CHAPTER 27

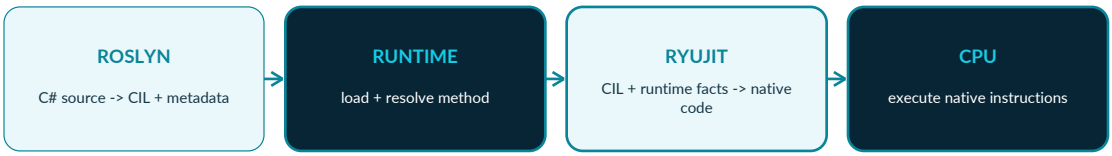
JIT Compilation

The runtime now has the method body. The next machine is a compiler - but not the same compiler that read your C# source.

Chapter 26 ended with a loaded assembly whose metadata and CIL are available to the runtime. That still does not give the CPU native instructions for every method. In ordinary CoreCLR execution, RyuJIT compiles methods when executable native code is needed.

This chapter is intentionally larger than the surrounding chapters. A JIT is easier to understand when it is placed inside the wider science of compilers: lexical analysis, parsing, semantic analysis, intermediate representation, optimization, lowering, register allocation, and code generation. But one boundary will be repeated until it becomes automatic: the C# compiler performs lexical and syntactic analysis; the JIT begins later, from CIL and runtime metadata.

FIGURE 27.1 - TWO COMPILERS, TWO DIFFERENT INPUT LANGUAGES



CENTRAL BOUNDARY

The JIT is a compiler, but it is not a C# lexer/parser. Compiler theory still applies because RyuJIT performs its own front-end work over CIL: it discovers basic blocks, imports stack-based IL into an internal representation, analyzes data/control flow, optimizes, lowers, allocates registers, and emits native code.

CHAPTER 27

The Entire Journey Is a Compiler Pipeline in Two Acts

Source-language compilation and runtime compilation solve different translation problems.

A traditional compiler is often explained as a front end, a middle end, and a back end. .NET makes the separation unusually visible because the source compiler and runtime compiler are different programs separated by a standardized intermediate language.

Roslyn understands C# grammar and C# semantics. RyuJIT understands CIL semantics, runtime type information, target ABI rules, and the instruction set of the current machine. The assembly is the contract between those two acts.

FIGURE 27.2 - FRONT END, MIDDLE END, BACK END ACROSS THE .NET TOOLCHAIN

STAGE	MAIN INPUT	MAIN OUTPUT	OWNED BY
Lex + parse	C# text	syntax tree	Roslyn
Declare + bind	syntax + references	symbols / semantic meaning	Roslyn
Lower + emit	bound program	CIL + metadata	Roslyn
Import	CIL + runtime facts	JIT IR + CFG	RyuJIT
Optimize	JIT IR	improved IR	RyuJIT
Lower + allocate + emit	IR + target ABI	native code + runtime tables	RyuJIT

WHY THIS MATTERS

If lexical/syntactic analysis is attributed to the JIT, the mental model becomes wrong at the first boundary. We will study those compiler principles here because they explain the family resemblance - then mark exactly where Roslyn ends and RyuJIT begins.

CHAPTER 27

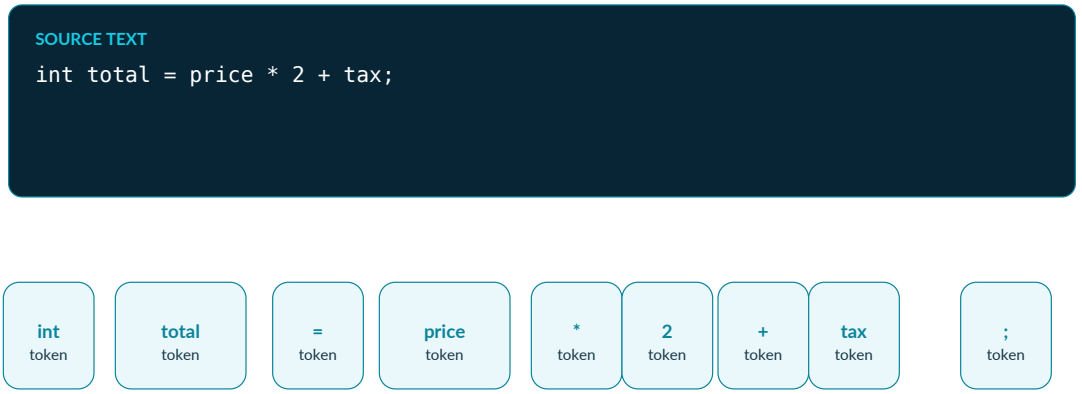
Lexical Analysis: Turning Characters into Tokens

The first compiler question is not what the program means. It is: what language elements are present in this character stream?

Lexical analysis groups characters into tokens: identifiers, keywords, literals, operators, punctuation, and related terminal elements. Roslyn exposes this lexical structure directly through its syntax model. Microsoft describes the parse phase as the stage that tokenizes and parses source text according to the language grammar.

Whitespace and comments matter for full-fidelity source tooling but usually do not carry ordinary executable semantics. Roslyn preserves them as trivia, which is why formatting and refactoring tools can round-trip source without throwing away comments.

FIGURE 27.3 - CHARACTER STREAM TO LEXICAL UNITS



COMPILER PRINCIPLE

The lexer recognizes legal token boundaries without yet deciding which declaration an identifier denotes. The token `price` is lexically an identifier; only later semantic binding can decide which variable, field, property, or other symbol that identifier means.

CHAPTER 27

One Line of C# as a Token Sequence

Compiler reasoning improves when the source text is stripped down to terminals before structure is discussed.

Consider the expression `total = price * 2 + tax;`. A simplified token stream is enough to show the lexical phase. The exact Roslyn syntax tree carries more detail, including trivia and source spans, but the conceptual point is stable.

FIGURE 27.4 - TOKEN CLASSIFICATION

TEXT	TOKEN CLASS	FIRST FACT KNOWN
total	IdentifierToken	a legal identifier-shaped token
=	EqualsToken	assignment punctuation/operator token
price	IdentifierToken	identifier-shaped token
*	AsteriskToken	multiplicative operator token
2	NumericLiteralToken	literal token with numeric value
+	PlusToken	additive operator token
tax	IdentifierToken	identifier-shaped token
;	SemicolonToken	statement terminator token

WHAT IS STILL UNKNOWN

Tokenization has not proved that `total` is assignable, that `price` and `tax` are numeric, or which overloaded operator should be used. Those are semantic questions, not lexical questions.

CHAPTER 27

Tokens, Trivia, and Lexical Failure Are Different Categories

A full-fidelity compiler front end preserves more than the parts that eventually become machine operations.

Roslyn syntax trees preserve nodes, tokens, and trivia. Tokens are terminals in the grammar. Trivia includes whitespace, comments, and preprocessor-related material that must survive source transformations. This distinction is invaluable in an IDE even though the JIT will never see the comments.

Lexical failure can arise before grammar is meaningfully considered: for example, an unterminated string literal or malformed numeric literal can prevent a clean token from being formed. Modern parsers still attempt recovery so the editor can continue providing a syntax tree and diagnostics.

FIGURE 27.5 - FULL-FIDELITY SOURCE STRUCTURE

BOUNDARY TO JIT

RyuJIT receives emitted CIL and metadata, not Roslyn syntax trivia. Comments, formatting, and C# lexical spelling are gone from the ordinary JIT input contract.

CHAPTER 27

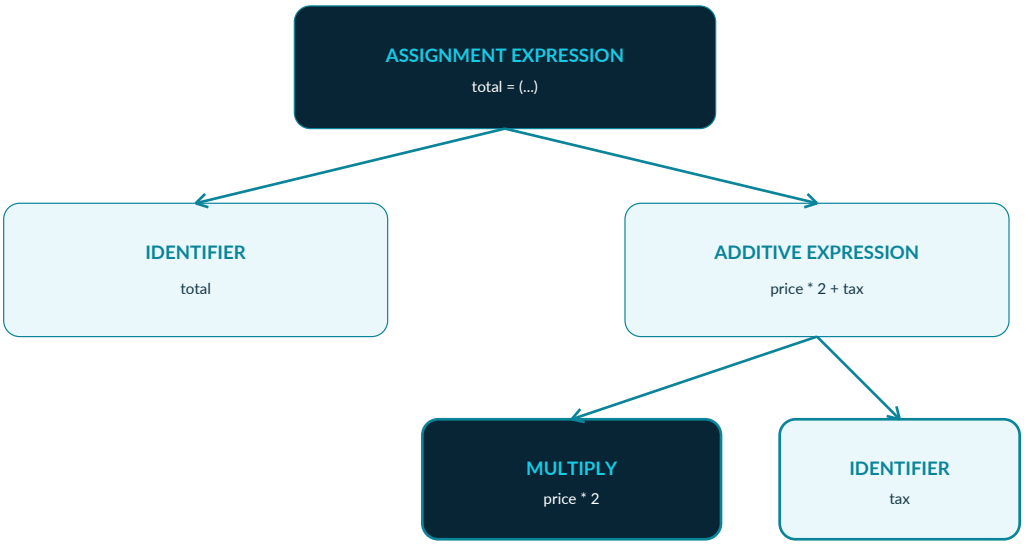
Syntactic Analysis: Does the Token Sequence Fit the Grammar?

Parsing groups tokens into grammatical constructs such as declarations, statements, and expressions.

After tokenization, the parser asks whether the sequence can be organized according to C# grammar. Syntax answers questions like: Is this an assignment expression? Which operands belong to the multiplication? Where does the statement end?

A parser therefore introduces hierarchy. The flat token stream becomes a tree whose parent-child relationships encode grammatical structure. Roslyn exposes this directly as a syntax tree.

FIGURE 27.6 - TOKENS BECOME A TREE



SYNTAX IS STRUCTURE, NOT MEANING

A syntactically valid expression can still be semantically invalid. For example, the tree can be perfectly legal even if `price` names a type for which multiplication by an integer is not defined.

CHAPTER 27

Grammar Encodes Precedence and Associativity

The parse tree must explain why price * 2 + tax is not grouped as price * (2 + tax).

Programming-language grammars are designed so that syntactic structure reflects precedence and associativity. In a simplified expression grammar, multiplicative expressions sit below additive expressions, giving multiplication tighter binding than addition.

This is compiler theory, not a JIT-specific implementation detail. By the time RyuJIT sees the method, the C# compiler has already resolved this syntax and emitted CIL in an order that carries the chosen semantics.

SIMPLIFIED GRAMMAR SHAPE

```
additive-expression:
    multiplicative-expression
    additive-expression  +  multiplicative-expression

multiplicative-expression:
    primary-expression
    multiplicative-expression  *  primary-expression
```

RESULT FOR THE EXAMPLE

The parse groups `price * 2` first, then adds `tax`. The emitted CIL therefore reflects an already-decided source-language structure. RyuJIT does not re-parse C# precedence rules.

CHAPTER 27

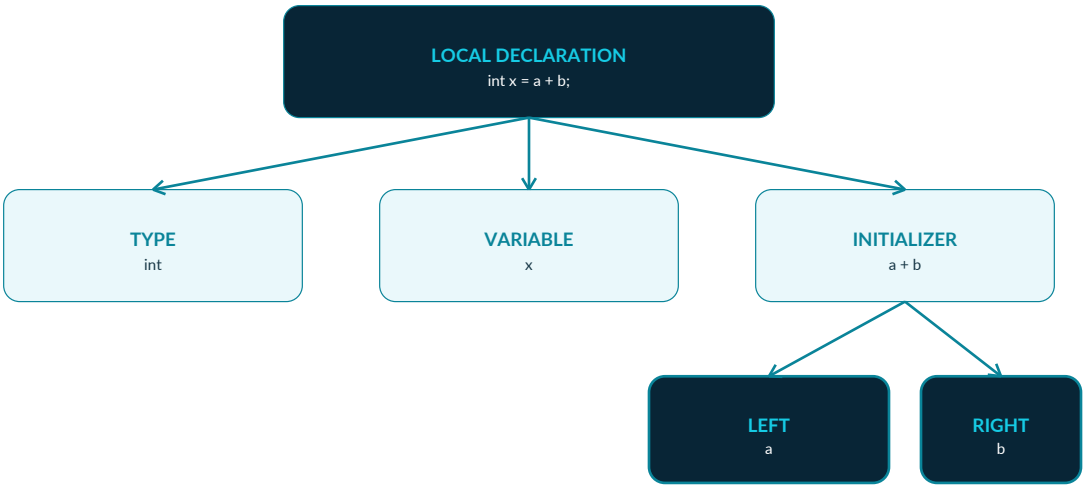
The Syntax Tree Is the Front End's Structural Record

Roslyn syntax trees retain every lexical token and grammatical construct in a form tools can inspect.

Microsoft describes syntax trees as the primary structure used for compilation, analysis, binding, refactoring, IDE features, and code generation. Nodes represent nonterminal constructs; tokens are terminals; trivia preserves source material around them.

The tree is immutable and full fidelity. That design serves editor and tooling needs as much as batch compilation. It also shows why a syntax tree is not an execution tree: it mirrors source grammar, not CPU operations.

FIGURE 27.7 - A SOURCE-ORIENTED TREE



LATER TRANSFORMATION

Lowering can replace one elegant source construct with a very different lower-level shape. Therefore, a source syntax tree and a JIT IR tree should never be assumed to have one-to-one nodes.

CHAPTER 27

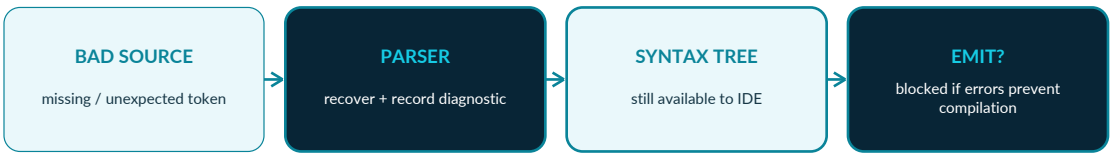
A Real Parser Must Recover from Incomplete or Wrong Source

Compilers used inside IDEs cannot simply stop at the first missing semicolon.

Roslyn represents malformed source with diagnostics while still producing a syntax tree. The parser can insert missing tokens or attach skipped tokens so the rest of the file remains analyzable. This is a classic practical compiler concern: error recovery must preserve enough structure to continue.

Runtime JIT compilation faces a different trust boundary. It is not maintaining an interactive source editor. It receives CIL and metadata under CLI/runtime rules; malformed or unsupported method input fails at that later boundary rather than being repaired as C# source.

FIGURE 27.8 - ERROR RECOVERY BELONGS TO THE SOURCE FRONT END



DO NOT TRANSFER BEHAVIORS ACROSS LAYERS

Roslyn recovery exists to keep source analysis productive. It should not be described as a JIT phase. The JIT has its own validation and failure paths over IL/runtime information.

CHAPTER 27

Declaration Analysis Builds the Symbol World

Syntax says a class declaration exists. Declaration analysis creates the named program entity it represents.

After parsing, the source compiler analyzes declarations and imported metadata to form symbols. Namespaces, types, methods, fields, properties, parameters, locals, and other language concepts participate in this semantic world.

A hierarchical symbol table allows later binding to answer questions such as which `Calculate` method an invocation refers to, or whether `Widget` denotes the source type or an imported metadata type.

FIGURE 27.9 - SYNTAX DECLARATIONS BECOME SYMBOLS

SOURCE AND METADATA MEET HERE

Roslyn symbols unify declarations from source with entities imported from assembly metadata. That is how a C# method can bind a call to a framework method whose source is not part of the current compilation.

CHAPTER 27

Semantic Binding Connects Identifiers to Meaning

The parser can recognize the word Count. The binder must decide what Count actually denotes here.

Binding matches identifiers and expressions to symbols under the language rules. Scope, accessibility, generic substitutions, overload sets, extension methods, conversions, and contextual typing all influence the result.

This is the point where the compiler moves from "this token has identifier syntax" to "this identifier denotes this specific property/method/local/type." Microsoft exposes the result through Roslyn's semantic model.

FIGURE 27.10 - ONE SPELLING, DIFFERENT POSSIBLE SYMBOLS

WHY THE JIT DOES NOT NEED C# BINDING

The emitted CIL already contains metadata tokens and signatures that identify runtime entities according to CLI rules. RyuJIT resolves those runtime references; it does not repeat C# overload resolution or namespace lookup.

CHAPTER 27

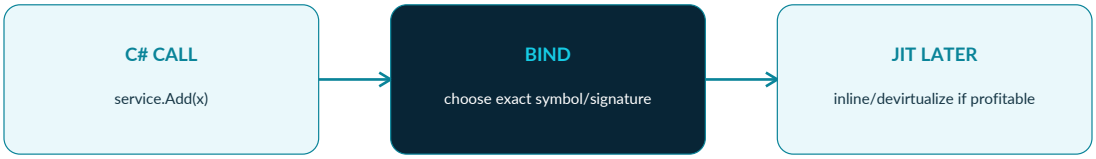
Type Checking and Overload Resolution Are Source-Language Work

Meaning is constrained before the program is emitted as CIL.

Semantic analysis verifies that operators, conversions, assignments, returns, pattern operations, and invocations obey the C# type system. Overload resolution chooses one applicable callable from a candidate set under C# rules.

By emit time, an invocation is no longer "some method named Add." The emitted method reference/signature identifies the selected target. The JIT may later optimize the call, inline it, or devirtualize it when safe, but that is a different question from C# overload selection.

FIGURE 27.11 - SELECT TARGET FIRST, OPTIMIZE CALL LATER



TWO DIFFERENT DECISIONS

Binding decision: what does the source program mean? JIT optimization decision: how should already-defined semantics be implemented efficiently on this runtime and target?

CHAPTER 27

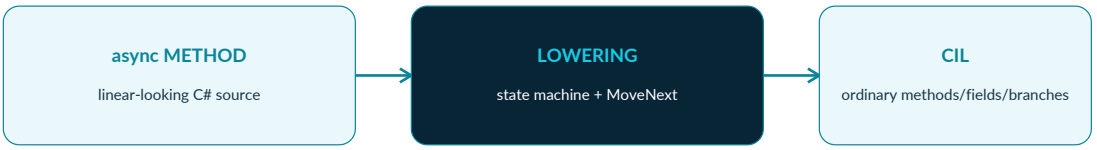
Lowering Rewrites High-Level C# into Simpler Semantics

Many C# constructs disappear before the JIT ever sees the method.

Chapter 23 showed that `foreach`, `using`, `lock`, async methods, iterators, and closures can be transformed by the source compiler. This transformation is called lowering: replace high-level language sugar with simpler operations that preserve behavior.

Compiler theory often uses multiple intermediate representations. Roslyn's exact internal lowering machinery is an implementation detail, but the architectural fact is visible in emitted IL: the JIT receives state machines, generated fields, helper calls, branches, and other lowered forms rather than the original C# syntax.

FIGURE 27.12 - SOURCE CONSTRUCT TO LOWERED SHAPE



CONSEQUENCE FOR JIT READING

If native code looks unlike the original C#, the first divergence may have happened before JIT compilation. Separate source lowering from JIT optimization when explaining a surprising result.

CHAPTER 27

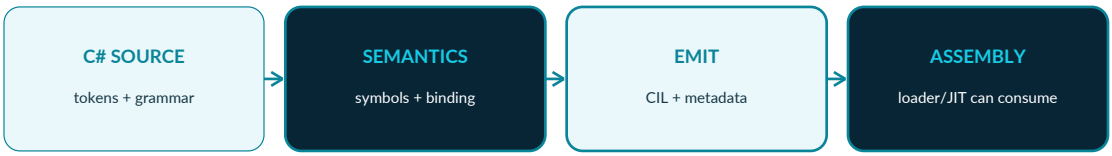
Emit Closes the Source-Compiler Act

The product is an assembly: CIL, metadata, signatures, resources, and related runtime information.

Microsoft's compiler-pipeline description places emit after parsing, declaration analysis, and binding. The emit API produces IL bytecodes and assembly metadata. At this point the source-language compiler has completed the decisions required to define the program's managed semantics.

The assembly is therefore not a half-parsed C# file. It is a different language artifact whose instruction set and metadata model are specified by the CLI.

FIGURE 27.13 - ROSLYN OUTPUT BECOMES RUNTIME INPUT



LINE IN THE SAND

Everything from lexical analysis through C# binding belongs on the left of this line. RyuJIT starts on the right, after loading has exposed a method body expressed in CIL plus runtime metadata/context.

CHAPTER 27

The JIT Does Not Lex or Parse C#

This sentence is the guardrail for the rest of the chapter.

RyuJIT is still a compiler and still has a front end, but its front-end language is CIL, not C#. It scans CIL control flow, imports stack-machine instructions, resolves runtime information through the JIT/runtime interface, and constructs its own IR.

So it is correct to compare Roslyn lex/parse -> syntax/semantic model with RyuJIT import -> IR/control-flow model as analogous compiler stages. It is incorrect to say that RyuJIT performs C# lexical or syntactic analysis.

FIGURE 27.14 - ANALOGOUS ROLES, DIFFERENT LANGUAGES

COMPILER QUESTION	ROSLYN	RYUJIT
What are the units?	tokens / grammar elements	CIL instructions / operands / runtime references
What is the structure?	syntax tree	basic blocks + JIT IR
What does a name/reference mean?	C# symbol binding	runtime metadata/type/method resolution
What is emitted?	CIL + metadata	native code + runtime tables

PEDAGOGICAL RULE

We study both because the JIT becomes far easier to understand once the reader recognizes the recurring compiler pattern: input language -> structural IR -> analysis -> transformation -> target code.

CHAPTER 27

What Exactly Enters a JIT Compilation?

A method body alone is not enough; the compiler also needs runtime context and target information.

For a method selected for JIT compilation, CoreCLR provides RyuJIT with information describing the method, its CIL bytes, signature, locals, exception-handling regions, generic context, and access to runtime services through the JIT/runtime interface.

The target architecture and ABI matter too. The same CIL method can require different register assignments, calling conventions, instruction selections, and prolog/epilog shapes on x64 versus Arm64.

FIGURE 27.15 - THE JIT INPUT CONTRACT

WHY RUNTIME COMPILATION CAN KNOW MORE

Unlike the source compiler, the JIT runs inside a concrete runtime on a concrete target. It can ask questions about actual runtime type layout, helper addresses, current generic instantiation, and target-specific capabilities.

CHAPTER 27

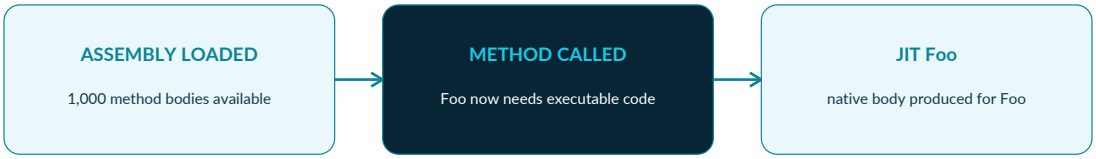
JIT Compilation Is Normally Method-Oriented and Demand-Driven

Loading an assembly creates availability. Reaching executable code creates the need for native code.

Ordinary JIT compilation is centered on individual methods. A method may remain only as CIL for the lifetime of a process if no execution path needs it. When a callable entry is required and no suitable native body is already available, the runtime invokes the JIT for that method.

The exact entry-stub and code-versioning mechanics can vary with runtime features such as tiered compilation. This chapter keeps the invariant idea: native method bodies are materialized under runtime control, not by compiling the entire assembly at load time.

FIGURE 27.16 - LOAD != JIT EVERY METHOD



CHAPTER 30 WILL REFINE THIS

Tiered compilation, code versions, and PGO can cause a method to be compiled more than once. For now, learn the compilation pipeline itself; Chapter 30 will add the policy that decides which quality tier to compile.

CHAPTER 27

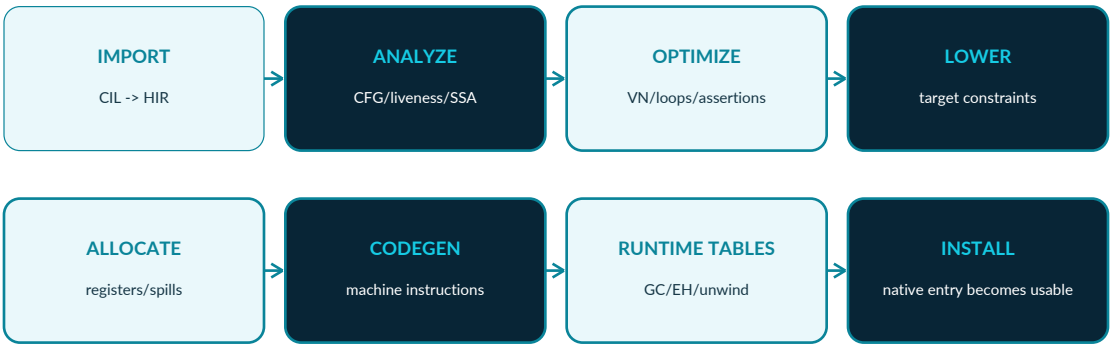
RyuJIT Is a Pipeline of Many Phases

The current dotnet/runtime documentation exposes a long phase list rather than one giant "optimize" function.

The RyuJIT overview identifies major phases including profile incorporation, importation, inlining, object stack allocation, morphing/normalization, loop discovery and transformations, SSA/value-numbering-based optimization, rationalization, lowering, register allocation, and code generation.

The exact phase list evolves. The stable lesson is architectural: each phase consumes an IR with certain invariants, establishes new facts or transforms the representation, and hands a more constrained form to later phases.

FIGURE 27.17 - A COMPRESSED RYUJIT PIPELINE



SOURCE ANCHOR

This page follows the current dotnet/runtime RyuJIT Overview and Tutorial. Individual phase names are implementation details; the book will label them as CoreCLR/RyuJIT specifics rather than universal JIT theory.

CHAPTER 27

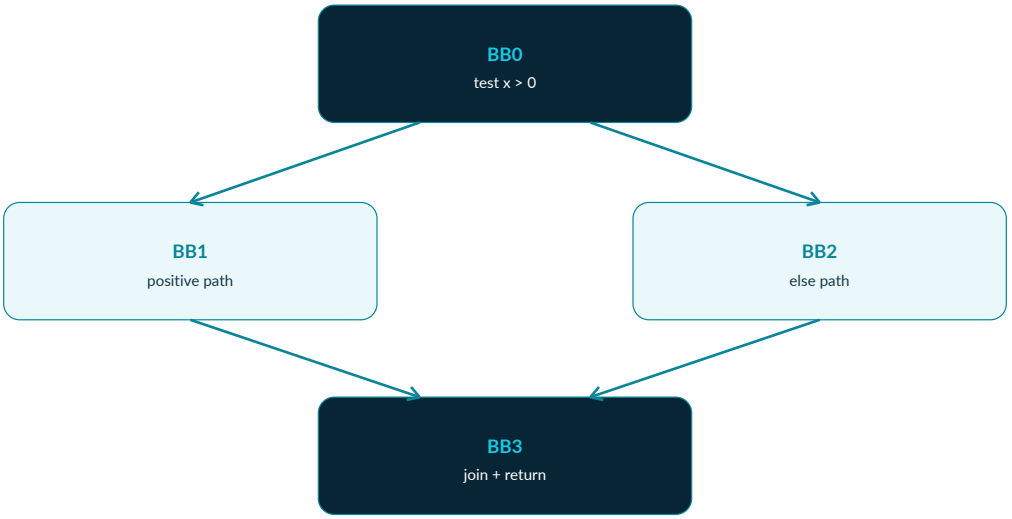
First Structural Job: Discover Basic Blocks and Control Flow

Before deep optimization, the JIT needs a graph of where execution can go.

A basic block is a straight-line region with one entry and control transfer at its end. Branch targets, fall-through edges, returns, and exception regions divide IL into these units. The blocks become nodes in a control-flow graph (CFG).

This is the JIT analogue of moving from a flat instruction stream to a structural program representation. It is not parsing C# grammar; it is reconstructing executable control flow from CIL branch semantics.

FIGURE 27.18 - IL BRANCHES BECOME A CONTROL-FLOW GRAPH



WHY CFG COMES EARLY

Dominance, liveness, loop discovery, SSA construction, unreachable-code reasoning, and many optimizations depend on a correct control-flow graph. Structure enables analysis.

CHAPTER 27

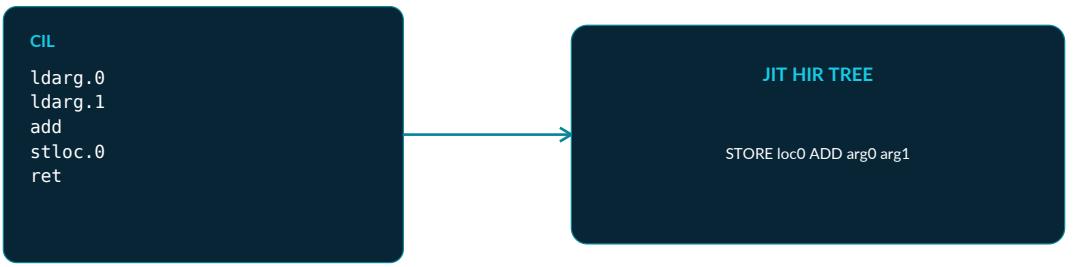
Importation Turns Stack-Based CIL into JIT IR

RyuJIT consumes the CIL evaluation-stack semantics and constructs compiler-owned nodes and statements.

The RyuJIT tutorial describes the importer as the phase that initializes local-variable information, scans MSIL to form basic blocks, and creates IR from the MSIL. The overview describes imported operations as `GenTree` nodes linked into statements and basic blocks.

This is a true compiler front-end transformation: the input is one IR/language (stack-based CIL); the output is another IR designed for analysis and transformation inside RyuJIT.

FIGURE 27.19 - CIL STACK OPERATIONS TO EXPRESSION IR



NO SOURCE SYNTAX NEEDED

The importer already knows operand stack effects from the CIL instruction set. It can therefore recover data dependencies without reconstructing the original C# expression text.

CHAPTER 27

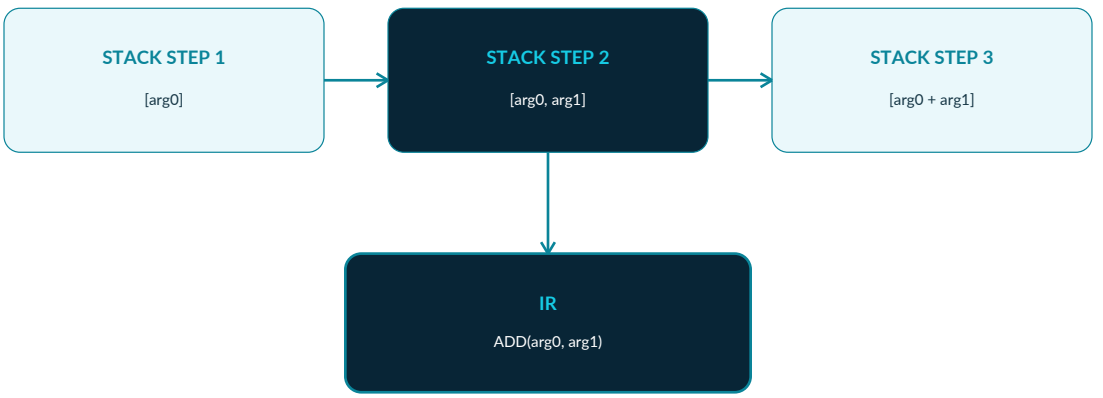
The Evaluation Stack Becomes Explicit Data Dependencies

Chapter 25's logical stack is consumed as the importer builds a graph/tree of operations.

For `ldarg.0; ldarg.1; add`, the evaluation stack temporarily holds the two argument values. Importation can represent the result as an ADD node whose children are the two argument-value nodes. The temporary stack discipline has been converted into explicit producer-consumer relationships.

This is one reason compiler IRs are powerful: later passes do not need to keep mentally simulating the original stack at every instruction. The dependencies are present in the representation.

FIGURE 27.20 - STACK HISTORY COMPRESSED INTO A DATAFLOW TREE



COMPILER PRINCIPLE

IR design chooses what facts become explicit. CIL makes evaluation order and stack transitions explicit; RyuJIT IR makes expression/data/control relationships easier for optimization passes to inspect.

CHAPTER 27

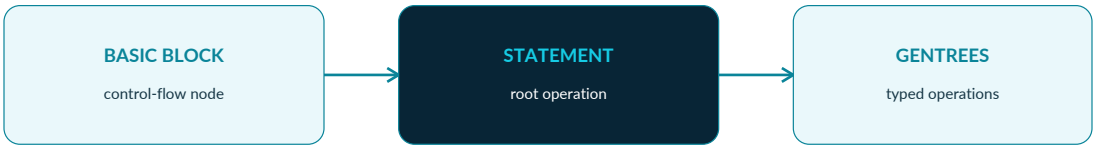
RyuJIT HIR Uses Trees Inside Basic Blocks

The JIT's high-level IR is not C# syntax and not raw assembly. It is a compiler-owned operation graph/tree.

The RyuJIT overview describes `GenTree` nodes representing operations and statements within `BasicBlock` objects. Local descriptors represent arguments, user locals, and JIT-created temporaries. The IR carries types, flags, and later analysis facts such as value numbers and register assignments.

At this stage, operations are still relatively target-independent. That allows a large middle-end optimization body to be shared across architectures.

FIGURE 27.21 - ONE BASIC BLOCK AS STATEMENTS OF OPERATION TREES



VOCABULARY DISCIPLINE

A JIT IR tree is not the Roslyn syntax tree. Both are trees because trees are useful, but they encode different languages, invariants, and optimization needs.

CHAPTER 27

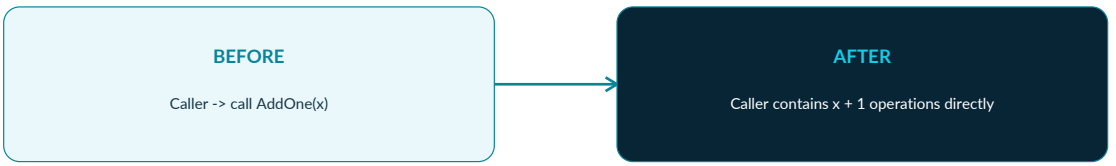
Inlining Moves a Callee's IR into the Caller

A method call can become an optimization boundary - or disappear as a call when inlining is profitable and legal.

During importation/inlining analysis, RyuJIT identifies candidate call sites and estimates whether substituting the callee body into the caller is worthwhile. Inlining can expose constants, eliminate call overhead, enable propagation across the former boundary, and unlock other optimizations.

But inlining increases generated code size and JIT work. The decision is therefore heuristic and runtime-version dependent. It is not a semantic guarantee and should never be assumed from source-level attributes or method size alone.

FIGURE 27.22 - CALL BOUNDARY BEFORE AND AFTER INLINING



WHY IT MATTERS FOR NATIVE INSPECTION

A source method can appear absent as a distinct native call because its body was inlined. Conversely, a tiny method can remain a call if policy, code shape, tier, generics, or other constraints make inlining undesirable.

CHAPTER 27

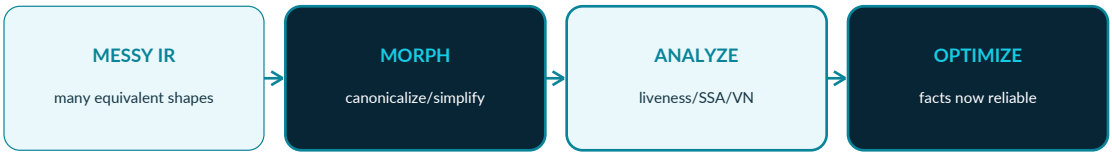
Morphing Normalizes and Simplifies the IR

Optimization becomes safer when later phases can rely on stronger representation invariants.

RyuJIT uses morphing passes to perform localized transformations and normalization. The overview calls out local and global morph phases, including simplification of local accesses, identification of address-exposed locals, and mandatory normalization work.

This illustrates a classic middle-end principle: not every transformation is an "optimization" in the narrow sense. Some passes reshape the program into forms that later analyses and back-end phases can handle consistently.

FIGURE 27.23 - NORMALIZATION ENABLES LATER ANALYSIS



COMPILER PRINCIPLE

A pass can be valuable even when it does not immediately make the program faster. Establishing canonical forms reduces the number of cases every later pass must reason about.

CHAPTER 27

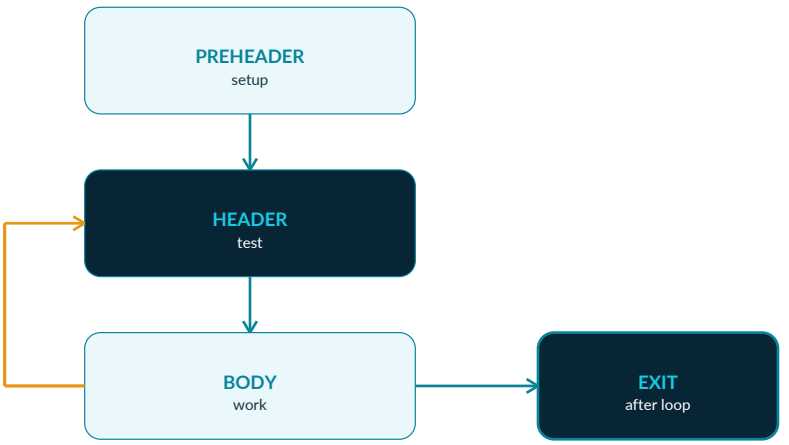
The Flow Graph Reveals Loops as Optimization Regions

A backward branch is only the beginning; the JIT needs loop structure suitable for transformation.

RyuJIT finds natural loops and canonicalizes them. The current phase list includes loop discovery plus transformations such as inversion, cloning, unrolling, and induction-variable-related optimization, subject to profitability and legality.

Loop structure matters because repeated execution magnifies both cost and opportunity. Moving one invariant computation out of a loop can remove work from every iteration.

FIGURE 27.24 - LOOP STRUCTURE IN THE CFG



HOTNESS IS POLICY, NOT SYNTAX

A source `for` loop does not guarantee unrolling or hoisting. The JIT must prove transformations safe and decide they are profitable under its current compilation policy.

CHAPTER 27

Liveness Analysis Asks Where Each Value Still Matters

Register allocation and many transformations depend on knowing whether a variable's current value may be used later.

For each block, liveness analysis computes values that are live on entry and exit, along with definitions and upward-exposed uses. RyuJIT tracks selected locals for this purpose; the overview notes that not every local must participate equally in global liveness/register-allocation treatment.

The core dataflow equation is a compiler classic: a value is live before a point if some future path can use that value before it is redefined.

FIGURE 27.25 - LIVE RANGE AS A DATAFLOW FACT

WHY THE BACK END CARES

If two values are simultaneously live, they compete for registers. If a value is dead, its register can be reused and some stores/computations may become removable.

CHAPTER 27

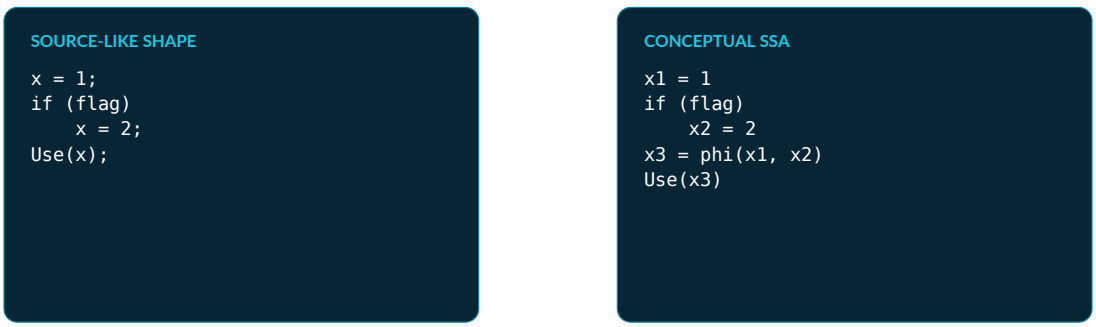
Static Single Assignment Gives Definitions Unique Names

SSA turns "which assignment produced this value?" into a much easier question.

RyuJIT constructs SSA form for tracked locals in a traditional manner. Each assignment produces a distinct SSA name, and uses can be connected to the defining value. The overview emphasizes that SSA is primarily used to represent use-def chains in the JIT.

For compiler reasoning, this is transformative. Instead of one variable x changing value many times, analyses can reason about x1, x2, x3 as distinct values.

FIGURE 27.26 - MUTABLE NAME TO SSA VALUES



NOT A SOURCE REWRITE

The phi notation is a compiler concept used to explain merging definitions at control-flow joins. It is not CIL syntax and not necessarily a literal node that survives unchanged to final code generation.

CHAPTER 27

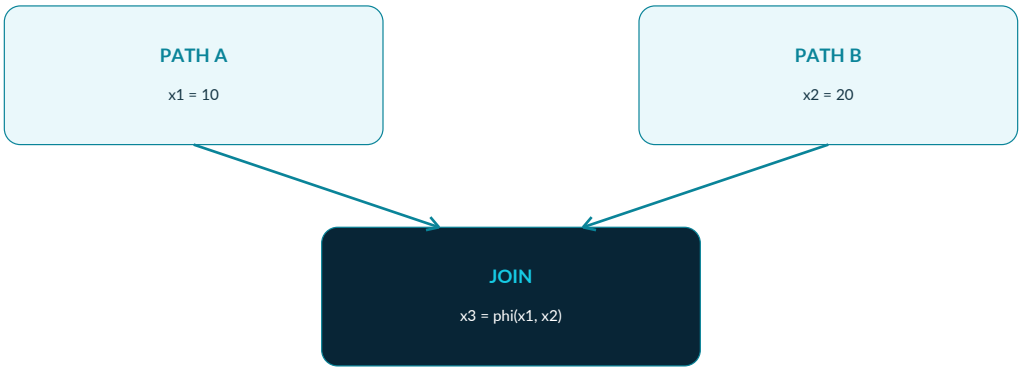
Control-Flow Joins Need a Value-Merge Story

Two predecessor paths can bring different definitions of the same logical local to one block.

In SSA reasoning, a join point selects a value according to the predecessor edge that executed. This is commonly represented with a phi function. The function does not execute as an ordinary runtime call; it is a representation device for dataflow.

Once definitions are explicit, value numbering, constant/copy propagation, assertion propagation, and other optimizations can reason more precisely about which value reaches which use.

FIGURE 27.27 - PHI AT A JOIN



COMPILER PRINCIPLE

SSA is valuable because it converts implicit "latest assignment along this path" reasoning into explicit value identities. Optimizers operate on values, not just source variable names.

CHAPTER 27

Value Numbering Recognizes Equivalent Computations

If two expressions are proven to compute the same value, the optimizer can reason about them as one equivalence class.

RyuJIT value numbering uses SSA for locals and also assigns value numbers to expression trees. The JIT can symbolically evaluate operations and record when different nodes represent the same value under the current analysis assumptions.

This fact becomes the foundation for optimizations such as common-subexpression elimination, assertion propagation, copy propagation, and range-check elimination.

FIGURE 27.28 - SAME VALUE, TWO COMPUTATIONS

SAFETY FIRST

Equivalent-looking source text is not enough. Aliasing, exceptions, volatile behavior, memory effects, overflow modes, and other semantics can prevent reuse. Value numbering encodes proven equivalence under the JIT's model, not textual similarity.

CHAPTER 27

Optimization Is a Family of Proven Transformations

There is no single "optimization step." Many passes exploit different facts and preserve the same observable semantics.

The RyuJIT phase list includes numerous middle-end transformations. The names matter less than the proof obligation behind them: every transformation must preserve required program behavior while reducing work, improving code shape, or enabling later transformations.

OPTIMIZATION	IDEA	EXAMPLE EFFECT
Copy propagation	replace a copy with the original value when safe	y=x; Use(y) -> Use(x)
CSE	compute an equivalent expression once and reuse it	reuse repeated a+b
Assertion propagation	use known facts such as non-null or relational constraints	remove impossible branches/checks
Range analysis	prove index bounds from facts	remove redundant bounds checks
Loop invariant hoisting	move repeated invariant work outside loop	compute constant expression once
Dead store elimination	remove a store whose value is never observed	delete overwritten local store

OBSERVATION RULE

A missing instruction in native code is not automatically a bug in your disassembler. It may be the evidence that the compiler proved the work unnecessary.

CHAPTER 27

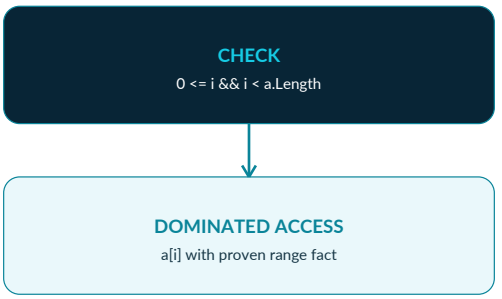
Assertions Can Turn Runtime Checks into Compile-Time Facts

The optimizer tracks facts like non-nullness, equalities, ranges, and relationships between values.

Assertion propagation uses analysis facts to transform later operations. Range analysis can combine such facts with array-index reasoning to eliminate checks when safety has already been proven for that path.

The key compiler idea is that optimization is often proof reuse. If an earlier condition established `0 <= i && i < a.Length`, a later access on the same proven path may not need to rediscover the same fact dynamically.

FIGURE 27.29 - CONDITION ESTABLISHES A FACT FOR THE DOMINATED REGION



NOT A PROMISE OF CHECK REMOVAL

Whether a bounds check is actually removed depends on exact IR shape and the JIT version/tier. The compiler principle is the important part: optimization requires a proof strong enough to preserve safety.

CHAPTER 27

Loop Optimizations Trade Code Size for Repeated-Work Savings

A transformation that is profitable once may be expensive when it duplicates too much machine code.

RyuJIT includes loop transformations such as inversion, cloning, unrolling, invariant-code hoisting, and induction-variable optimization. Each changes the shape of the loop to expose or remove repeated work.

Unrolling is a classic example of a trade-off: fewer branch/control operations and more optimization opportunities versus larger generated code. The JIT must make this decision under a compilation-time budget, not with unlimited whole-program analysis.

FIGURE 27.30 - UNROLLING AS A CODE-SIZE / CONTROL-OVERHEAD TRADE

COMPACT LOOP

```
for (i=0; i<4; i++)
    sum += a[i];
```

CONCEPTUAL FULL UNROLL

```
sum += a[0];
sum += a[1];
sum += a[2];
sum += a[3];
```

COMPILER ENGINEERING

A JIT optimizes while the application is trying to run. It therefore balances execution quality against compilation latency and code-cache footprint. Chapter 30 will connect that pressure to tiered compilation.

CHAPTER 27

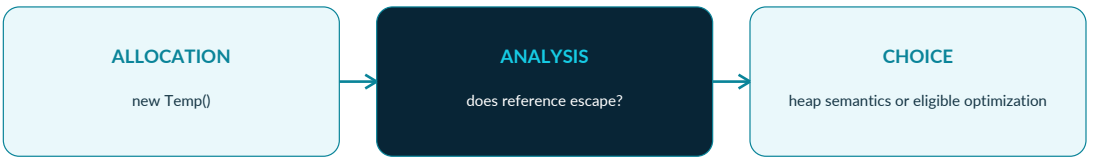
Escape Analysis Can Change Where Some Objects Are Allocated

The JIT may prove that an allocation does not need ordinary heap lifetime in specific supported cases.

The RyuJIT phase list includes Object Stack Allocation, where escape analysis can allow eligible objects to be allocated on the stack when the compiler proves they do not escape in ways that require normal heap identity/lifetime. This is a RyuJIT implementation feature, not a general C# semantic rule.

The broader compiler principle is escape analysis: determine whether a value can be observed outside a region. Stronger non-escape proofs can unlock allocation and scalar-replacement transformations.

FIGURE 27.31 - ESCAPE QUESTION



DO NOT PROGRAM AGAINST THE OPTIMIZATION

Source correctness must never depend on stack allocation happening. The JIT is free to optimize or not optimize as long as observable language/runtime semantics remain valid.

CHAPTER 27

Rationalization Begins the Transition toward the Back End

The representation becomes less tree-context-dependent and more suitable for target-oriented lowering.

The RyuJIT tutorial describes Rationalization as a transitional pass that removes some high-level tree context before back-end processing. The representation then moves toward LIR, where execution order is explicit in a linear node sequence within blocks.

This illustrates another compiler pattern: the IR that is ideal for high-level optimization is not necessarily the IR that is ideal for register constraints and final machine emission.

FIGURE 27.32 - HIR TO LIR

MIDDLE END MEETS BACK END

Until this region, much reasoning is relatively target-independent. From lowering onward, target instruction constraints, ABI rules, registers, and machine-specific details become increasingly dominant.

CHAPTER 27

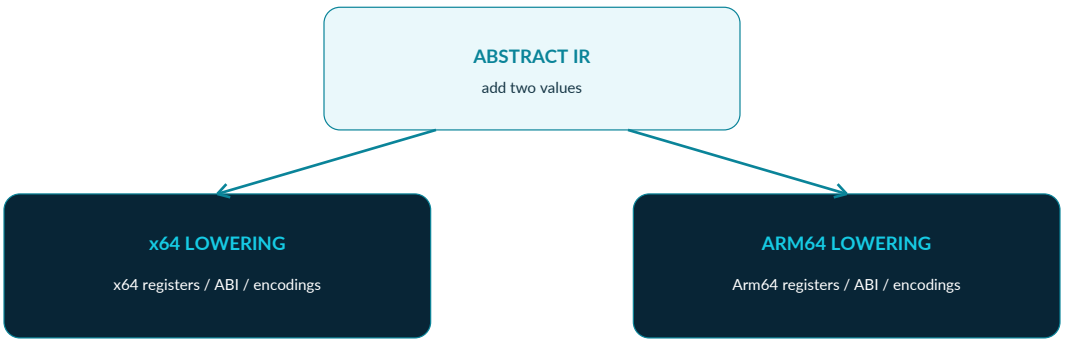
Lowering Exposes Target Constraints

A generic operation must be reshaped into something the current architecture and ABI can actually implement.

The RyuJIT tutorial says lowering fully exposes control flow and register requirements. This is where abstract operations are prepared for target instruction selection and register allocation. Some operations may require helpers; others may map naturally to one or a few target instructions.

Lowering is therefore not merely "translate ADD to add instruction." The compiler must account for operand forms, destructive/non-destructive instruction rules, calling conventions, immediate encodings, address modes, SIMD capabilities, and runtime helper contracts.

FIGURE 27.33 - ONE ABSTRACT OPERATION, DIFFERENT TARGET SHAPES



WHY THE SAME IL PRODUCES DIFFERENT NATIVE CODE

Portability ends at semantics, not instruction bytes. The JIT preserves managed behavior while choosing a code shape appropriate to the current target architecture and runtime contract.

CHAPTER 27

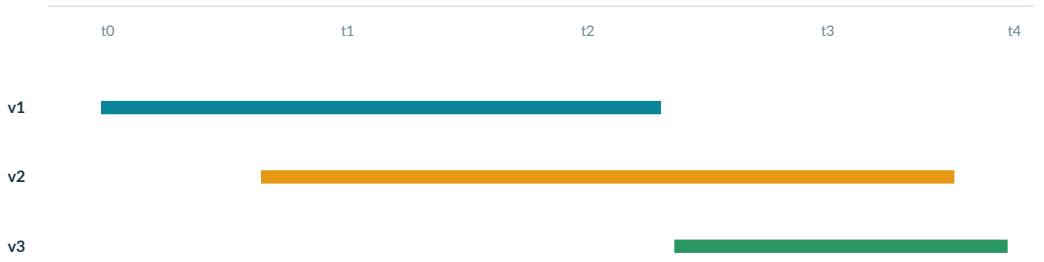
Register Allocation Maps Live Values onto a Scarce Hardware Resource

The IR can name many values; the CPU exposes only a limited set of usable registers.

RyuJIT uses a linear-scan register allocator with splitting, according to the current tutorial. The allocator walks live ranges, assigns physical registers where possible, and handles conflicts by splitting ranges or placing values in stack locations.

This is where liveness analysis becomes concrete. Two values whose live ranges overlap cannot occupy the same register at the same time. A dead value releases its register for reuse.

FIGURE 27.34 - LIVE RANGES COMPETE FOR REGISTERS



ALLOCATOR OUTPUT

After allocation, the IR is annotated with physical register choices and spill locations. Code generation no longer talks only about abstract locals; it knows where values must live at each point.

CHAPTER 27

When Registers Run Out, Values Spill to the Stack Frame

Spilling is not a compiler failure. It is the ordinary consequence of register pressure.

If too many values are simultaneously live, some cannot remain in registers. The allocator can spill values to stack locations and later reload them. These memory operations add cost, so earlier optimizations and good live-range management can indirectly reduce spills.

The resulting native stack frame may also hold address-exposed locals, saved registers, outgoing call arguments under ABI rules, and other runtime/compiler state. This is a native-code concern, distinct from the CIL evaluation stack studied in Chapter 25.

FIGURE 27.35 - REGISTER PRESSURE AND SPILL

FOUR DIFFERENT "STACKS" NOW EXIST IN THE BOOK

Do not collapse the CIL evaluation stack, the native thread stack, a method stack frame, and a conceptual compiler work stack. They answer different questions.