R# shorthand

Parts 1–16 · quick reference

The same language map, compressed to the decisions that matter.

Shorthand foundations edition · 11 September 2026

Use the map, then open the full part

READING RULE

Each part keeps the main manual’s sequence and slide topics. Use the full manual for explanation and examples.

What R\# is

Intent first; Rust later.

Make choices visible

SHAPE

`struct`

OUTCOME

`Option` / `Result`

ACCESS

`var mut` / `borrowed<T>` / `shared<T>`

Ask two questions

MODEL

What value or behavior?

BOUNDARY

Who owns, changes, or observes it?

Source is not generated Rust

```
borrowed<string> Choose(borrowed<string> first)
{
    return first;
}
```

RULE

Write R# semantics; do not backport Rust lifetimes.

Intent becomes checked Rust

```
.rsharp → solver → .partial → .rs → Cargo
```

Read the right layer

- ☐ Read the mapped R# span.
- ☐ Fix authored source.
- ☐ Never patch `.feo/` or generated Rust.

Run an existing R\# program

Find, solve, check, run.

Run the solution

```
feo run app.feo-solution
```

TARGET

Start from `.feo-solution`, not a loose source file.

Choose the solver

CODEX

Default backend.

RANDOM

`--solver random`; local reproduction.

CARGO

Final acceptance check.

Match command to question

SOLVE

Populate decisions.

CHECK

Compile; do not run.

RUN

Compile and start.

Keep the path

```
feo solve --solver random app.feo-solution
feo check --solver random app.feo-solution
feo run   --solver random app.feo-solution
```

Clean, then solve

TEST

`feo test` runs Cargo tests.

CLEAN

`feo clean` removes generated state.

Solutions, projects, source trees

Know the boundary before editing.

Solution ≠ project

SOLUTION

Groups projects.

PROJECT

One crate-shaped unit.

STARTUP

Executable selected for `run`.

Read top down

```
app/
├── app.feo-solution
└── src/Program.rsharp
```

List projects

```
projects = ["app/app.feo-project"]
startup = "app"
```

Describe one crate

```
name = "app"
kind = "bin"
source = "src"
```

Resolve from the owner

SOLUTION

`projects` is relative to `.feo-solution`.

PROJECT

`source` and `references` are relative to `.feo-project`.

Keep source below `source`

NESTING

Folders improve source maps; names still share one crate root.

Folders are not modules

NAMING

Keep declarations unique across a project.

Treat `.feo/` as evidence

```
.feo/generated/.../*.partial
.feo/generated/.../*.rs
.feo/solver/.../types.toml
```

Read an R\# source file

Names, declarations, boundaries.

Source is a declaration list

```
import domain.Core;
struct Program { }
```

Top-level forms

IMPORT

Name a dependency.

STRUCT

Define concrete state.

FUNCTION

Define behavior.

Type before name

```
public string Greeting(string name)
```

Semicolon ends a declaration

```
import domain.Core;
void Write(string message);
```

Braces contain bodies

```
struct Counter
{
    int value;
}
```

Comments disappear

```
// Explain the external constraint.
```

Comment the why

KEEP

Constraint, trade-off, or external contract.

Import the usable name

```
import logistics.Fleet;
using crossterm.terminal.Clear as Clear;
```

Manifest vs source

MANIFEST

References and Cargo dependencies.

SOURCE

R# imports.

Folders do not isolate names

CURRENTLY

Generated includes share one crate root.

Name each declaration once

CHECK

Unique top-level structs, interfaces, and functions.

Scan before editing

- ☐ Right file?
- ☐ Right project?
- ☐ Unique name?
- ☐ Correct boundary?

Your first R\# program

One binary, one entrypoint, one loop.

Start at static Main

```
struct Program
{
    public static void Main() { }
}
```

Main is static and parameterless

BINARY

The generated launcher calls a static method on a struct.

R\# Main is not Rust main

LOWERING

You write R#; the compiler creates the Rust wrapper.

Smallest complete program

```
struct Program
{
    public static void Main()
    {
        Console.WriteLine("Hello, R#");
    }
}
```

Read Main top to bottom

FLOW

Create → call → print.

Name intermediate values

```
var name = "Ada";
string greeting = "Hello";
```

Plain locals do not reassign

MUTABILITY

Use `var mut` only when reassignment is required.

Print with Console.WriteLine

```
Console.WriteLine(greeting);
```

Return one useful result

```
string Greeting(string name) { return "Hello, " + name; }
```

Check, then run

```
feo check --solver random app.feo-solution
feo run --solver random app.feo-solution
```

Check source and manifest

☐ Startup name?

☐ `kind = "bin"`?

☐ Generated state untouched?

Change one thing

LOOP

Solve or check after each source-level change.

Values and basic types

Name the value you mean.

Start with the value

```
var title = "R# training";
```

Use `string` for text

DEFAULT

Text is owned unless the signature says otherwise.

Whole numbers begin as `int`

```
var retries = 3;
var debt = -2;
```

Choose the numeric contract

SIGNED

`int` / `i64`

UNSIGNED

`uint` / `u64`

INDEX

`usize`

Use `bool` for one question

```
bool ready = true;
```

Type fractional boundaries

TYPES

`float` and `double` lower to `f64`.

Do not invent syntax

CURRENT

A type name does not guarantee decimal literal support.

Annotate when the contract matters

```
string greeting = "Hello";
```

Meaning before spelling

TEXT

```
string
```

COUNT

```
int
```

CONDITION

```
bool
```

Review the value first

- ☐ What does it represent?
- ☐ Who consumes it?
- ☐ Does the type expose that contract?

Bindings and mutability

Change must be visible.

Separate type from mutability

```
var title = "R#";
string name = "Ada";
var mut retries = 3;
```

Reassignment needs `mut`

```
var mut attempts = 3;
attempts = attempts - 1;
```

Fix mut in source

RULE

Cargo rejects reassignment to an immutable binding.

Keep updates local

```
var mut remaining = 3;
remaining = remaining - 1;
```

Ask two questions

TYPE

What is the value?

MUT

May this name change?

Fields belong to the struct

```
struct Counter
{
    int value;
}
```

Update through `this.field`

OBJECT STATE

Fields do not spell `mut`; the assignment shows change.

Local or field?

LOCAL

One invocation.

FIELD

Object lifetime.

MUT

Explicit reassignment.

Use `var`, not `let`

FORMS

`var name = value` or `Type name = value`.

Justify every update

☐ Is reassignment necessary?

☐ Is state owned by the object?

☐ Is the smallest scope used?

Functions and parameters

Read the signature first.

Modifiers → signature → body

PROMISE

The body must keep the signature’s contract.

Type before name

```
string Greeting(string name)
```

Call it by name

```
var message = Greeting("Ada");
```

Arguments fill local names

ORDER

Call arguments match parameter order.

Return data or effect

```
string Label(string name) { return name; }
void Print(string name) { Console.WriteLine(name); }
```

Put type-owned behavior on a type

```
struct Text
{
    public static string Greeting(string name) { return name; }
}
```

Call static behavior through its type

```
Text.Greeting("Ada");
```

Locals end at the brace

BOUNDARY

Return values needed outside.

Modifiers are contract

PUBLIC

Visible API.

STATIC

No receiver.

VOID

Effect, no value.

Notice `async` and `test`

LATER

Async waits; test runs through `feo test`.

Check from signature outward

☐ Return type?

☐ Argument count and types?

☐ Static or instance call?

Extract one operation

PRACTICE

Give one repeated expression a clear input and result.

Structs and methods

Give one concept one home.

Name the domain shape

STRUCT

Concrete state plus its operations.

Fields are object state

```
struct Task
{
    string title;
}
```

Show the receiver

```
this.title = taskTitle;
```

Establish a usable start

```
public Task(string title)
{
    this.title = title;
}
```

Every new object is valid

CONSTRUCTOR

Initialize what later methods require.

Use `new`

```
Task task = new Task("Pack water");
```

Methods may change state

```
public void Complete()
{
    this.complete = true;
}
```

Queries return an answer

```
public bool IsComplete()
{
    return this.complete;
}
```

Call behavior on the instance

```
task.Complete();
task.IsComplete();
```

Give related rules one owner

STRUCT

Data + domain rules.

OUTSIDE

Unrelated workflow.

Prefer behavior over field poking

API

Name the domain action; hide representation.

Complete one small model

- ☐ Fields
- ☐ Constructor
- ☐ Command
- ☐ Query

PART 10 · CAPABILITIES

Interfaces and traits

Promise less; implement concretely.

Interface = capability

```
interface Greeter
{
    string Greet(string name);
}
```

Implement after `:`

```
struct ConsoleGreeter : Greeter { }
```

Match the callable shape

RULE

Every interface method keeps its signature.

Store the promise

```
Greeter greeter = new ConsoleGreeter();
greeter.Greet("Ada");
```

Write the interface name

NOT RUST

Do not write trait-object syntax in R#.

Leave one type open

```
interface Repository
{
    associated Item;
    Item Load();
}
```

Contract names the placeholder

IMPLEMENTATION

The struct supplies the concrete type.

Bind it beside implementation

```
struct MemoryRepository : Repository
{
    associated Item = string;
}
```

Keep source type simple

SOURCE

Write `Repository`; lowering derives the Rust boundary.

Struct or interface?

STRUCT

Concrete state.

INTERFACE

Caller-facing capability.

Keep trait mechanics out

R\#

Declare the capability; let lowering derive Rust details.

Extract one capability

- ☐ Small promise
- ☐ Concrete implementation
- ☐ No stored fields in interface

Generic types and collections

Read inside out.

Container before operation

TYPE

The argument says what travels inside.

Angle brackets carry shape

```
List<Task>
Option<string>
Result<Task, string>
```

Use `List<T>` for owned order

LIST

One element type; growable sequence.

Construct with `List<T>`

```
var mut tasks = List<Task>();
```

Mutation needs `var mut`

```
tasks.Add(new Task("Pack water"));
```

Read with `At` or brackets

```
var first = tasks.At(0);
var second = tasks[1];
```

Replace with `Set` or brackets

MUTABLE

The list must be mutable.

Absence is a type

```
Option<string> caption = Some("ready");
Option<string> missing = null;
```

Failure is a return case

```
return Ok(3);
return Err("not ready");
```

Read inner value first

ORDER

Value → optionality → collection → result.

Unwrap the shape mentally

```
Result<List<Task>, string>
```

Do not `unwrap()` by habit

RULE

Unwrap only after proving success or presence.

Choose container first

☐ What value?

☐ Which container?

☐ Which operation?

PART 12 · CONTROL

Control flow

Condition, sequence, exit.

Use `if` for one condition

```
if (ready)
{
    Console.WriteLine("go");
}
```

Condition = boolean question

Name the alternate path

```
if (ready) { return "go"; }
else { return "wait"; }
```

Keep two paths visible

Name complicated reasons

Repeat while true

```
while (remaining > 0)
{
    remaining = remaining - 1;
}
```

Show progress

RULE

Move state toward `false`.

Visit every item

```
foreach (var day in days)
{
    Console.WriteLine(day);
}
```

Prefer values to indexes

Generate an index

```
for (var index in 0..count)
{
    Console.WriteLine(index);
}
```

Range supplies each index

Choose the endpoint

```
0..count    // exclusive
0..=count   // inclusive
```

Ranged `for` only

Skip or stop

CONTINUE

Next pass.

BREAK

Leave loop.

SCOPE

Inside loop only.

Handle exceptional items first

Keep exit local

Let work choose the loop

CONDITION

`while`

ITEMS

`foreach`

INDEX

Range `for`

Write the smallest honest flow

PRACTICE

Condition → loop → exit.

PART 13 · OUTCOMES

Patterns, absence, errors

Make every outcome visible.

Put uncertainty in the type

OPTION

Absent?

RESULT

Error?

MATCH

Answer.

Use `Some` or `null`

```
Some("ready")
null
```

Handle one useful shape

```
if let Some(value) = caption
{
    return value;
}
```

The pattern creates the local

Bind after matching

Use match for value-producing cases

```
var label = match count
{
    0 => "empty",
    _ => "items",
};
```

Read arms as answers

Answer present and absent

```
Some(value) => value,
None => "untitled",
```

Return `Ok` or `Err`

```
return Ok(3);
return Err("not ready");
```

Give success and failure answers

One case or every case?

IF LET

One path.

MATCH

All paths.

Arms are expressions

Match supported outcome shapes

Do not erase uncertainty

Make the caller decide

PRACTICE

Return → match → decide.

Ownership and pass-by-reference

Say who keeps the value.

Ordinary types are owned

DEFAULT

Write `string`, `Task`, or `List<Task>`.

Owned argument = handoff

CALL

Non-copy values may move to the callee.

Borrow read-only input

```
int Length(borrowed<string> title)
```

Caller keeps ownership

BORROW

The callee receives a temporary view.

Return a view of existing data

NOT NEW

A borrowed result is not an owned result.

Root the view

RULE

Returned data must outlive the borrow.

Trace it to its owner

```
var longer = Choose(first, second);
```

Let the signature state the relation

R\#

Solver/Cargo check the Rust lifetime relationship.

Choose the API promise

HANDOFF

Owned.

INSPECT

Borrowed.

NEW

Owned return.

Keep Rust lifetimes out

SOURCE

Use `borrowed<T>`, not `&'a T`.

Do not hide an owner

CHOICE

If it must live alone, return owned data.

Make one promise

- ☐ Who owns input?
- ☐ Who may mutate?
- ☐ How long must output live?

Sharing, concurrency, async

Choose the access boundary.

Start owned

Use `shared<T>` for immutable sharing

Shared is not writable

Use `shared_mutable<T>`

Write the domain call

```
counter.Increment();
```

Use `shared_readable<T>`

Choose the promise

SHARED

Immutable.

MUTABLE

Synchronized writes.

READABLE

Reader/writer access.

Choose least power

Non-default shape is recorded

```
representation = "shared_mutable"
access = "mutex"
```

Read; never author

GENERATED

`types.toml` is evidence, not source.

Mark awaited work

```
async string ReadMessage()
{
    return "ready";
}
```

Await the result

```
var message = await ReadMessage();
```

Await ≠ sharing

Put waiting in the contract

```
async Func<T, Result<U, E>> f
```

Name later results

TYPE

Use `future<T>` when a stored future shape matters.

Keep Rust proof out

SOURCE

Solver/Cargo supply function, future, and thread bounds.

Tell the access story

- ☐ Who owns?
- ☐ Who mutates?
- ☐ Who waits?

Choose one boundary

PART 16 · VERIFICATION

Testing, debugging, lowering

Prove the promise; inspect the stages.

Test observable behavior

```
test public static void Works()
{
    AssertEqual(1, 1);
}
```

Keep the boundary small

FORM

Zero-argument static function + `test`.

Choose the assertion

ASSERT

Condition.

EQUAL

Expected value.

FAIL

Impossible path.

Name what failed

TEST

Assert the behavior, not the generated spelling.

Test source-visible behavior

Solve, then test

```
feo solve --solver random app.feo-solution
feo test  --solver random app.feo-solution
```

Await inside `async test`

Failure still points home

MAP

Source maps retain the R# location.

Read source first

- ☐ Authored span
- ☐ Semantic choice
- ☐ Generated detail

Inspect the stages

```
.rsharp → .partial → .rs → Cargo
```

Partial = checkpoint

Map both sides

MAP

`.map` connects R# and Rust spans.

Evidence lives in `.feo/`

```
.feo/generated/
.feo/solver/
```

Clean stale state

```
feo clean app.feo-solution
```

Fix the source boundary

RULE

Do not patch generated Rust.

Build one small solution

Run the whole loop

- ☐ Project
- ☐ R# source
- ☐ Solve/check/test
- ☐ Mapped failure

Ask the next source question

DONE

You have the R# map.

Know what to ask next

ORGANIZE

Solution → project → source.

MODEL

Types → behavior → boundaries.

VERIFY

Solve → check → test → inspect.

EDITION RECORD

Shorthand counts and build information

2341

NON-CODE WORDS

493

CODE WORDS

212

PAGES

MEASUREMENT

Each part is checked below 250 combined Wordometer words. The PDF is A5 landscape and uses the same semantic slide elements and R# syntax theme as the full manual.