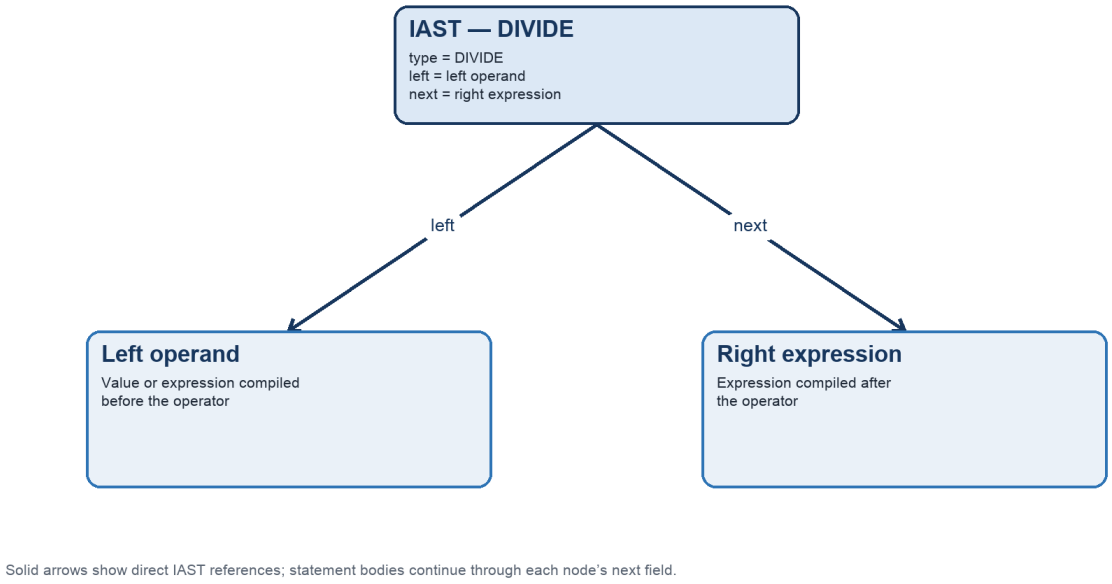

Figure 91. Compiler-produced IAST layout for DIVIDE.

Field contract

Field	Meaning
`type`	DIVIDE.
`left`	The operand appearing before the operator.
`next`	The operand or recursively compiled expression appearing after the operator.

Implementation note. The compiler assigns this type from the current parser symbol after recognising a mathematical operator. The right side is stored in next rather than middle, allowing the expression compiler to retain the remaining operator chain.

Source basis: `YASSCompiler.compileInnerExpression(boolean); LAMEX3 arithmetic-expression evaluation`

NODE 92 / COMPILED STRUCTURE

MODULO

Represents remainder calculation as a binary node within a compiled arithmetic expression.

```
$left % $right
```

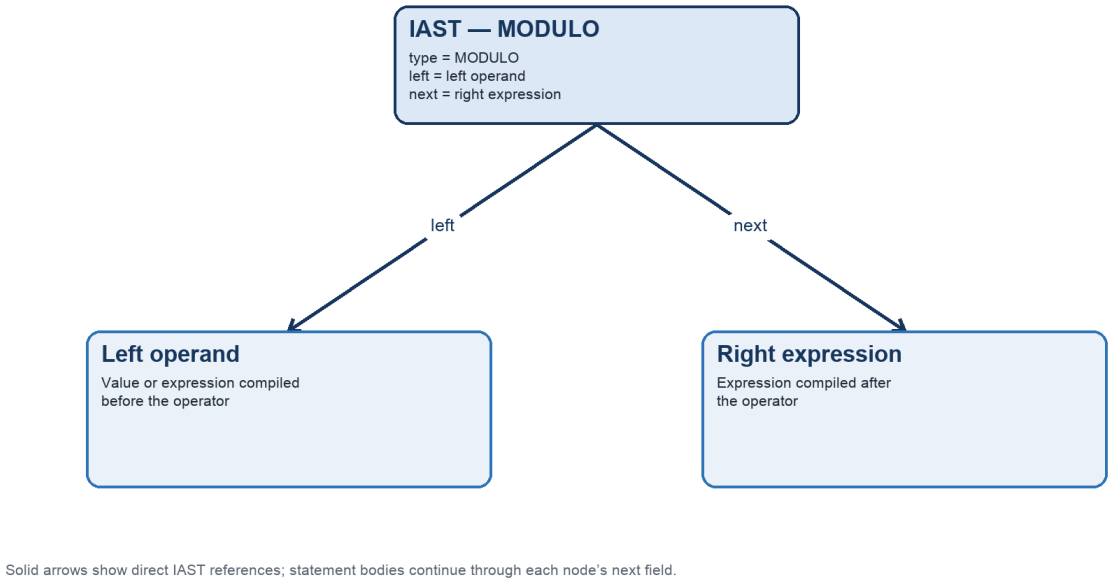

Figure 92. Compiler-produced IAST layout for MODULO.

Field contract

Field	Meaning
`type`	MODULO.
`left`	The operand appearing before the operator.
`next`	The operand or recursively compiled expression appearing after the operator.

Implementation note. The compiler assigns this type from the current parser symbol after recognising a mathematical operator. The right side is stored in next rather than middle, allowing the expression compiler to retain the remaining operator chain.

Source basis: `YASSCompiler.compileInnerExpression(boolean); LAMEX3 arithmetic-expression evaluation`

NODE 93 / COMPILED STRUCTURE

CIRCUMFLEX

Represents exponentiation as a binary node within a compiled arithmetic expression.

```
$left ^ $right
```

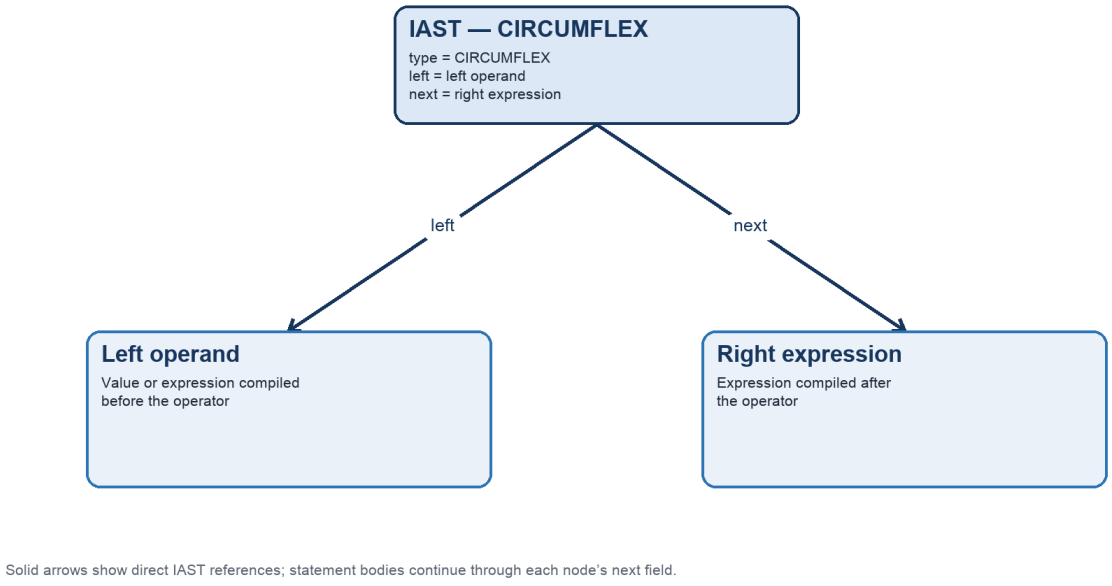

Figure 93. Compiler-produced IAST layout for CIRCUMFLEX.

Field contract

Field	Meaning
`type`	CIRCUMFLEX.
`left`	The operand appearing before the operator.
`next`	The operand or recursively compiled expression appearing after the operator.

Implementation note. The compiler assigns this type from the current parser symbol after recognising a mathematical operator. The right side is stored in next rather than middle, allowing the expression compiler to retain the remaining operator chain.

Source basis: `YASSCompiler.compileInnerExpression(boolean); LAMEX3 arithmetic-expression evaluation`

NODE 94 / COMPILED STRUCTURE

EQUAL

Represents value equality within a compiled logical or conditional expression.

```
$left == $right
```

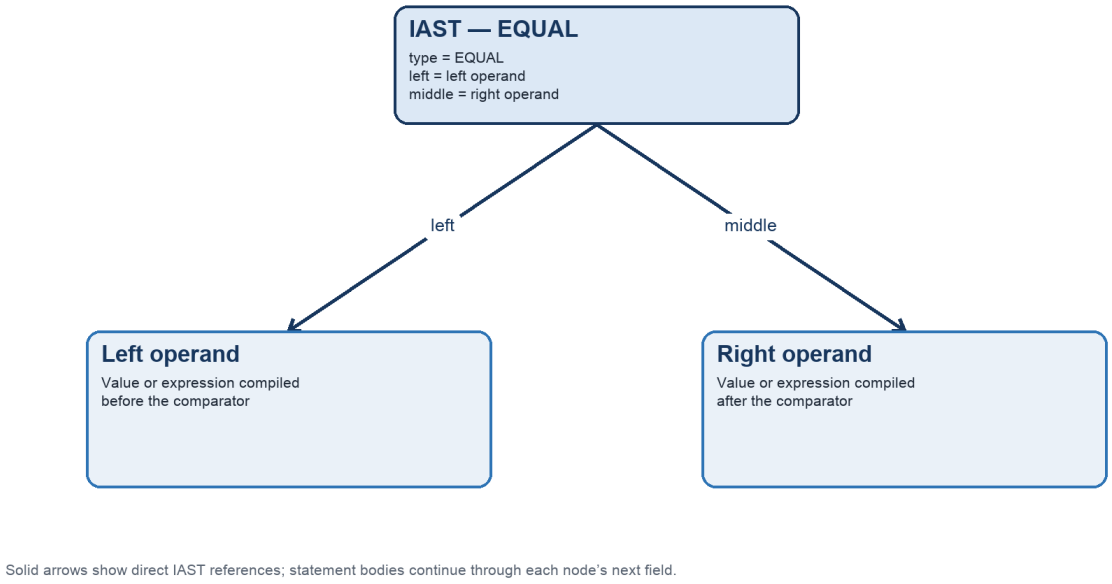

Figure 94. Compiler-produced IAST layout for EQUAL.

Field contract

Field	Meaning
`type`	EQUAL.
`left`	The expression on the left-hand side of the comparison.
`middle`	The expression on the right-hand side of the comparison.
`next`	Normally null unless the node is linked by an enclosing structure.

Implementation note. Comparator nodes are normally stored in LOGICAL_EXPRESSION.value. The compiler also supports shortened joined conditions, in which the previously compiled left operand may be copied into this node before the right operand is compiled.

Source basis: `YASSCompiler.compileLogicallyJoinedExpression(boolean, IAST); LAMEX3 comparison evaluation`

NODE 95 / COMPILED STRUCTURE

EXACTLY_EQUAL

Represents strict value-and-type equality within a compiled logical or conditional expression.

```
$left === $right
```

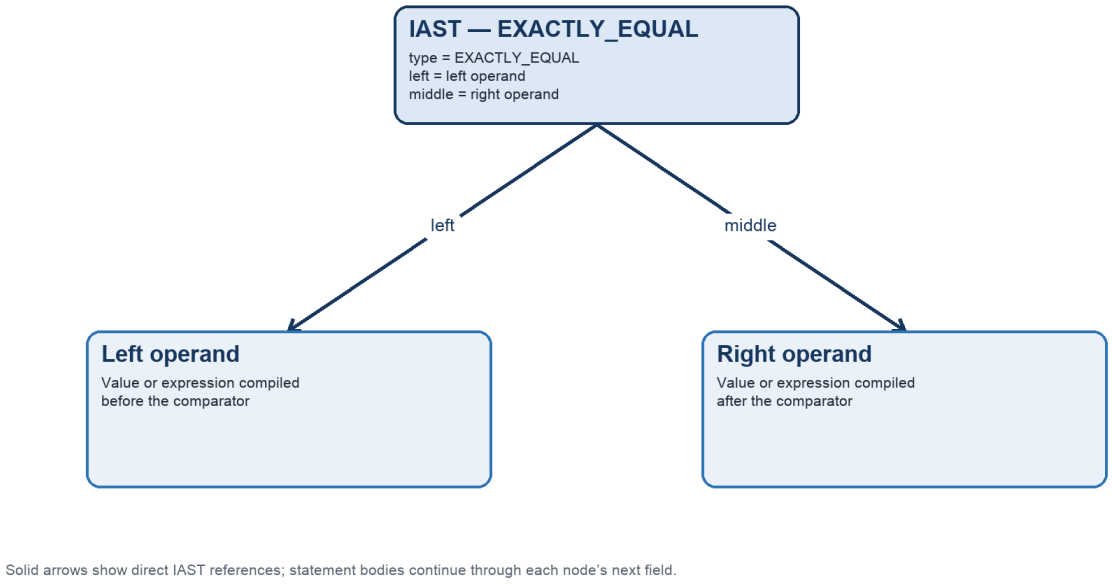

Figure 95. Compiler-produced IAST layout for EXACTLY_EQUAL.

Field contract

Field	Meaning
`type`	EXACTLY_EQUAL.
`left`	The expression on the left-hand side of the comparison.
`middle`	The expression on the right-hand side of the comparison.
`next`	Normally null unless the node is linked by an enclosing structure.

Implementation note. Comparator nodes are normally stored in LOGICAL_EXPRESSION.value. The compiler also supports shortened joined conditions, in which the previously compiled left operand may be copied into this node before the right operand is compiled.

Source basis: `YASSCompiler.compileLogicallyJoinedExpression(boolean, IAST); LAMEX3 comparison evaluation`

NODE 96 / COMPILED STRUCTURE

NEQUAL

Represents value inequality within a compiled logical or conditional expression.

```
$left != $right
```

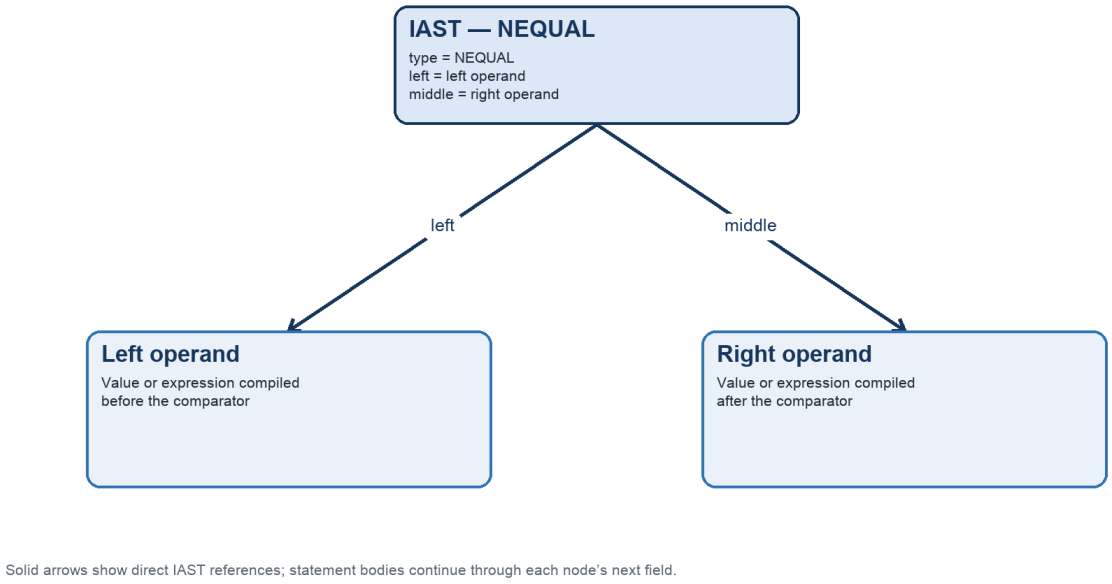

Figure 96. Compiler-produced IAST layout for NEQUAL.

Field contract

Field	Meaning
`type`	NEQUAL.
`left`	The expression on the left-hand side of the comparison.
`middle`	The expression on the right-hand side of the comparison.
`next`	Normally null unless the node is linked by an enclosing structure.

Implementation note. Comparator nodes are normally stored in LOGICAL_EXPRESSION.value. The compiler also supports shortened joined conditions, in which the previously compiled left operand may be copied into this node before the right operand is compiled.

Source basis: `YASSCompiler.compileLogicallyJoinedExpression(boolean, IAST); LAMEX3 comparison evaluation`

NODE 97 / COMPILED STRUCTURE

LT

Represents a less-than comparison within a compiled logical or conditional expression.

```
$left < $right
```

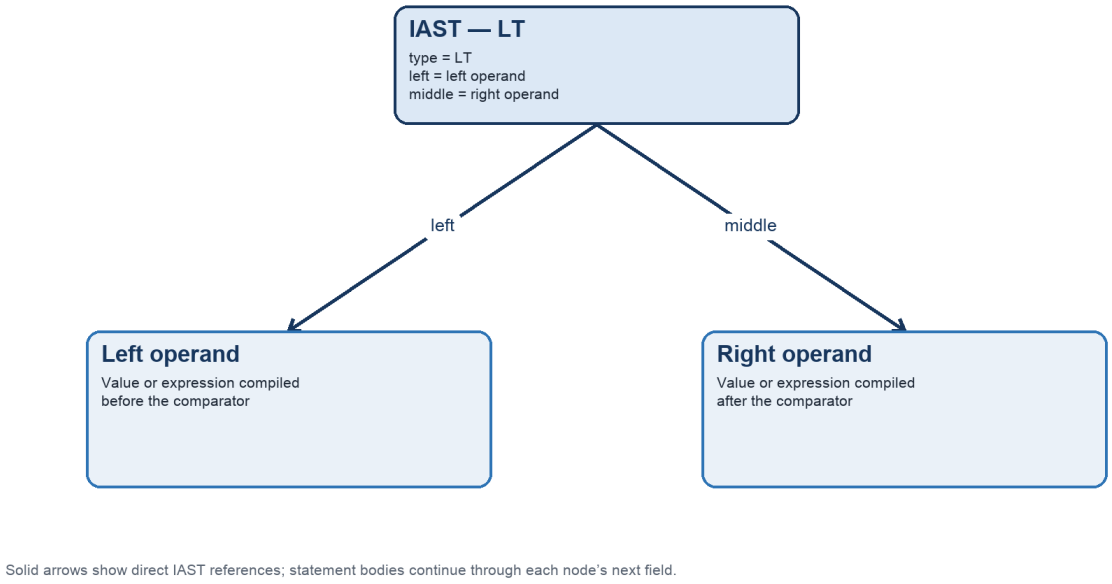

Figure 97. Compiler-produced IAST layout for LT.

Field contract

Field	Meaning
`type`	LT.
`left`	The expression on the left-hand side of the comparison.
`middle`	The expression on the right-hand side of the comparison.
`next`	Normally null unless the node is linked by an enclosing structure.

Implementation note. Comparator nodes are normally stored in LOGICAL_EXPRESSION.value. The compiler also supports shortened joined conditions, in which the previously compiled left operand may be copied into this node before the right operand is compiled.

Source basis: `YASSCompiler.compileLogicallyJoinedExpression(boolean, IAST); LAMEX3 comparison evaluation`

NODE 98 / COMPILED STRUCTURE

GT

Represents a greater-than comparison within a compiled logical or conditional expression.

```
$left > $right
```

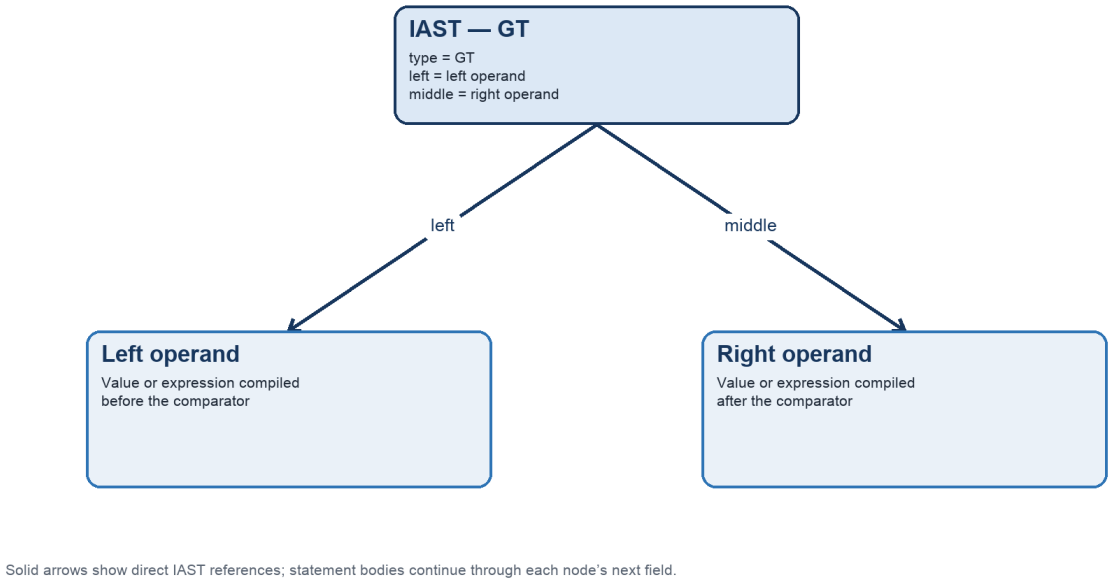

Figure 98. Compiler-produced IAST layout for GT.

Field contract

Field	Meaning
`type`	GT.
`left`	The expression on the left-hand side of the comparison.
`middle`	The expression on the right-hand side of the comparison.
`next`	Normally null unless linked by an enclosing structure.

Implementation note. Comparator nodes are normally stored in LOGICAL_EXPRESSION.value. The runtime evaluates both operands and applies the comparison selected by type.

Source basis: `YASSCompiler.compileLogicallyJoinedExpression(boolean, IAST); LAMEX3 comparison evaluation`

NODE 99 / COMPILED STRUCTURE

LTE

Represents a less-than-or-equal comparison within a compiled logical or conditional expression.

```
$left <= $right
```

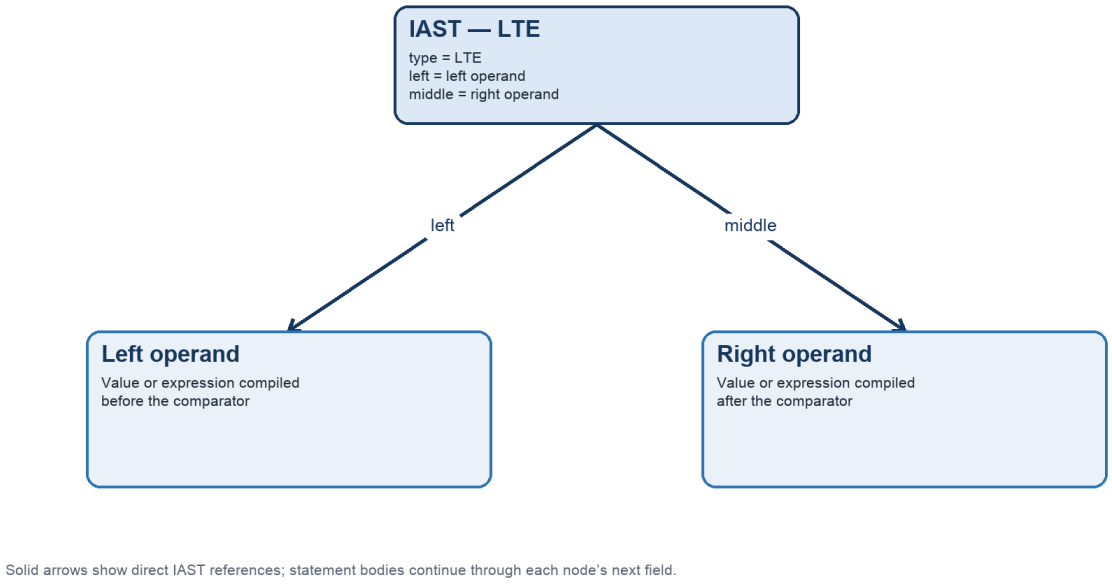

Figure 99. Compiler-produced IAST layout for LTE.

Field contract

Field	Meaning
`type`	LTE.
`left`	The expression on the left-hand side of the comparison.
`middle`	The expression on the right-hand side of the comparison.
`next`	Normally null unless linked by an enclosing structure.

Implementation note. Comparator nodes are normally stored in LOGICAL_EXPRESSION.value. The runtime evaluates both operands and applies the comparison selected by type.

Source basis: `YASSCompiler.compileLogicallyJoinedExpression(boolean, IAST); LAMEX3 comparison evaluation`

NODE 100 / COMPILED STRUCTURE

GTE

Represents a greater-than-or-equal comparison within a compiled logical or conditional expression.

```
$left >= $right
```

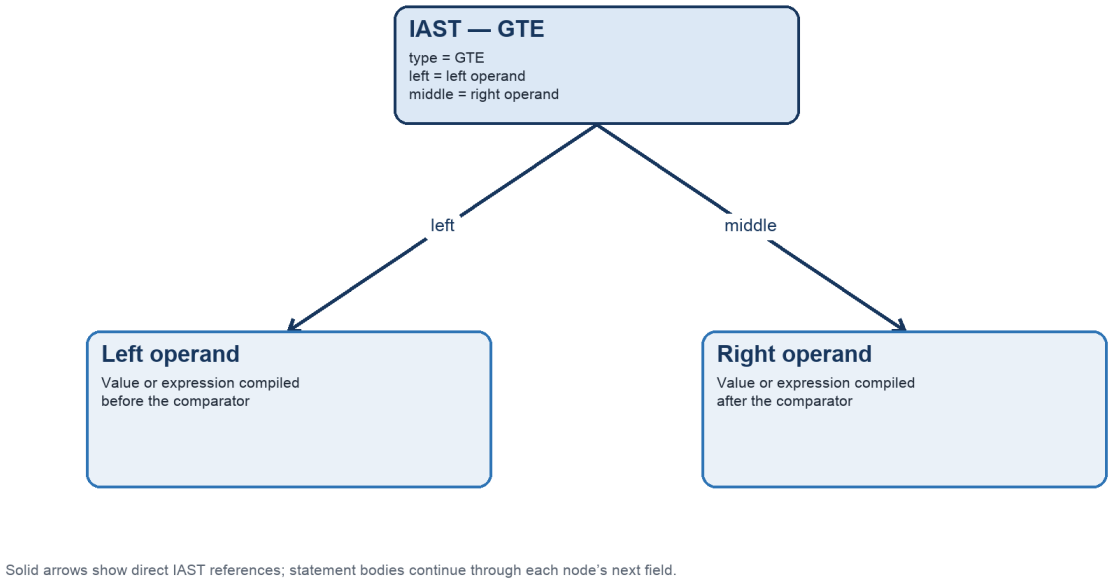

Figure 100. Compiler-produced IAST layout for GTE.

Field contract

Field	Meaning
`type`	GTE.
`left`	The expression on the left-hand side of the comparison.
`middle`	The expression on the right-hand side of the comparison.
`next`	Normally null unless linked by an enclosing structure.

Implementation note. Comparator nodes are normally stored in LOGICAL_EXPRESSION.value. The runtime evaluates both operands and applies the comparison selected by type.

Source basis: `YASSCompiler.compileLogicallyJoinedExpression(boolean, IAST); LAMEX3 comparison evaluation`

NODE 101 / COMPILED STRUCTURE

LAND

Represents logical conjunction between two compiled logical branches.

```
$conditionA && $conditionB
```

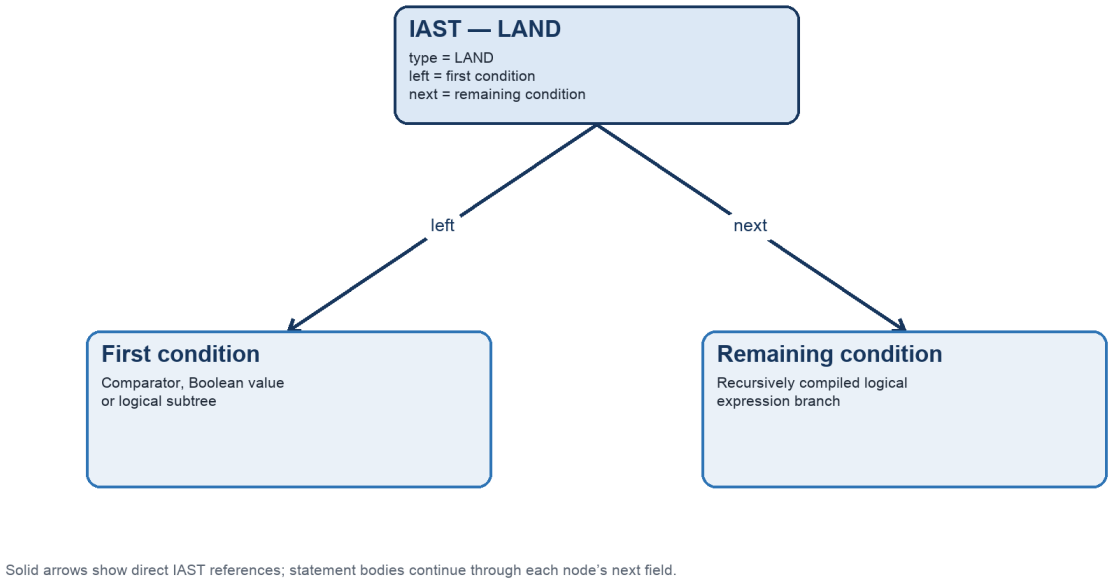

Figure 101. Compiler-produced IAST layout for LAND.

Field contract

Field	Meaning
`type`	LAND.
`left`	The condition appearing before the logical operator.
`next`	The recursively compiled condition appearing after the operator.

Implementation note. The compiler creates this node while joining conditions inside a LOGICAL_EXPRESSION. The right side is recursively represented through next, allowing longer logical chains to retain their source ordering and grouping.

Source basis: `YASSCompiler.compileLogicallyJoinedExpression(boolean, IAST); LAMEX3 logical evaluation`

NODE 102 / COMPILED STRUCTURE

LOR

Represents logical disjunction between two compiled logical branches.

```
$conditionA || $conditionB
```

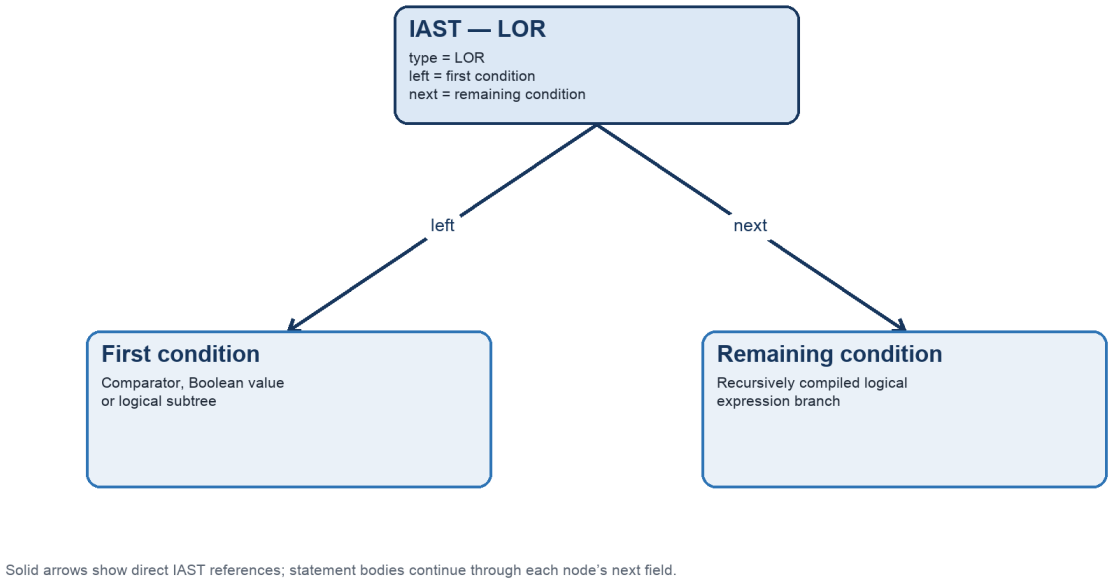

Figure 102. Compiler-produced IAST layout for LOR.

Field contract

Field	Meaning
`type`	LOR.
`left`	The condition appearing before the logical operator.
`next`	The recursively compiled condition appearing after the operator.

Implementation note. The compiler creates this node while joining conditions inside a LOGICAL_EXPRESSION. The right side is recursively represented through next, allowing longer logical chains to retain their source ordering and grouping.

Source basis: `YASSCompiler.compileLogicallyJoinedExpression(boolean, IAST); LAMEX3 logical evaluation`

NODE 103 / COMPILED STRUCTURE

LBRA

Preserves an explicitly parenthesised subexpression as a grouped expression node.

```
($left + $right)
```

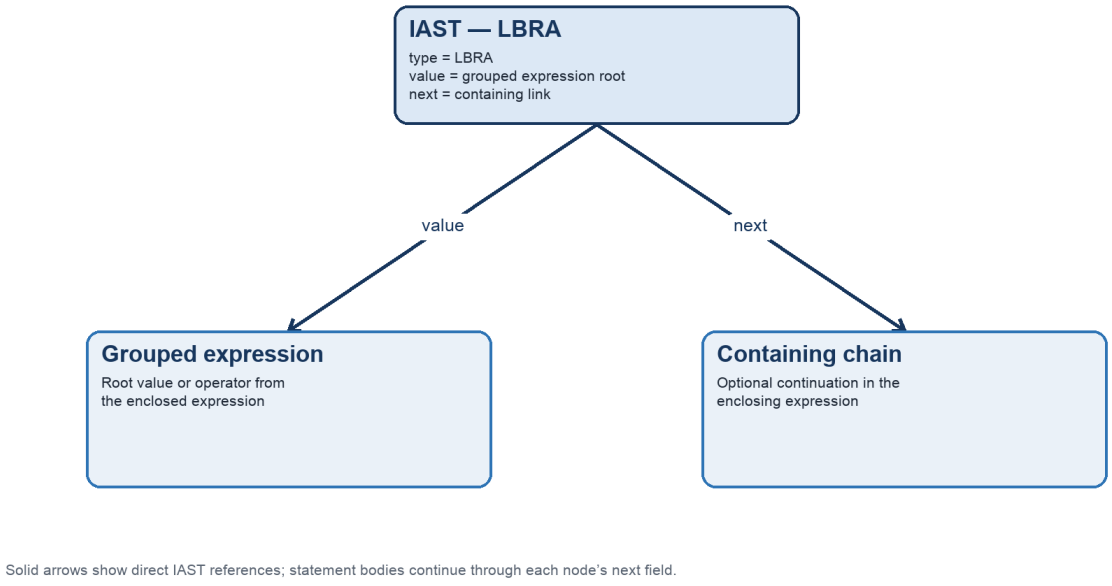

Figure 103. Compiler-produced IAST layout for LBRA.

Field contract

Field	Meaning
`type`	LBRA.
`value`	The compiled root of the expression enclosed by parentheses.
`next`	Optional continuation in the enclosing expression tree.

Implementation note. Although the closing bracket is discarded, the opening-bracket byte is retained as the node type so the enclosed expression remains grouped. Evaluation processes value as a single subexpression before applying surrounding operators.

Source basis: `YASSCompiler.compileInnerExpression(boolean); LAMEX3 grouped-expression evaluation`

APPENDIX A / LANGUAGE DEFINITION

Formal specification of YASS

The official formal specification for the YASS language has been developed using a variation of Backus-Naur Form (BNF). It defines the grammatical structure of valid YASS source programs, including lexical elements, values, expressions, declarations, functions, control-flow constructs, modules, structures and records.

The specification provides a concise and implementation-independent description of the language's surface syntax. It complements the IAST node contracts described in the main whitepaper: the formal grammar defines which source forms may be written, while the node documentation explains how those forms are represented after compilation. Together, they describe the boundary between YASS source code, the YASS Compiler and execution by the ZPE Runtime Environment.

> **Interpretation.** Angle brackets identify nonterminal symbols; ::= means "is defined as"; | separates alternatives; and ε represents an empty production. Repetition is expressed recursively. Where the formal specification and a particular implementation differ, the specification records the intended language grammar and the corresponding compiler version should be consulted.

Purpose of the specification

- Provide a normative reference for the accepted grammatical forms of YASS.
- Support consistent implementation across the compiler, runtime, editor, debugger and transpilers.
- Make additions and compatibility changes to the language explicit and reviewable.
- Give implementers and language users a compact reference independent of parser source code.

APPENDIX A / FORMAL GRAMMAR 01

Lexical elements and fundamental symbols

Nonterminal		Production
⟨varsym⟩	::=	$
⟨math_symb⟩	::=	+ | - | / | * | %
⟨comparator⟩	::=	< | > | <= | >= | != | ==
⟨logical_operator⟩	::=	|| | && | OR | AND
⟨concatenate⟩	::=	&
⟨comma⟩	::=	, | ;
⟨type⟩	::=	string | integer | real | number | boolean | list | map | function | object | record | mixed
⟨identifier⟩	::=	_?[A-Za-z][A-Za-z0-9_]*
⟨import⟩	::=	import ⟨identifier⟩ | import ⟨identifier⟩ as ⟨identifier⟩

APPENDIX A / FORMAL GRAMMAR 02

Program bodies and basic control statements

Nonterminal		Production
⟨body⟩	::=	⟨assert⟩ | ⟨import⟩ | ⟨includes⟩ | ⟨assign⟩ | ⟨function⟩ | ⟨if⟩ | ⟨for⟩ | ⟨for_to⟩ | ⟨for_each⟩ | ⟨until⟩ | ⟨while⟩ | ⟨do_while⟩ | ⟨when⟩ | ⟨lambda_call⟩ | ⟨function_call⟩ | ⟨try⟩ | ⟨object_accessor⟩ | ⟨scope_resolution⟩ | ⟨increment⟩ | ⟨decrement⟩ | ⟨module⟩ | ⟨structure⟩ | ⟨record⟩ | ⟨return⟩ | ⟨break⟩ | ⟨label⟩ | ⟨goto⟩ | ⟨breakpoint⟩ | ⟨scope_block⟩
⟨breakpoint⟩	::=	#breakpoint#
⟨label⟩	::=	label ⟨identifier⟩
⟨goto⟩	::=	goto ⟨identifier⟩
⟨return⟩	::=	return ⟨expression⟩ | return
⟨break⟩	::=	break

APPENDIX A / FORMAL GRAMMAR 03

Variables, literals and values

Nonterminal		Production
⟨variable⟩	::=	⟨varsym⟩⟨identifier⟩ | ⟨identifier⟩
⟨increment⟩	::=	++⟨variable⟩ | ⟨variable⟩++
⟨decrement⟩	::=	--⟨variable⟩ | ⟨variable⟩--
⟨undefined⟩	::=	undefined
⟨null⟩	::=	null
⟨boolean⟩	::=	true | false
⟨integer⟩	::=	((-?[1-9]+)|[0-9][0-9]*)(E[0-9]+)?
⟨double⟩	::=	[+-]?(0|[1-9]+)\.\d+([eE][+-]?\d+)?
⟨hex⟩	::=	0x[0-9A-f]+
⟨octal⟩	::=	0[1-7][0-7]*
⟨string⟩	::=	".*" | '.*'
⟨list⟩	::=	[⟨actual_parameters⟩]
⟨map⟩	::=	{=>} | {⟨map_pairs⟩}
⟨internal_doc⟩	::=	<< (.*) >>
⟨concatenation⟩	::=	⟨string⟩ ⟨concatenate⟩ ⟨string⟩ | ⟨string⟩ ⟨concatenate⟩ ⟨concatenation⟩
⟨template⟩	::=	` ⟨template_parts⟩ `
⟨template_parts⟩	::=	⟨template_part⟩ ⟨template_parts⟩ | ε
⟨template_part⟩	::=	⟨template_text⟩ | ⟨template_eval⟩
⟨template_eval⟩	::=	{{ ⟨template_expr⟩ }}
⟨template_expr⟩	::=	⟨expression⟩
⟨template_text⟩	::=	any sequence of characters excluding ` and the opening sequence {{
⟨value⟩	::=	⟨concatenation⟩ | ⟨string⟩ | ⟨integer⟩ | ⟨double⟩ | ⟨boolean⟩ | ⟨null⟩ | ⟨list⟩ | ⟨map⟩ | ⟨hex⟩ | ⟨octal⟩ | ⟨object⟩ | ⟨lambda1⟩ | ⟨lambda2⟩ | ⟨lambda_function⟩ | ⟨increment⟩ | ⟨decrement⟩ | ⟨internal_doc⟩ | ⟨template⟩
⟨count⟩	::=	count(⟨value⟩)
⟨cast⟩	::=	cast(⟨value⟩, ⟨type⟩)
⟨match_inner⟩	::=	⟨value⟩ => ⟨value⟩ ⟨match_inner⟩ | ⟨value⟩ => ⟨value⟩, ⟨match_inner⟩ | ⟨value⟩ => ⟨value⟩ | ε
⟨match_expression⟩	::=	match(⟨value⟩ : ⟨match_inner⟩)
⟨scope_block⟩	::=	{{ ⟨body⟩ }}

APPENDIX A / FORMAL GRAMMAR 04

Functions and parameter forms

Nonterminal		Production
⟨power⟩	::=	⟨value⟩ ^ ⟨value⟩
⟨construct_body⟩	::=	⟨body⟩ ⟨construct_body⟩ | ε
⟨formal_parameters⟩	::=	⟨variable⟩, ⟨formal_parameters⟩ | ⟨variable⟩ | ⟨type⟩ ⟨variable⟩, ⟨formal_parameters⟩ | ⟨type⟩ ⟨variable⟩ | ⟨variable⟩ ... | ε
⟨lambda1⟩	::=	function (⟨formal_parameters⟩) ⟨construct_body⟩ | function (⟨formal_parameters⟩) {⟨construct_body⟩}
⟨lambda2⟩	::=	(⟨formal_parameters⟩) => ⟨construct_body⟩ | (⟨formal_parameters⟩) => {⟨construct_body⟩}
⟨lambda_function⟩	::=	⟨identifier⟩
⟨function_type⟩	::=	: ⟨type⟩ | ε
⟨function⟩	::=	function ⟨identifier⟩ (⟨formal_parameters⟩) ⟨function_type⟩ ⟨construct_body⟩ end function | function ⟨identifier⟩ (⟨formal_parameters⟩) ⟨function_type⟩ {⟨construct_body⟩}
⟨function_call⟩	::=	⟨identifier⟩ (⟨actual_parameters⟩)
⟨lambda_call⟩	::=	⟨variable⟩ (⟨actual_parameters⟩)

APPENDIX A / FORMAL GRAMMAR 05

Expressions and assertions

Nonterminal		Production
⟨term⟩	::=	⟨identifier⟩ | ⟨variable⟩ | ⟨increment⟩ | ⟨decrement⟩ | ⟨value⟩ | ⟨lambda_call⟩ | ⟨function_call⟩ | ⟨object_accessor⟩ | ⟨scope_resolution⟩ | ⟨power⟩ | ⟨match_expression⟩ | ⟨count⟩ | ⟨cast⟩ | (⟨term⟩)
⟨ternary_op⟩	::=	⟨expr_inner⟩ ? ⟨value⟩ : ⟨value⟩
⟨optional_value⟩	::=	⟨expr_inner⟩ ?? ⟨value⟩
⟨expr_inner⟩	::=	⟨term⟩ ⟨math_symb⟩ ⟨term⟩ | ⟨term⟩ ⟨comparator⟩ ⟨term⟩ | ⟨term⟩ ⟨logical_operator⟩ ⟨term⟩ | ⟨term⟩ ⟨concatenate⟩ ⟨term⟩ | ⟨term⟩ | ε
⟨expression⟩	::=	⟨expr_inner⟩ | ⟨ternary_op⟩ | ⟨optional_value⟩ | ε
⟨index⟩	::=	⟨expression⟩ [⟨expression⟩]
⟨assert⟩	::=	assert(⟨expression⟩)

APPENDIX A / FORMAL GRAMMAR 06

Collections, assignment and arrow expressions

Nonterminal		Production
⟨map_pairs⟩	::=	⟨expression⟩ => ⟨expression⟩, ⟨map_pairs⟩ | ⟨expression⟩ => ⟨expression⟩ | ε
⟨actual_parameters⟩	::=	⟨expression⟩, ⟨actual_parameters⟩ | ⟨expression⟩ | ⟨arrow_expression⟩ | ε
⟨assign_variable⟩	::=	let ⟨variable⟩ | ⟨variable⟩ | ⟨const⟩
⟨const⟩	::=	const ⟨identifier⟩ | define ⟨identifier⟩
⟨assign⟩	::=	⟨assign_variable⟩ = ⟨expression⟩ | ⟨assign_variable⟩ = new ⟨identifier⟩ (⟨expression⟩) | ⟨assign_variable⟩ = create new ⟨identifier⟩ record | ⟨assign_variable⟩ = new ⟨identifier⟩ record | let ⟨variable⟩ be ⟨expression⟩ | declare ⟨variable⟩ as ⟨type⟩ = ⟨expression⟩
⟨arrow_parameters⟩	::=	⟨identifier⟩ | ⟨identifier⟩ , ⟨arrow_parameters⟩
⟨arrow_expression⟩	::=	⟨arrow_parameters⟩ => ⟨expression⟩
⟨includes⟩	::=	includes ⟨string⟩

APPENDIX A / FORMAL GRAMMAR 07

Conditional selection

Nonterminal		Production
⟨if⟩	::=	if (⟨expression⟩) ⟨construct_body⟩ ⟨else_if⟩ end if | if (⟨expression⟩) {⟨construct_body⟩} ⟨else_if⟩
⟨else_if⟩	::=	elseif (⟨expression⟩) ⟨construct_body⟩ ⟨else_if⟩ | elseif (⟨expression⟩) {⟨construct_body⟩} ⟨else_if⟩ | ⟨else⟩
⟨else⟩	::=	else ⟨construct_body⟩ | else {⟨construct_body⟩} | ε
⟨when⟩	::=	when (⟨variable⟩) ⟨is⟩ ⟨otherwise⟩ end when | when (⟨variable⟩) { ⟨is⟩ ⟨otherwise⟩ }
⟨is⟩	::=	is ⟨value⟩ do ⟨construct_body⟩ ⟨is⟩ | is ⟨value⟩ ⟨construct_body⟩ ⟨is⟩ | ε
⟨otherwise⟩	::=	otherwise do ⟨construct_body⟩ | otherwise ⟨construct_body⟩ | ε

APPENDIX A / FORMAL GRAMMAR 08

Iteration

Nonterminal		Production
⟨for_increment⟩	::=	⟨integer⟩ | ⟨expression⟩
⟨for_variable⟩	::=	⟨variable⟩ | ⟨assign⟩
⟨for⟩	::=	for (⟨for_variable⟩ ⟨comma⟩ ⟨expression⟩ ⟨comma⟩ ⟨for_increment⟩) ⟨construct_body⟩ end for | for (⟨for_variable⟩ ⟨comma⟩ ⟨expression⟩ ⟨comma⟩ ⟨for_increment⟩) {⟨construct_body⟩}
⟨for_to_start⟩	::=	⟨for_variable⟩
⟨for_to_end⟩	::=	⟨expression⟩
⟨for_to⟩	::=	for (⟨for_to_start⟩ to ⟨for_to_end⟩) ⟨construct_body⟩ end for | for (⟨for_to_start⟩ to ⟨for_to_end⟩) {⟨construct_body⟩}
⟨break_when⟩	::=	break when ⟨variable⟩ is ⟨expression⟩ | ε
⟨as_expression⟩	::=	⟨expression⟩ as ⟨variable⟩ | ⟨expression⟩ as ⟨variable⟩ => ⟨variable⟩
⟨for_each_as⟩	::=	for each (⟨as_expression⟩ ⟨break_when⟩) ⟨construct_body⟩ end for | for each (⟨as_expression⟩ ⟨break_when⟩) {⟨construct_body⟩}
⟨for_each_in⟩	::=	for each (⟨variable⟩ in ⟨expression⟩ ⟨break_when⟩) ⟨construct_body⟩ end for | for each (⟨variable⟩ in ⟨expression⟩ ⟨break_when⟩) {⟨construct_body⟩}
⟨for_each⟩	::=	⟨for_each_as⟩ | ⟨for_each_in⟩
⟨until⟩	::=	loop until (⟨expression⟩) ⟨construct_body⟩ end loop | loop until (⟨expression⟩) {⟨construct_body⟩}
⟨while⟩	::=	while (⟨expression⟩) ⟨construct_body⟩ end while | while (⟨expression⟩) {⟨construct_body⟩}
⟨do_while⟩	::=	do ⟨construct_body⟩ loop while (⟨expression⟩) | do {⟨construct_body⟩} while (⟨expression⟩)

APPENDIX A / FORMAL GRAMMAR 09

Exceptions and object access

Nonterminal		Production
⟨try⟩	::=	try ⟨construct_body⟩ ⟨catch⟩ end try | try {⟨construct_body⟩} ⟨catch⟩
⟨catch⟩	::=	catch (⟨variable⟩) ⟨construct_body⟩ | catch (⟨variable⟩) {⟨construct_body⟩} | ε
⟨object_inner⟩	::=	⟨identifier⟩ : ⟨value⟩, ⟨object_inner⟩ | ⟨string⟩ : ⟨value⟩, ⟨object_inner⟩ | ⟨identifier⟩ : ⟨value⟩ | ⟨string⟩ : ⟨value⟩
⟨object⟩	::=	{⟨object_inner⟩} | { }
⟨object_accessor⟩	::=	⟨object_root⟩ -> ⟨object_accessor⟩ | ⟨object_root⟩ -> ⟨variable⟩ | ⟨object_root⟩ -> ⟨function_call⟩ | ⟨object_root⟩ -> ⟨lambda_call⟩
⟨object_root⟩	::=	⟨variable⟩ | this | self | super
⟨scope_resolution⟩	::=	⟨identifier⟩ :: ⟨identifier⟩ (⟨actual_parameters⟩)

APPENDIX A / FORMAL GRAMMAR 10

Modules, structures and records

Nonterminal		Production
⟨inherits⟩	::=	(inherits | extends) ⟨identifier⟩ | ε
⟨namespace⟩	::=	namespace ⟨identifier⟩ | ε
⟨module⟩	::=	module ⟨identifier⟩ ⟨namespace⟩ ⟨construct_body⟩ end module | module ⟨identifier⟩ { ⟨namespace⟩ ⟨construct_body⟩ }
⟨structure_inner⟩	::=	⟨structure_member⟩ ⟨structure_inner⟩ | ε
⟨structure_member⟩	::=	⟨variable⟩ | ⟨assign⟩ | ⟨function⟩ | ⟨method⟩
⟨structure_id⟩	::=	structure | struct | class
⟨structure⟩	::=	⟨structure_id⟩ ⟨identifier⟩ ⟨inherits⟩ ⟨namespace⟩ ⟨structure_inner⟩ end structure | ⟨structure_id⟩ ⟨identifier⟩ ⟨inherits⟩ { ⟨namespace⟩ ⟨structure_inner⟩ }
⟨record_inner⟩	::=	⟨record_field⟩, ⟨record_inner⟩ | ⟨record_field⟩ | ε
⟨record_field⟩	::=	⟨type⟩ ⟨identifier⟩ | ⟨type⟩ ⟨identifier⟩ = ⟨expression⟩
⟨record⟩	::=	record ⟨identifier⟩ {⟨record_inner⟩} | record ⟨identifier⟩ is {⟨record_inner⟩} | record structure ⟨identifier⟩ {⟨record_inner⟩}
⟨method⟩	::=	⟨method_modifiers⟩ function ⟨identifier⟩ (⟨formal_parameters⟩) ⟨function_type⟩ ⟨construct_body⟩ end function | ⟨method_modifiers⟩ function ⟨identifier⟩ (⟨formal_parameters⟩) ⟨function_type⟩ {⟨construct_body⟩}
⟨method_modifiers⟩	::=	⟨scope⟩ static | ⟨scope⟩ | static | ε
⟨scope⟩	::=	public | private | protected | friendly

Publication, ownership and governance

This appendix defines the status, authority, maintenance and publication context of the YASS technical whitepaper and its supporting formal specification.

Document status

This document forms part of the official technical documentation for the YASS programming language and the ZPE Programming Environment. It describes the internal structures produced by the YASS Compiler and consumed by the ZPE Runtime Environment at the time of publication.

The information contained within this document reflects the implementation of the corresponding ZPE release. As YASS and ZPE continue to develop, individual language features, compiler structures and runtime behaviours may be extended or revised. Where changes affect an established IAST contract, the relevant documentation should be updated alongside the implementation.

Authority and implementation

The YASS Compiler implementation is the authoritative source for determining which IAST nodes are generated and how their fields are populated. The ZPE Runtime Environment is the authoritative implementation of the execution semantics applied to those compiled structures.

The formal language specification included in Appendix A describes the intended grammar of YASS. Where an inconsistency is discovered between the published specification, the compiler and the runtime, it should be treated as a documentation or implementation issue requiring review. Such an inconsistency does not, by itself, redefine the intended language.

> **Contract principle.** The formal grammar defines accepted source forms; the compiler defines their IAST representation; and the runtime defines how that representation is executed.

APPENDIX B / COMPATIBILITY AND REVISION

Compatibility and revision

Compatibility

The IAST structures documented in this whitepaper establish an internal contract between the compiler and runtime. They may also be consumed by other ZPE components, including the debugger, profiler, transpilers, ZIDE, YASS Unfold, compiled-program loaders and development tools.

Changes to a documented node layout may therefore have consequences beyond the compiler and runtime. Any structural change should be assessed for compatibility with all components that create, inspect, serialise, transform or execute an IAST.

Compiled programs are not guaranteed to remain compatible indefinitely across versions of ZPE where the underlying executable format or IAST contract has changed. Compatibility guarantees, where provided, are determined by the relevant ZPE release and executable-format documentation.

Versioning and revision

This whitepaper should be associated with the ZPE and YASS versions against which its contents were verified. Revisions may be issued when:

- new language constructs are introduced;
- the compiler begins emitting a new IAST node type;
- an existing node layout changes;
- runtime interpretation of a node changes;
- the formal grammar is amended; or
- an omission, ambiguity or technical error is identified.

Minor editorial corrections that do not alter the documented technical contract may be made without indicating a language change. Structural or semantic revisions should be recorded in the document revision history.

APPENDIX B / RIGHTS AND LIMITATIONS

Rights and limitations

Intellectual property

YASS, ZPE, the ZPE Programming Environment, the ZPE Runtime Environment, ZPE Native, ZPEX, ZIDE, YASS Unfold and their associated documentation are developed and maintained by Jamie Balfour.

Unless otherwise stated, the contents of this document, its diagrams and its supporting materials are protected by copyright. No transfer of ownership or intellectual-property rights is made through publication of this technical documentation.

Permission to read, reference and cite this document does not imply permission to reproduce, modify, redistribute or commercially exploit the ZPE implementation or its source code. Any licence applying to a distributed ZPE component is governed by the terms supplied with that component.

Third-party technologies, product names and trademarks remain the property of their respective owners. Their inclusion within this document is for technical identification only and does not imply endorsement, sponsorship or affiliation.

Disclaimer

This document has been prepared with care to provide an accurate technical description of YASS and ZPE. Nevertheless, it is supplied for informational and engineering-reference purposes and may contain errors, omissions or descriptions of functionality that is experimental, reserved or subject to change.

No warranty is given that the information is complete or suitable for a particular purpose. Users and implementers should verify behaviour against the relevant ZPE release before relying upon it in production, security-sensitive or compatibility-critical systems.

APPENDIX B / CITATION AND FURTHER INFORMATION

Citation and further information

Citation

When citing this whitepaper, the citation should identify the document title, author, publication or revision date, and the applicable ZPE and YASS versions.

> **Suggested citation.** Balfour, Jamie. YASS Abstract Syntax Tree Generation: Compiler and Runtime Node Contracts. ZPE Programming Environment, [revision date]. Applicable to ZPE [version] and YASS [version].

Further information

Current information about YASS, ZPE, supporting tools and official documentation is available from the ZPE project pages on Jamie Balfour's website.

Technical corrections and documentation issues should include the affected node or grammar production, the applicable ZPE build, and a concise description of the observed inconsistency.

> **Contact details.** Formal publication and project contact information is provided separately in Appendix C.

APPENDIX C / CONTACT INFORMATION

Contact information

Enquiries concerning YASS, ZPE, this technical whitepaper or the associated project documentation may be directed to the project developer and maintainer.

Contact field	Details
Name	Jamie Balfour
Email	jamie@balfour.scot
Website	www.jamiebalfour.scot
ZPE project	www.jamiebalfour.scot/projects/zpe/
GitHub	github.com/jamiebalfour04

When reporting a technical issue

To help identify and reproduce a compiler, runtime or documentation issue, please include:

- the applicable ZPE build and YASS language version;
- the relevant IAST node, grammar production or language feature;
- a minimal source example where practical; and
- the observed behaviour and the behaviour that was expected.

Project information. The ZPE project page provides the current project overview, documentation and related information. Public source repositories and supporting projects are available through GitHub.

APPENDIX D / DOCUMENT CHANGES

Document changes

Date	Version	YASS	Changes
25/07/2026	1.0	26.8	Initial development of this document.
09/08/2026	1.1	26.9	Documented super parent dispatch, operational static structure methods, structure-qualified calls, instance/static ownership rules and the corresponding formal grammar.