ifxtools
Document Version: 1.0 – 2026-08-13

Informix 4GL on GraalVM

Airtool

Content

This paper describes a second backend for Informix 4GL: one that does not translate the program, but **runs it as itself** on GraalVM as a registered language of the platform — parsed into a tree of executable nodes, specialised by partial evaluation, and compiled to machine code by the same optimising compiler that handles the Java in the same process.

It is written for two readers. The **database architect** will want to know what happens to transactions, cursors, isolation, decimal fidelity and the catalogue when a 4GL program stops being a client process and becomes a routine inside an application server. The **platform architect** will want to know what GraalVM actually is, how a compiler written in Java attaches to a production JVM, what Truffle does to an interpreter to make it fast, and what the operational surface looks like afterwards. Both questions are answered here in the terms the answer is normally hidden behind.

> **The claim, precisely.** Informix 4GL is implemented as a Truffle language registered under the identifier `i4gl`, MIME type `application/x-4gl`, and reachable through JSR-223 as the script engine named `4gl`. A hot 4GL routine is compiled to native instructions by the Graal compiler and appears in its compilation log under its own name and 4GL source line. It shares one heap with Java, JavaScript and Python in the same process, and is lent the host's pooled, transacted connection rather than opening one of its own. This is not emulation, and it is not a program in another language reading yours: it is a language implementation on a production JVM.

1. Why a second backend exists at all

The companion paper, `Informix 4GL to Java`, describes a migration compiler: it reads 4GL and emits Java a team can review and adopt as the source whenever it chooses. That answers the question `what should this program become`.

There is a second question a compiler cannot answer: `what does this program do, right now, against real data` — before anyone has agreed to keep a translation, and without keeping the original machine alive to ask. And a third, which arrives once an organisation looks honestly at its estate: `why are we rewriting a rule that is correct`.

Both are answered by running the 4GL as itself. The same front end serves both backends — parsing, `LIKE` resolution against the live catalogue, and module linking are the compiler's, unchanged — and only the last step differs. One emits Java source; the other builds executable nodes and runs them.

	Migration backend	GraalVM backend
Produces	Java source you own	Nothing on disk; the program runs
Uses `javac`	Yes	Never
The 4GL afterwards	Optional — keep it or retire it	Stays the source, permanently
Screens	Reproduced on a terminal library	Not provided — see the limits
Runs inside	Any JVM, as an application	A host application server, as a language
Owns its connection	Yes — opens the database it declares	No — is lent the host's

The dependency runs one way: the GraalVM backend is built on the compiler's front end, and nothing in a migrated application ships it. An estate can adopt either, or both, without one constraining the other.

2. GraalVM — the platform, in the detail an architect has to approve

GraalVM is Java. It is an OpenJDK-based JDK, developed as open source in the open, freely downloadable and redistributable in its Community Edition. It installs where a JDK installs, and existing Java applications run on it unchanged — because it is not a different virtual machine, but the same one with a different optimising compiler inside it.

2.1. How a compiler written in Java attaches to the JVM

For two decades the JVM's optimising compiler was written in C++ and understood one language. Replacing it was not a matter of ambition but of interface: HotSpot had no way to accept a compiler from outside itself.

The **JVM Compiler Interface** — JVMCI, a standard part of Java since JDK 9 — is that interface. It exposes to Java code the things a compiler needs from the virtual machine: the bytecode of a method, the profile the interpreter collected while running it, the layout of objects, the constant pool, and a way to install the resulting machine code so the VM will use it. A compiler written in Java can therefore be a first-class compiler for the JVM, called by the same runtime that calls the C++ one.

This matters commercially more than it looks. It means GraalVM is an evolution of the standard platform rather than a fork of it: the class library is OpenJDK's, the garbage collectors are OpenJDK's, the tooling is OpenJDK's, and the compiler is a component attached through a documented interface that is part of Java itself. An organisation adopting it is making a JDK decision, not taking a dependency on a parallel runtime.

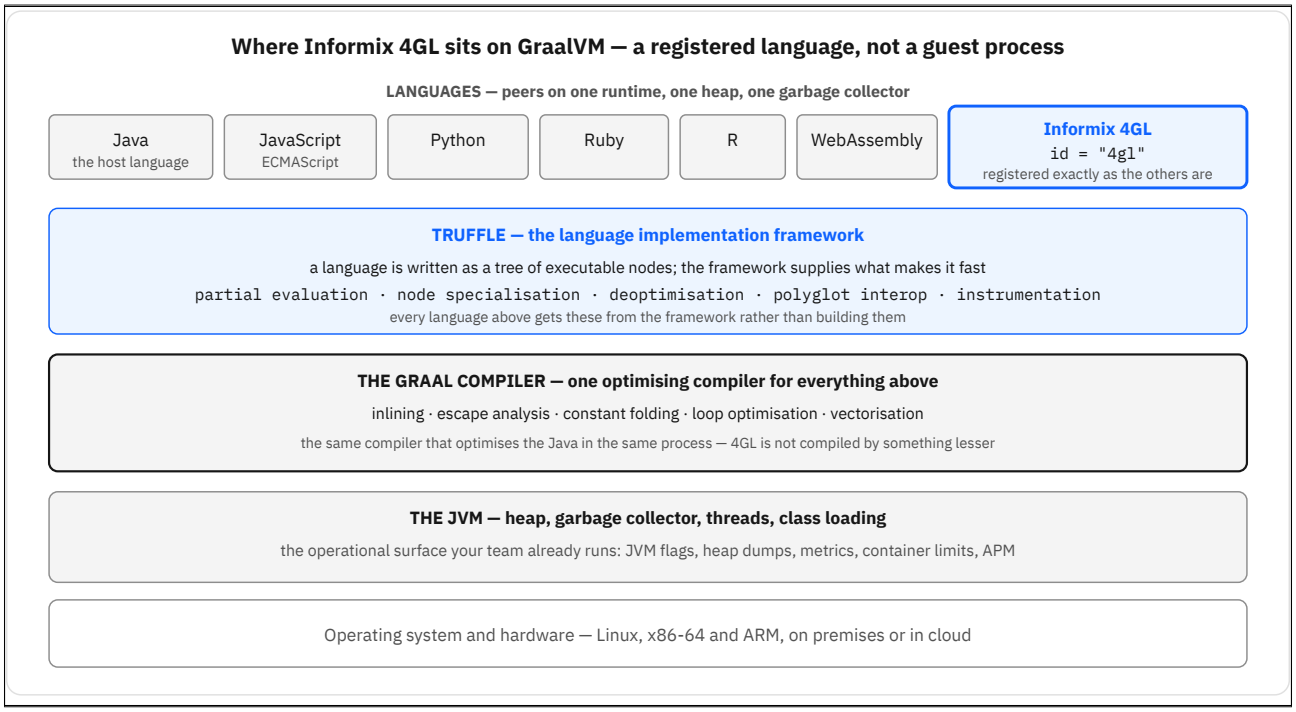

Figure 1 — The GraalVM stack, and the row Informix 4GL occupies in it

2.2. What the Graal compiler does differently

Graal's intermediate representation is a **graph of nodes** rather than a list of instructions — often called a sea-of-nodes IR. Data flow and control flow are edges in one graph, which means many optimisations are local graph rewrites rather than passes that must reason about statement order. Nodes float until scheduling pins them, so a computation naturally sinks to the place it is actually needed, and dead computation simply has no edges into it.

The optimisations themselves are the ones an architect expects — inlining, escape analysis, constant folding, loop unrolling and peeling, dead-code elimination, strength reduction — but two are worth naming because they carry disproportionate weight for the workload described in this paper.

Partial escape analysis. Classical escape analysis removes an allocation only if the object never escapes on any path. Graal's is path-sensitive: it can keep an object `virtual` along the paths where it does not escape and materialise it only on the paths where it does. For an interpreter — which allocates frames, boxes values and creates short-lived intermediates constantly — this is the difference between eliminating that overhead and merely reducing it.

Aggressive inlining across the interpreter. Because a compiled 4GL routine is produced by specialising the interpreter, the "call graph" being inlined is the interpreter's own node `execute` methods. Graal inlines through them until what remains is the arithmetic and control flow the 4GL statement actually performs.

2.3. Tiers, and what they cost

Nothing is compiled immediately, because compiling code that runs twice is waste. Execution starts interpreted and gathers profile; a method that becomes warm is compiled quickly with limited optimisation; a method that dominates is recompiled with the full pipeline. Truffle applies the same principle one level up, to `guest` code: a 4GL routine crosses a call threshold, is compiled at a first tier, and if it keeps running is recompiled at a second.

The operational consequences are the ones any JVM workload has, and are worth stating for estates whose 4GL currently starts as a process per user. Compilation happens on background threads and costs CPU while it runs — the trace later in this paper shows tens of milliseconds per routine. Peak performance arrives after warm-up rather than at the first request, which is irrelevant for a long-running server and relevant for a short batch. And a process that runs for hours amortises all of it, which is the normal shape of the work 4GL is used for.

3. Truffle — how a language becomes native on this platform

Writing a language implementation used to mean writing a compiler back end: instruction selection, register allocation, a garbage collector, a debugger. Truffle removes that work with one idea, and the idea is worth understanding rather than taking on trust, because everything this backend claims follows from it.

3.1. The interpreter, and the projection that compiles it

A Truffle language is written as an ordinary abstract-syntax-tree interpreter. Each node knows how to execute itself and calls its children; a function becomes a tree rooted at a `call target`. Written naively this is the slowest respectable way to run a language.

Partial evaluation is what makes it fast. An interpreter is a program of two inputs: the program to run, and its data. Hold the first fixed — this specific 4GL routine — and specialise the interpreter with respect to it, and what remains is a program of one input, the data. That residual program is exactly a compiled version of the routine, derived mechanically. This is the `first Futamura projection`, a result from 1971 that spent decades as a curiosity and is now the engine underneath every language on this platform.

Concretely: the node objects, the virtual calls between them, the field loads that fetch a child node, and the interpreter's own dispatch all become compile-time constants, and constant-fold away. What is left in the machine code is the arithmetic and the branches the 4GL statements describe.

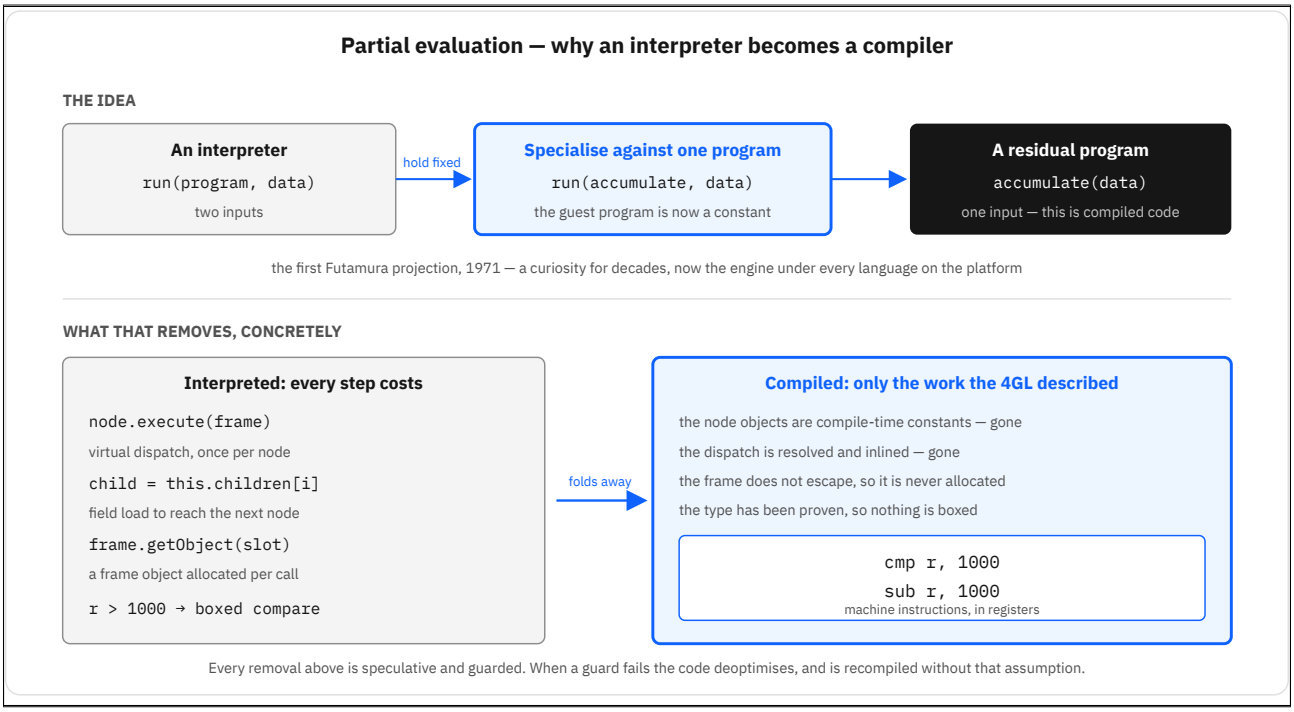

Figure 2 — Partial evaluation — why an interpreter becomes a compiler

3.2. Self-specialisation: the interpreter rewrites itself

Partial evaluation on its own would compile a generic interpreter — one that still checks, at every operation, which of the possible types it has. Truffle nodes avoid that by specialising themselves `while interpreting`, before compilation happens.

An addition node starts in an uninitialised state. The first time it executes with two integers it rewrites itself into an integer-addition node; if it later sees a decimal it rewrites again into a more general form. By the time the routine is hot, the tree has recorded what the program actually does, and the compiler is specialising

a tree that is already narrow. This is why a language with no static types can approach the performance of one that has them — and why a language like 4GL, whose variables carry declared types from their `DEFINE`, starts out narrower than most.

The state a node reaches is a speculation, and it is guarded. If the guard fails, the node generalises and any compiled code that relied on the narrower assumption is discarded — which is the subject of the deoptimisation section below.

3.3. Assumptions, frames, and boundaries

Three further mechanisms account for most of the difference between a fast Truffle language and a slow one, and this implementation uses all three.

Assumptions. A value the runtime believes stable — a resolved function target, a configuration that has not changed — can be marked as a compile-time constant with an assumption attached. Compiled code contains no check for it at all. Changing the value invalidates the assumption, which invalidates every compilation that depended on it, and the affected routines are recompiled. The cost of the check moves from every execution to the rare moment the world changes.

Virtual frames. A guest function's local variables live in a frame object. Allocating one per call would dominate the cost of small functions, so frames are declared in a way the compiler can see through: partial escape analysis proves the frame does not escape the compiled region and removes the allocation entirely, leaving the locals in machine registers. A 4GL `FUNCTION` called three million times in a loop allocates nothing per call once compiled.

Compilation boundaries. Some calls must not be compiled into the specialised body — the SQL session, the report engine, anything that ends in JDBC or I/O. They are marked as boundaries, and the compiler treats them as opaque calls rather than following them. Without this discipline a single compilation unit would grow to include the whole runtime and the driver beneath it; with it, a compiled 4GL routine stays a compact piece of code that calls out to the database exactly where the 4GL said it should.

3.4. Splitting, and why one routine can have several compiled forms

A utility function called from many places sees many different argument profiles, and a single specialised tree for it would have to generalise until it is fast for nobody. Truffle addresses this by **splitting**: duplicating a call target for a particular call site so each copy can specialise to what that site actually passes.

This is worth knowing when reading a compilation log, because the same 4GL function name can legitimately appear more than once with different identifiers. It is not the runtime thrashing; it is one function being compiled twice for two different callers, each version narrower than a shared one could be.

3.5. Interoperability: how another language calls a 4GL routine

Truffle defines a message-based protocol that all its languages implement — is this value a number, does it have members, is it executable, read this member, execute it with these arguments. A language does not know about any other language; it knows about the protocol.

The consequence is that values cross language boundaries without serialisation, because there is no boundary in the memory sense: a 4GL value handed to JavaScript is the same object on the same heap, and the compiler can inline through the protocol calls, so a cross-language call in a hot path costs approximately what a same-language call costs.

3.6. Instrumentation — what the framework offers, and what this implementation exposes

Truffle also provides an instrumentation framework: a way for tools to attach to a running program at node granularity, which is where the platform's debugger, profiler and coverage tools come from. A language opts in by tagging the nodes that constitute a statement, a root, or a call.

This implementation does not yet expose those tags. Its nodes carry accurate source sections — which is why a refused statement names its file and line, and why the compilation log attributes machine code to a 4GL line — but the statement and root tags a debugger attaches to are not implemented. Stepping a 4GL program in the platform's debugger is therefore not available today. The data model it needs is in place; the tags are the remaining work, and this paper will claim stepping when they land and not before.

4. The 4GL language implementation

The identifiers below are the checkable surface of the claim that this is a language on the platform rather than a library that reads 4GL.

Property	Value	Where it shows
Truffle language ID	`i4gl`	Polyglot API, `Context.eval("i4gl", ...)`
Display name	Informix 4GL	Tooling and engine listings
MIME type	`application/x-4gl`	Source detection, `.4gl` files
JSR-223 engine name	`4gl` (also `i4gl`, `fgl`, `informix-4gl`)	`getEngineByName("4gl")`
Root node naming	`4gl:` + function name	The compilation log, profilers, stack traces
Context policy	`SHARED`	See `Contexts, threads and isolation`

One consequence is worth drawing out immediately, because it is the reason a compilation log is readable at all. Root nodes are named for the 4GL function they came from, so every tool that reports on compiled code — the compilation log, a sampling profiler, a stack trace — speaks in terms of the source the business wrote, not in terms of the interpreter's internals.

5. Editing 4GL, on a platform that runs it directly

The shared front end has a consequence that is easy to overlook next to the compilation machinery: everything an engineer sees while `writing` 4GL is the same on both paths, because it comes from the same parser. The vocabulary an editor colours and the wording of a syntax error are properties of the front end, not of the backend that runs afterwards.

5.1. An editor that colours exactly what the front end accepts

The distribution carries syntax colouring for the editors an enterprise team already standardises on: a TextMate grammar that serves Eclipse and Visual Studio Code, and a file type definition for IntelliJ IDEA and the other JetBrains IDEs. Nobody is asked to adopt a new tool in order to keep maintaining code they have maintained for twenty years.

Both files are generated from the lexer, not written by hand — and the lexer in question is the one this backend parses with, so the words an editor colours are exactly the words that will run. A hand-maintained keyword list is a second, informal statement of what the language is; it starts slightly wrong and drifts with every grammar change, and its errors are invisible, because a word that should be coloured and is not looks merely like a word. A test in the build fails when the grammar gains a keyword the editor files do not know.

296 keywords, in four groups: 131 statement and control keywords, 40 types, 74 SQL keywords, and 51 built-in functions, display attributes and constants — together with comments in all three of the language's forms, both kinds of string literal, and the name in a `FUNCTION` or `REPORT` header. Four groups because the JetBrains format offers exactly four keyword lists; the partition is required to be total, so a keyword added to the grammar and forgotten here is a failing test rather than a word that quietly stays black on screen.

None of it parses 4GL. Colouring is a lexical judgement made a token at a time and will happily colour a program that is not valid, which is what the next part is for.

5.2. One front end, one diagnostic

A syntax error is reported against the file, the line and the column that caused it, in the form every compiler has used for forty years. Here is a program with a deliberate typo — `LET s xLIKE "hello"` on line 6 — run directly on this platform, with no translation step anywhere in the picture:

```
$ ./i4gl broken.4gl

broken.4gl has syntax errors:
  broken.4gl:6:10: error: missing EQUAL at 'xLIKE'
  broken.4gl:6:16: error: mismatched input '"hello"' expecting {EQUAL, '[', '.'}
```

Two properties of that output matter more than its format. The first is that it is the `whole file's` errors rather than the first one — a parser that stops at the first error turns fixing a file into a sequence of builds. The second is that translating the same file to Java produces those two lines and no others, character for character, because there is one front end and it has one opinion. An estate that moves a program between the two paths does not get a second dialect of correctness along with it.

On this path the diagnostic arrives when the source is evaluated — a 4GL program is parsed, resolved against the catalogue and linked at the moment the context asks for it, so a broken file fails there rather than at a build step that this path does not have. In an embedding, that is the `eval` call, and the failure is an ordinary polyglot exception the host can catch.

5.3. Testing a program that runs on this backend

The companion paper devotes a chapter to putting migrated screens under JUnit, driven by an in-memory terminal with keystrokes queued ahead of the run. None of that applies here, because this backend has no

screen. What it has instead is a language reachable from ordinary Java, which turns out to be the more direct way to test the thing worth testing.

A 4GL program is evaluated through the standard scripting interface, and its output is captured through a writer rather than a terminal — deliberately, because a host embeds a language in order to send its output to a response or a log, never to the process's own standard output:

```
@Test
void thePricingRuleStillPrices() throws Exception {
    ScriptEngine engine = new ScriptEngineManager().getEngineByName("4gl");

    StringWriter output = new StringWriter();
    engine.getContext().setWriter(output);        // not the process's stdout

    engine.eval(program);

    assertThat(output.toString()).contains("total=         15");
}
```

Two things about that test matter more than its brevity. It is `ordinary` JUnit — no subprocess, no harness, no 4GL-specific test framework to learn or maintain — and it runs in the build the organisation already has. And a failure inside the 4GL arrives as an ordinary scripting exception naming the 4GL line, so a red test points at a statement rather than at a stack trace that has to be decoded first.

For an estate mid-migration there is a stronger test available than either backend can offer alone, and it is the one we hold ourselves to on every build: **run the same program through both backends and compare the output line for line**, and reports byte for byte.

```
@Test
@DisplayName("both backends print the same lines for the same program")
void backendsAgree() throws Exception {
    List<String> onGraal = Programs.run(program).output();
    List<String> asJava  = JavaBackend.displayedLines(program, out);

    assertThat(onGraal).isEqualTo(asJava);
}
```

That is worth more than it looks. Two implementations of a language are two answers to the question of what the language means, and the dangerous failure between them is never a crash — it is both running and quietly disagreeing about a total. The same technique compares either backend against the original Informix binaries, which is how a migration wave earns the right to go into production: not because the translation compiled, but because a test that runs on every build says it still produces what the original produced.

5.4. Where the IDE integration comes from, and what it belongs to

The migration backend adds one thing this path does not have, and the distinction is worth being exact about because a screenshot invites the wrong conclusion. Translation runs as a Maven build participant, so it reports through the IDE's own problem model: the marker lands on the statement, and the error is counted with everything else the project has to say about itself.

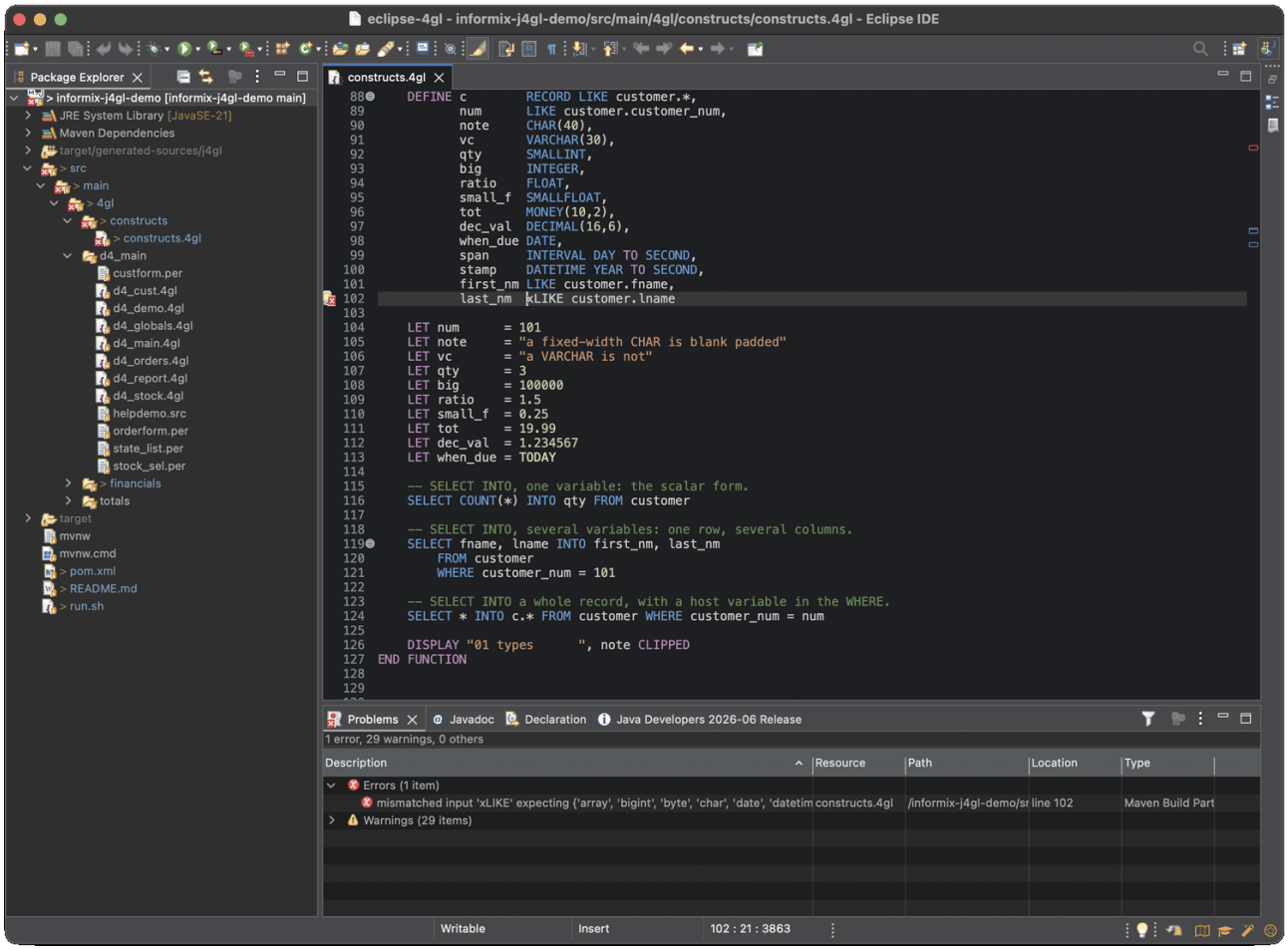

Figure 3 — A 4GL syntax error in Eclipse, reported by the migration build

The colouring in that window is this path's as much as the other's — same files, same lexer, same 296 words. What belongs to the migration backend specifically is the marker in the ruler and the row in the Problems view, because those come from the build participant that produces Java. Running a `.4gl` file directly on the platform is not a build, so there is no build to report through: the same two lines of diagnostic arrive on the console, or as an exception in whatever host called `eval`.

The same division decides the rest of the loop. On the migration path a program becomes Java, so it is debugged, profiled and covered by the tools the organisation already owns — nothing was built for it, which is why nothing has to be maintained for it. On this path the equivalent is Truffle's instrumentation framework, and this implementation does not yet expose the tags a debugger attaches to: the nodes carry accurate source sections, which is why a refused statement names its file and line and the compilation log attributes machine code to a 4GL line, but stepping is not available today and this paper will claim it when the tags land and not before.

> **This is the practical shape of the choice the paper keeps returning to.** An estate being migrated gets the IDE integration because it has a build. A 4GL routine embedded in a Java service, or run as itself while the rules are still being written in 4GL, gets the same diagnostics through the channel that suits it. Neither path asks the team to learn a different language, a different editor, or a different definition of what a valid program is.

6. A compilation trace, read line by line

The following program does nothing interesting except repeat itself, which is exactly what is needed to make the runtime decide a routine is worth compiling.

```
MAIN
    DEFINE i INTEGER, total INTEGER
    LET total = 0
    FOR i = 1 TO 3000000
        LET total = total + accumulate(i)
        IF total > 1000000 THEN
            LET total = 0
        END IF
    END FOR
    DISPLAY "total = ", total
END MAIN

FUNCTION accumulate(n)
    DEFINE n INTEGER, r INTEGER
    LET r = n
    IF r > 1000 THEN
        LET r = r - 1000
    END IF
    RETURN r
END FUNCTION
```

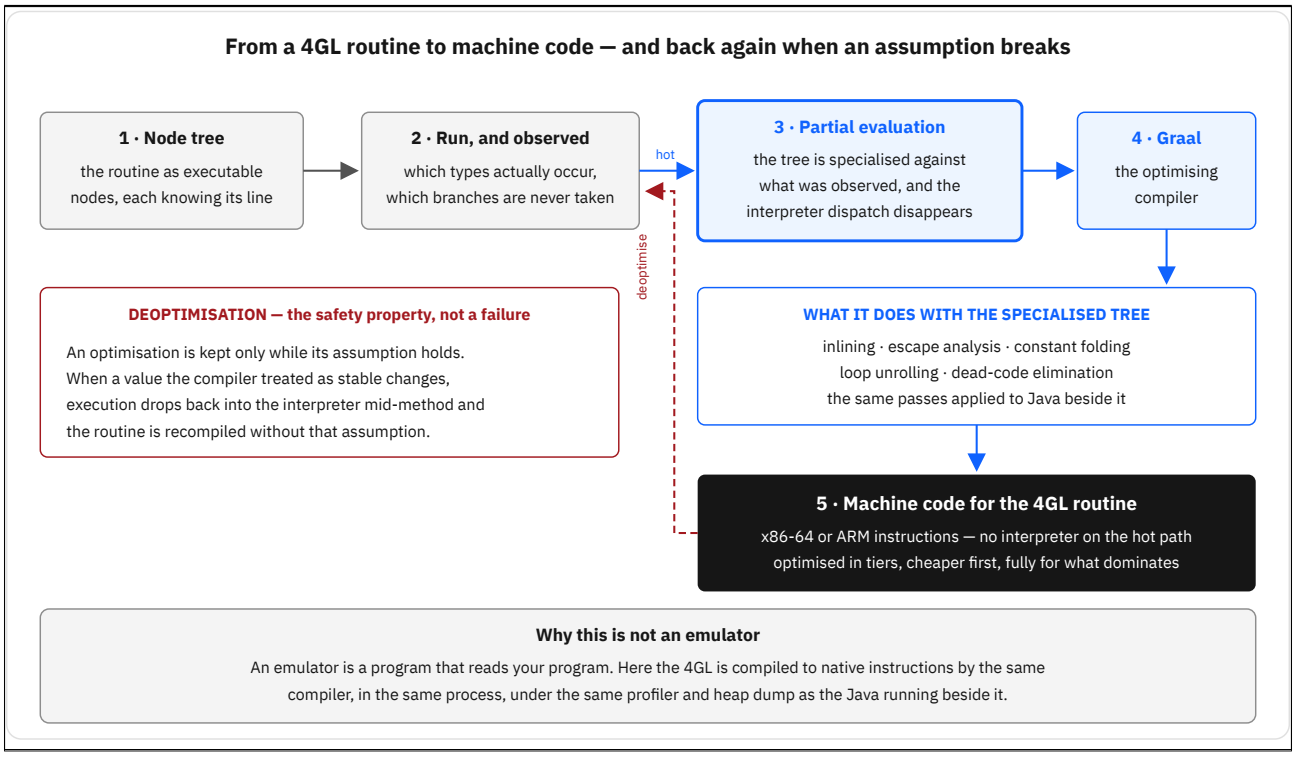

Figure 4 — From a 4GL routine to machine code, and the loop that keeps it honest

Run on a GraalVM JDK with compilation tracing enabled, the engine reports what it compiled. This is the actual output. The engine prints one line per compilation, wider than a page; the `[engine]` prefix and the columns explained in the table below are kept, the rest trimmed:

```
opt done  4gl:accumulate  |Tier 1|Time 21ms|AST 20|IR 186/320|Code 1520|Src hot.4gl:13
opt done  4gl:accumulate  |Tier 2|Time 13ms|AST 20|IR 114/218|Code 1324|Src hot.4gl:13
opt done  ForNode.Body<OSR> |Tier 2|Time 29ms|AST 19|IR 349/1191|Code 6276|Inlined 1Y 0N
```

Field	What it tells you
`4gl:accumulate`	The unit of compilation is your `FUNCTION`, by name — not a block or a synthetic method.
`Src hot.4gl:13`	Machine code attributed to a 4GL source line. Line 13 is where `FUNCTION accumulate` is declared.
`Tier 1` then `Tier 2`	Tiered compilation of guest code: a cheaper pass for warm code, the full pipeline for code that dominates. Seven milliseconds apart here.
`AST 20`	The routine is twenty interpreter nodes — the size of the tree being specialised.
`IR 186/320` → `114/218`	Graal graph nodes after and before optimisation. The second tier produced a `smaller` graph from the same tree, because it had more proven information to work with.
`CodeSize 1520` → `1324`	Machine-code bytes. Less code doing the same work is the clearest evidence that optimisation, not translation, is happening.
`ForNode.Body<OSR>`	On-stack replacement: the loop body was compiled `while running` and execution transferred into the compiled version.
`Inlined 1Y 0N`	One call inlined, none rejected. The call to `accumulate` disappeared into the loop.
`Time 21( 18+3 )ms`	Compilation cost, split between the Truffle phase and the Graal phase. This is CPU spent on a background thread, not latency added to the request.

Why on-stack replacement matters here more than in most workloads. A nightly batch typically enters one loop and stays in it for the entire run. Compilation triggered by `method entry` would never fire for that loop, because the method containing it is never re-entered — the program would interpret for four hours. OSR compiles the loop body mid-flight and moves execution into it. For 4GL, whose characteristic program is exactly this shape, it is the difference between the platform being suitable and not.

> **On performance expectations.** A 4GL business program spends nearly all of its wall-clock time in the database — cursors, fetches, locks, commits — and very little inside the language. That SQL is unchanged here: the same statements, against the same Informix server, over the host's pooled connection. What the trace above governs is the arithmetic, loops and string handling `around` the SQL, and after warm-up that work is compiled by the same compiler that optimises the Java beside it. The realistic expectation is a program bounded by the database exactly as it is today, with the compute around it faster than the original runtime made it.

7. Deoptimisation — the property that makes speculation safe

Every optimisation described so far is speculative. Compiling a variable as an integer because it has only ever held integers is a bet; inlining a call because the target has never changed is a bet; removing a frame allocation because the frame has never escaped is a bet. The runtime must be able to lose any of them without producing a wrong answer.

Deoptimisation is the mechanism, and it is more than a fallback path. When a guard fails, execution must leave compiled code in the middle of a method and resume in the interpreter at exactly the corresponding point — which means the interpreter's state has to be reconstructed from a machine-code frame whose variables may be in registers, folded into constants, or eliminated entirely. The compiler therefore records, for every point where a guard can fail, enough metadata to rebuild the interpreter frame. That metadata is why aggressive optimisation is safe: the compiler may remove anything it can prove is unnecessary, provided it can describe how to put it back.

For an architect assessing risk, this inverts the usual question. It is not whether an aggressive optimiser might be wrong; it is whether the runtime can retreat when its assumptions stop holding. This one can, at instruction granularity, with the guest program's semantics preserved exactly. A platform without that property must either optimise timidly or accept being occasionally wrong, and neither is acceptable for financial arithmetic.

It also explains behaviour that would otherwise look like a defect. A routine may be compiled more than once; a log may show the same name recompiled after a period of stability. That is the system correcting a speculation — a value that had been constant changed, an assumption was invalidated, and the affected code was rebuilt without it.

8. Contexts, threads and isolation

A polyglot **context** is the unit of guest-program state: its global values, its open resources, its bindings. An **engine** owns the compiled code and the profile behind it. The relationship between the two is what determines how a language behaves inside a server, and it is the part most often glossed over.

This language declares a **shared** context policy, which means one language instance can serve many contexts and the compiled code produced for a routine is reusable across them. The practical consequence for a request-scoped deployment is direct: a server can give each request its own context — its own variables, its own connection, no state leaking between users — while the machine code for a pricing routine is compiled once and used by every request that follows. Isolation of state does not cost re-compilation.

Threading follows the ordinary Truffle model, and the safe deployment shape is the obvious one: a context is entered by one thread at a time, and a request-scoped context is entered by the thread serving that request. This is also what the database contract requires — the connection lent to the context belongs to that request, and JDBC connections are not meant to be shared across threads.

Host access is a policy rather than a default. A context is built with an explicit statement of what the guest may reach in the host: which Java classes and members are visible, whether other languages may be called. The embedding described later grants what the host intends and nothing more, and the 4GL side of it needs very little — the connection is the whole interface, which is a security property as much as a design one.

9. The database contract

This is the section a database architect should read first, because moving a 4GL program from a client process into an application server changes who owns the connection — and everything about transactions, isolation and identity follows from that.

9.1. The connection is the host's, and stays that way

Inside a server the connection is not the language's to open. The host already has one: pooled, transacted, audited, and authenticated as the user whose request is being served. A language that opened its own would run outside that transaction, outside the pool, and as whatever user its own settings named — which is how a scripting facility becomes a hole in an access-control model that every other component respects.

So the connection is lent, and three consequences follow deliberately.

The engine opens no connection of its own when one is lent. The 4GL runs inside the caller's transaction: its `INSERT` and `UPDATE` statements are part of the request's unit of work, and they commit or roll back with it. A rule invoked twice in one request sees its own earlier writes, exactly as it would inside a stored procedure.

It never closes the one it is given. Closing is disarmed on the borrowed connection, because closing a pooled connection mid-request would end the caller's transaction and return a live handle to the pool. A script cannot terminate the transaction that invoked it.

The catalogue is read from that same connection. Informix resolves column types at compile time, so `DEFINE n LIKE customer.customer_num` and `DEFINE p RECORD LIKE customer.*` must resolve against the database the script will actually read. This is not a convenience: a record built from a stale catalogue in a different column order will bind a `FOREACH … INTO p.*` positionally and put the city in the postcode, silently. Reading the catalogue from the lent connection makes that class of error impossible.

A script therefore needs no `DATABASE` statement when a connection is lent — the database is whichever one the host handed over. If no connection is lent and the script runs SQL, it stops at that statement and says how to lend one, rather than quietly finding one elsewhere.

9.2. Transactions, isolation and locking

Because the connection is the host's, the transactional properties are the host's too, and this is the answer to most of the questions a DBA will raise. Isolation level, lock-wait behaviour and the logging mode of the database are whatever the host configured for that connection. The 4GL does not set them and cannot silently change them for its own convenience.

4GL's own transaction statements — `BEGIN WORK`, `COMMIT WORK`, `ROLLBACK WORK` — are the point where an estate must think, and the thinking is the same thinking that applies to any code called inside someone else's transaction. A rule intended to be embedded should not commit; committing inside a request that has not finished makes the rest of the request non-atomic. In practice the routines worth embedding are calculations and validations that read, and the ones that write belong at the edges where the host controls the boundary.

Cursors follow the connection's lifetime, and forward-only iteration — `FOREACH`, or a hand-managed `OPEN` / `FETCH` / `CLOSE` — is what this backend implements today. A cursor left open at the end of a script is closed with the context rather than leaked into the pooled connection.

9.3. Type fidelity, which is the whole reason to preserve the rules

A rule is only preserved if its arithmetic is preserved, and this is where translations to a general-purpose language most often fail quietly.

`DECIMAL(p,s)` and `MONEY(p,s)` carry a declared precision and scale, and every operation on them is exact within it. Both backends implement them with the same runtime decimal type, which carries its declared

precision and scale as data and rounds where Informix rounds. There is no floating point anywhere in the path, no scale argument a developer must remember, and — because both backends call the same implementation — no possibility of the two disagreeing about what a money value is.

The widths and scales themselves come from the catalogue at load time, which is what makes a `CHAR(15)` hold fifteen characters rather than whatever the row happened to contain, and what makes a record's members appear in the order the table defines. Character decoding follows the locale settings the connection was established with, which are the host's, so an estate with a non-default `DB_LOCALE` gets its own encoding rather than an assumed one.

`NULL` is the other place a naive implementation loses fidelity. 4GL's `NULL` semantics — including the fact that `SUM` over no rows is `NULL` rather than zero, and that comparisons involving `NULL` are not false but unknown — are implemented in the shared runtime rather than mapped onto the host language's notion of absence.

9.4. What this changes for the database, operationally

Very little, and that is the point. The statements reaching Informix are the statements the 4GL contained. They arrive over the host's JDBC connection, through the host's pool, under the host's credentials, inside the host's transaction. Query plans do not change because the SQL does not change. Monitoring that watches sessions, locks and long-running statements sees the same shapes it saw before, attributed to the application server rather than to a per-user 4GL process.

The one genuine change is topology: a per-user client process becomes a thread in a server holding a pooled connection. For most estates this reduces connection count and makes it predictable, which is usually an improvement in the database's eyes rather than a complication.

10. Embedding: 4GL inside an application server

The deliverable is one jar on the classpath. A host that already evaluates scripts in other languages evaluates 4GL by adding it, with no host-side knowledge of Truffle, the compiler or the intermediate representation.

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("4gl");
engine.put("connection", request.getConnection());   // the host's, not ours
engine.getContext().setWriter(responseWriter);       // DISPLAY goes here
engine.eval(scriptBody);
```

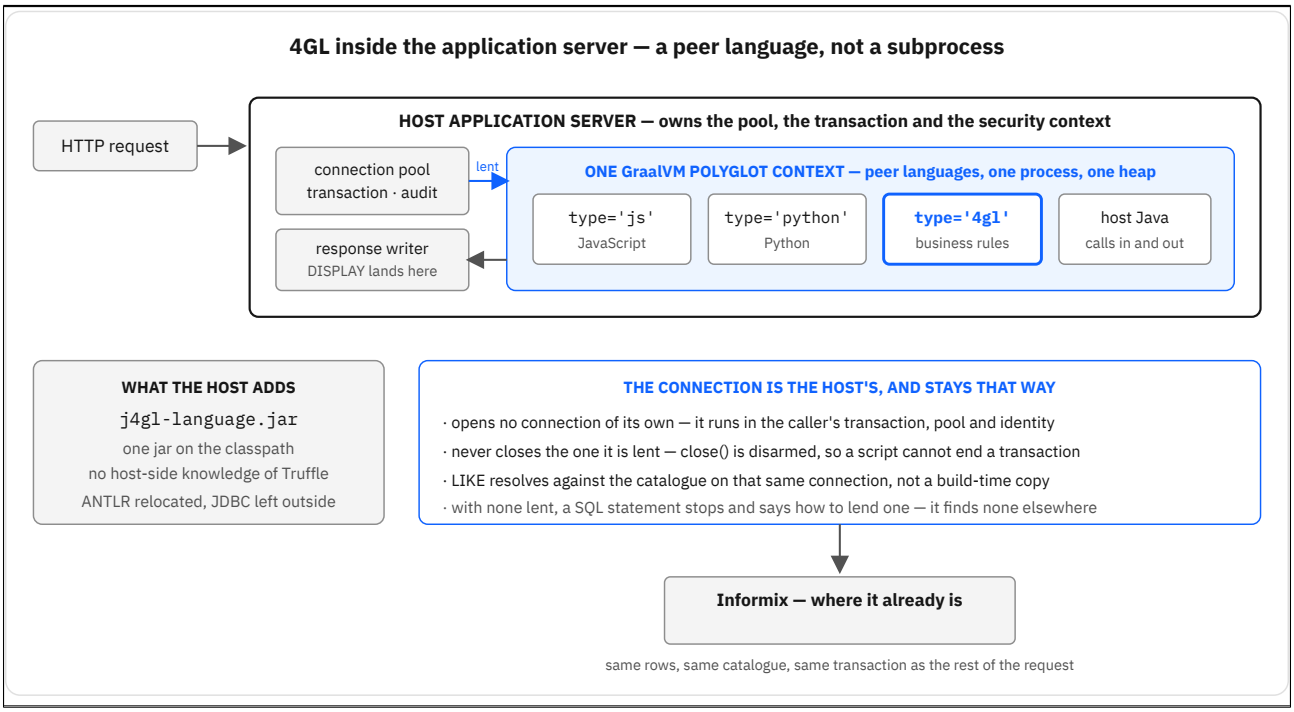

Figure 5 — 4GL inside the application server — a peer language, not a subprocess

Inside the jar	Why
The 4GL front end, IR and emitters	Compiling a script body is what the language does
The 4GL runtime	The same semantics a migrated program uses — one implementation, not two
Parser and terminal libraries, relocated	A host with its own copy of them is unaffected

Deliberately outside	Why
The polyglot runtime	A language jar shares the `host's` runtime; a second copy in one process does not load beside the first
The Informix JDBC driver	The host owns the pool and the driver; two drivers for one server in one classloader is a support call nobody wants

Peer languages share one heap. A pricing rule written in 4GL in 1994 can be called by a service handler written last week, and the result used without marshalling — there is no process boundary, no serialisation format and no socket. This is what separates the approach from an emulator or a subprocess: 4GL is not being simulated

by a program in another language, and it is not being shelled out to. It is a language in the same process, compiled by the same compiler, sharing the same heap and garbage collector — and therefore visible to the same profiler, heap dump and monitoring agent as everything else the server runs.

11. Operating it

Concern	What an operator sees
Runtime dependency	A GraalVM JDK. On a JVM without it the language still runs, as a plain tree interpreter — correct, and slower.
Memory	Guest values are ordinary heap objects. Heap sizing, garbage-collector choice and container limits apply exactly as to the Java beside them.
CPU	Compilation runs on background threads and costs CPU during warm-up. Steady state is compiled code.
Profiling	A sampling profiler attributes time to `4gl:`-prefixed frames, so a hot 4GL function is identifiable by name.
Heap dumps	Ordinary. Guest objects are Java objects; the usual tooling reads them.
Diagnostics	A refused statement names the 4GL file and line. Compilation behaviour is observable through the engine's own tracing, as shown earlier.

The summary an operations team needs is that nothing new is introduced. There is no additional daemon, no separate licence service, no side-channel to monitor, and no unfamiliar failure mode. A 4GL routine that misbehaves does so inside a JVM, and every instrument already pointed at that JVM reports on it.

12. The limits, stated plainly

This backend has no screen, and that is a design decision rather than an omission. A program reached from another language, from a debugger, or from an HTTP request has no terminal to put into raw mode. The statements that need one — `OPEN FORM`, `MENU`, `INPUT`, `CONSTRUCT`, `DISPLAY ARRAY` and their neighbours — stop the program and name the file and line.

They are **refused rather than skipped**, and the difference is the whole argument. A `MENU` that quietly did nothing would not run the program without its menu; it would run a different program, leaving every variable the dialog should have set unset and failing later somewhere with no visible connection to the cause. That is the class of silent mistranslation this project ranks below a missing feature.

Loading a program reports what it will need before anything runs, so the answer arrives at load time rather than halfway through a batch. Several constructs are also not yet lowered and are refused the same way, among them dynamic SQL, report `GROUP` blocks and their aggregates, `GOTO`, and non-forward `FETCH`. Step debugging is not available, for the reason given in the instrumentation section.

Which of your programs this reaches is a question about your programs rather than a general claim, and it is answered the same way every time: load them and read what comes back. A program that needs nothing on that list runs; one that needs something says so by name and line, before it starts. That takes minutes over a directory of 4GL.

> **The division that falls out in practice.** Batch, reports, calculation and rule evaluation embed well — they are the programs whose value is their logic and whose interface is incidental. Operator screens do not, and should not: an interactive 4GL program is a terminal application, and carrying it into a browser unchanged reproduces the terminal in a new place. Screen programs take the migration route, where the full dialog implementation lives, or are rebuilt as a modern interface in front of the rules this backend runs.

13. Two backends, one meaning

Two implementations of a language are two answers to what that language means, and the dangerous failure is not a crash. It is both implementations running, printing something plausible, and disagreeing.

Three things hold the line, and they are structural rather than procedural.

One runtime. Every operator calls the same runtime helper the generated Java calls, from the same table. 4GL arithmetic, comparison and `NULL` semantics have exactly one implementation. In particular the decimal type is shared, so a `MONEY(10,2)` computed here and the same computation in migrated Java produce the same value because they are the same code.

One SQL binder. The component that places statement parameters serves both backends. It was extracted from the Java emitter for precisely this reason: parameter placement is easy to get subtly different twice.

Differential testing between them. The same programs run through both paths and their output is compared line for line; report output, which both produce without a terminal, is compared byte for byte. The reference for both remains the original Informix 4GL compiler.

One place this backend deliberately does not call the shared runtime is integer addition, which is implemented natively so the compiler can emit a machine `add`. Its fast path is bounded by exactly the rule the shared implementation uses — an Informix `INTEGER` stops being one past nine digits — and that boundary is pinned by its own test. Any future fast path gets the same treatment, because an optimisation that changes an answer is not an optimisation.

14. Where this leaves an estate

Once 4GL is a language on a polyglot runtime rather than a program compiled ahead of time, the interesting question stops being how to leave it and becomes where to keep it. A pricing rule refined over twenty years is not technical debt because of the language it is written in; it is an asset written in a language whose `runtime` became a liability. Separating those two things is what this backend is for.

4GL remains a good language for business processing: exact decimal arithmetic with declared precision and scale, cursors and reports in the language rather than bolted on, and rules a domain expert can still read. What made it a liability was the proprietary runtime, the shrinking hiring market, and a screen model welded to a character terminal. Running it here removes the first, confines the second to the rules themselves — new work is written in Java, JavaScript or Python beside them — and replaces the third with whatever interface the business actually wants.

Neither backend forecloses the other, and the same front end serves both. An estate can translate the programs whose language is the risk, keep the rules whose 4GL is correct, and change its mind later without having lost anything — because in both cases the 4GL remained readable source throughout.

15. Glossary

Term	Meaning
GraalVM	An OpenJDK-based JDK whose optimising compiler is written in Java, with a framework for implementing other languages against it.
JVMCI	JVM Compiler Interface: the standard Java interface through which a compiler written in Java receives bytecode and profile from the VM and installs machine code back into it.
Sea-of-nodes IR	A compiler representation in which data flow and control flow are edges in one graph, so most optimisations are local rewrites.
Truffle	The language-implementation framework: a language is written as a tree of executable nodes, and the framework supplies compilation, deoptimisation, interoperability and instrumentation.
Partial evaluation	Specialising an interpreter against one fixed program so that what remains is compiled code for that program — the first Futamura projection.
Self-specialisation	A node rewriting itself into a narrower form once it has seen what types actually occur, before compilation.
Assumption	A guarded belief that a value is stable, allowing compiled code to omit the check entirely and be invalidated if it changes.
Partial escape analysis	Path-sensitive removal of allocations: an object stays virtual on the paths where it does not escape.
Compilation boundary	A call marked as not to be compiled into the specialised body, keeping a compiled routine compact.
Splitting	Duplicating a call target per call site so each copy can specialise to the arguments that site passes.
OSR	On-stack replacement: compiling a loop while it runs and transferring execution into the compiled version — necessary when a batch enters one loop and stays there.
Deoptimisation	Leaving compiled code mid-method when a speculation fails, reconstructing the interpreter frame, and recompiling without the failed assumption.
Context / engine	A context holds guest state; an engine holds compiled code. A shared context policy lets many contexts reuse one engine's compilations.
Polyglot interop	The message protocol by which languages exchange values on one heap without serialisation.

16. License, Trademarks and Disclaimer

The compiler and the GraalVM backend described in this document form a **proprietary, commercial software product** owned and published by **Airtool Technologies** (airtool.io). **All rights reserved.** It is licensed, not sold, and its use is governed by a commercial license agreement. No right to use, copy, modify, distribute, sublicense, or create derivative works is granted except as expressly permitted in a written license from Airtool Technologies. Unauthorized reproduction or distribution is prohibited. Contact info@airtool.io for licensing terms.

Purpose of this document. This document is provided for information only. It describes the product as at the date of publication, is subject to change without notice, forms no part of any agreement, and creates no representation, warranty or commitment. The terms governing any use of the software are those of the written license agreement between you and Airtool Technologies.

Warranty. Warranties, if any, are those expressly set out in that agreement. Except as expressly so provided, and to the maximum extent permitted by applicable law, the software is provided **"as is"**, and Airtool Technologies disclaims all other warranties and conditions, whether express, implied or statutory, including the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

Limitation of liability. Except as expressly provided in that agreement, and to the maximum extent permitted by applicable law: neither party is liable for indirect, incidental, special, punitive or consequential damages, or for loss of profits, revenue, data or business, however caused; and each party's total aggregate liability is limited as set out in that agreement. Nothing in this document excludes or limits liability for death or personal injury caused by negligence, for damage to tangible property caused by negligence, for fraud or fraudulent misrepresentation, for wilful misconduct, or for any other liability that cannot be excluded by law.

Trademarks — no affiliation. IBM® and **Informix®** are trademarks or registered trademarks of International Business Machines Corporation; Informix is currently developed and distributed by HCL under license. **Java**, **GraalVM** and **Truffle** are trademarks or registered trademarks of Oracle Corporation. This product is **independent** and is **not** affiliated with, sponsored by, or endorsed by any of them. Product names are used solely for identification and interoperability (nominative fair use); no third-party logos are distributed.