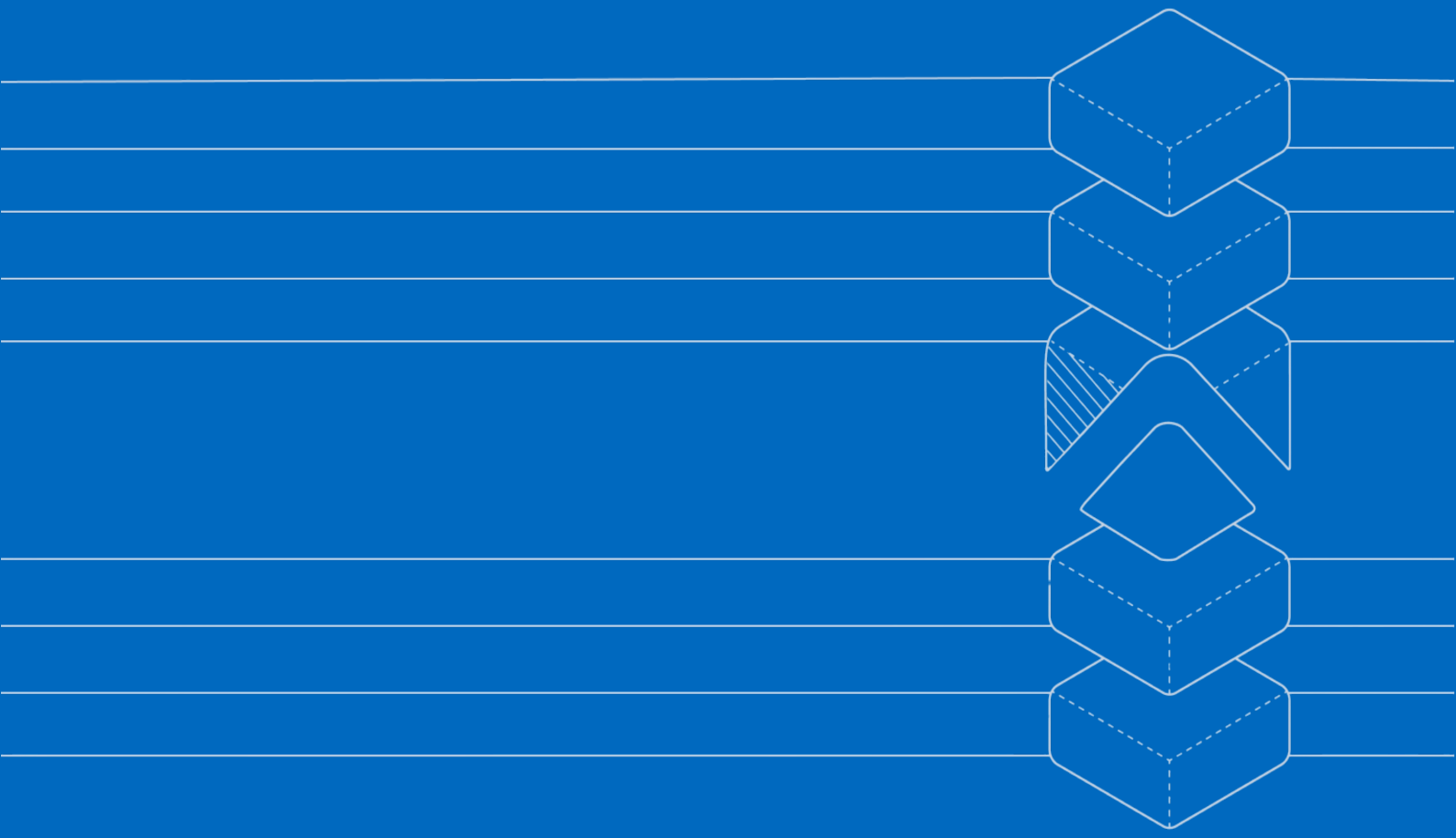

ifxtools
Document Version: 1.0 – 2026-08-13

Informix 4GL to Java

Airtool

Content

This paper describes a **migration compiler** for classic IBM Informix 4GL: a translator that reads `.4gl` and `.per` sources, resolves their types against the live Informix catalogue, and emits Java that a developer is expected to read, review and be able to take over — whenever, and if, the team decides to.

It is written for the enterprise architect who has to decide what happens to an estate of Informix 4GL — typically twenty to forty years of accumulated pricing, credit, despatch and period-close rules, most of which exist in no other written form. It covers the constraint that shapes every design decision in the compiler, the pipeline and its intermediate representation, how an untyped 4GL signature becomes a typed Java method, why decimal arithmetic is the part a naive translation loses, how the terminal surface is reproduced, what the build produces, and how translation fidelity is proven rather than asserted.

> **The governing property: the output is meant to be read.** A translator optimised for producing something that merely `runs` makes different choices at almost every step, and produces a codebase nobody can maintain. This one is built so that the emitted Java is worth owning — reviewable by a developer who has never written a line of 4GL.
>
> **What you then do with it is your decision, and every option stays open.** Keep the 4GL as the source and let the build translate it on every compile, indefinitely. Or take the Java over and maintain it by hand. Or run both for as long as a transition needs — new work in Java, existing rules still authored in 4GL. Or keep the rules in 4GL entirely and run them on a polyglot runtime, which is a companion path rather than a competing one. The compiler does not force a moment of retirement; it makes sure that whichever of these you choose, the Java in front of you is code a team can live with.

1. Motivation — what is actually at risk in a 4GL estate

A typical Informix 4GL application has been in production for decades and still processes the transactions the business runs on. It works. The risk it carries is not correctness — it is that the set of people who can safely change it is shrinking, the toolchain is proprietary, and the knowledge inside it was never written down anywhere else.

That last point is the one that decides how a migration must be run. The authoritative statement of how an order is priced, when a credit limit blocks a despatch, or which customers are exempt from a surcharge is the 4GL source. The specification was lost, the analysts moved on, and the code became the specification. Any approach that asks an organisation to re-derive those rules in another language is asking it to reconstruct something it no longer has, and then to prove the reconstruction behaves identically — which is where these programmes overrun.

A source migration answers that differently: carry the rules across **mechanically**, so nothing is re-derived and there is nothing to reconstruct. The function keeps its name, its parameters, its comments and its position in the module. What changes is the language it is written in, and therefore the market of people who can maintain it.

Informix itself is not the problem and is not moved. The schema stays, the data stays, the stored procedures and triggers stay in the database tier, and the migrated Java reaches them over ordinary JDBC. One variable changes at a time, which is what makes the exercise auditable.

2. Readable output — the constraint behind every other decision

There are two defensible ways to build a 4GL-to-Java translator, and they are not variations of one design.

Design	What the Java is	What follows from it
Opaque output	A build artefact nobody reads	The output may be any shape at all. The team can never take it over, so the 4GL is permanently the only source they can work in — and the option of adopting Java is closed whether or not they ever want it.
Readable output	Code a Java developer can review, and take over whenever they choose	Structure, names, comments and determinism become correctness requirements. Every route stays available: stay in 4GL, move to Java, or run the two side by side.

This compiler is built for the second, and the reason is optionality rather than ideology. An organisation part-way through a migration should be able to change its mind — to keep authoring 4GL for another two years because the people who know the rules are the people who write it, and still have a Java codebase worth adopting on the day that changes. Output nobody can read forecloses that decision at the start, before anyone knows enough to make it.

Four consequences reach into every pass, and they are the properties a reviewer will check.

Structure survives. One `.4gl` module becomes one Java class. One `FUNCTION` becomes one method, in source order. Nothing is merged, split, inlined or reordered, so a reviewer can hold the 4GL and the Java side by side and follow them together. The module name is the single exception: it is capitalised to satisfy Java's class-naming convention, so `credit.4gl` becomes `Credit.java`.

Comments and names survive. 4GL has three comment forms — a leading `--`, a leading `#`, and text inside braces. All three are carried on a hidden token channel rather than discarded by the lexer, and re-emitted in place. Function, variable and parameter names cross over unchanged unless they collide with a Java keyword, and then by a documented, stable rule, because a regenerated file must not churn names.

The runtime is the seam. Generated code stays deliberately thin and calls a runtime library that implements 4GL semantics. Inlining that behaviour to make the output look more idiomatic is precisely what would make twenty thousand lines of generated Java unreviewable.

Output is deterministic. The same input produces the same output, byte for byte: no hash-ordered iteration, no incidental reordering. Without that property a regenerate-and-review workflow drowns in spurious diffs. The one intentional exception is the file header, which records when the file was produced.

3. The pipeline

Java is never emitted straight from the parse tree. Name resolution, `LIKE` resolution against the catalogue, form binding, `GOTO` elimination and report lowering are rewriting passes over a mutable tree, and direct emission collapses under them.

Stage	Input	What it produces
Parse	`.4gl` and `.per` sources	A parse tree, with a `collecting` error listener so a file reports all of its diagnostics rather than only the first
Build IR	Parse tree	A sealed-record intermediate representation, described below
Resolve	IR + the live Informix catalogue	Names bound; `LIKE table.column` and `RECORD LIKE table.*` replaced by concrete types, widths and scales
Bind forms	IR + compiled `.per` forms	Screen fields attached to the variables the dialogs drive
Lower	IR	`GOTO`, `WHENEVER`, dialogs and reports rewritten into forms the emitter can express
Gate	Diagnostics	The build stops here rather than emitting Java from a program it did not fully understand
Emit	IR	Java source, one class per module, plus a program entry point that links the modules

The catalogue dependency in the resolve stage is not incidental and cannot be designed away: Informix resolves column types at compile time, so `DEFINE n LIKE customer.customer_num` has no meaning without the database that defines `customer`. The original `fglpc` requires a reachable server for the same reason. What matters for a migration is that the type a column has in the `database the program will actually read` is the type that reaches the Java — not one captured months earlier from a database that has since moved on.

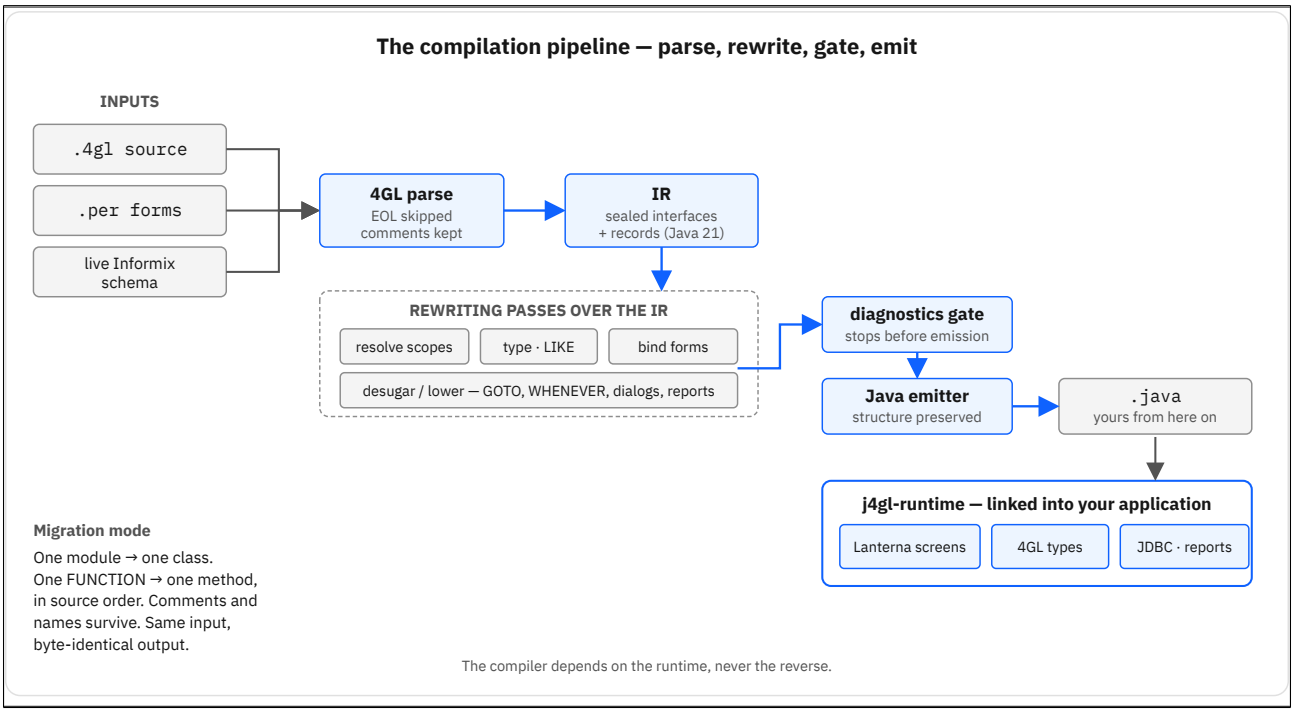

Figure 1 — How a program is compiled — parse, rewrite, gate, emit

3.1. Why an intermediate representation of sealed records

The IR is built from Java sealed interfaces and records, with pattern-matching `switch` in every pass. That buys exhaustiveness checking from the Java compiler itself: when a new statement kind is added to the IR, the build fails in every pass that does not yet handle it, and lists them. Across a language surface as broad as 4GL — dialogs, reports, cursors, windows, exception handling — that property is worth more than any amount of test coverage bolted on afterwards, because it converts an omission into a compile error instead of a silent gap discovered by a customer.

Emission is a hand-written emitter rather than a template engine. A template engine is the right choice when the shape of the output is known in advance, and a translated 4GL program is not that: the emitter must decide, statement by statement, whether a local has to be boxed because a dialog callback captures it, what decimal scale an expression carries, and which of several record shapes a target list refers to. Those decisions are taken from the IR while emitting, and they are clearer as ordinary Java than as template logic pushed into a second language.

4. Recovering types that 4GL never wrote down

4GL is a smaller language than Java, and an honest migration argument starts by conceding it. A 4GL function declares its parameters without types, returns whatever it likes, and says `owed + order_total` where Java must know whether that is integer arithmetic, floating point, or a decimal carrying two places of scale.

A naive translator resolves this by giving up: every parameter becomes `Object`, every return becomes `Object`, and the body fills with casts and unboxing. It runs, and it is unreadable — which under migration mode is a failure, not a trade-off.

So the compiler does the analysis instead. Parameter types are recovered from the `DEFINE` block that follows the signature, which is where 4GL actually states them. Return types are inferred from the `RETURN` statements in the body. Expression types are propagated through the IR so that the scale of a decimal expression is known at the point of assignment. What cannot be inferred is reported rather than guessed.

5. Decimal arithmetic — the part a naive translation loses quietly

Business rules are arithmetic before they are anything else, and 4GL's arithmetic is not the arithmetic most target languages give you by default. `DECIMAL(p,s)` and `MONEY(p,s)` carry a declared precision and scale, and every operation on them is exact within it. The obvious mapping — 4GL `MONEY` to Java `double` — compiles, runs, passes a smoke test, and is wrong.

Consider three lines that every invoice performs: a price, a quantity, a tax rate.

```
DEFINE line_total MONEY(10,2), vat MONEY(10,2), gross MONEY(10,2)

LET line_total = 0.10 * 3          -- three items at ten cents
LET vat        = line_total * 0.21 -- twenty-one per cent
LET gross      = line_total + vat
```

Expression	Mapped to `double`	Translated by this compiler
`line_total`	0.30000000000000004	0.30
`vat`	0.063	0.06
`gross`	0.36300000000000004	0.36
On an invoice	Cents appear and disappear; totals do not reconcile against the ledger	Identical to what the 4GL produced yesterday
When you find out	After go-live, in a period close, with no obvious cause	Before cut-over: arithmetic is compared against the original binaries on real data

The declared type is therefore preserved as a declared type. A `MONEY(10,2)` becomes a runtime decimal object that carries its own precision and scale, rounds where Informix rounds, and refuses to silently widen. The generated Java contains no floating point, no manual rounding call and no scale argument a developer has to remember to pass — the type carries it.

> **This is the single most common defect in machine-translated business code, and the hardest to detect after the fact.** It does not crash, it does not log, and it does not fail a functional test that checks whether the screen displays. It reports a total that is wrong in the last cent, on some rows, some of the time — and it is discovered in a reconciliation weeks later, by which point the migrated system has been writing those values into the ledger.

6. A worked translation: a credit-limit rule

The pair below is not an illustration. The 4GL on the left was written against the Informix demonstration schema; the Java on the right is the compiler's verbatim output for it. The rule is the point — whether an order breaches a customer's credit limit, expressed once, in a function, and written down nowhere else in the business.

```
FUNCTION check_credit_limit(cust_num, order_total, credit_limit)
    DEFINE cust_num INTEGER,
           order_total MONEY(10,2),
           credit_limit MONEY(10,2),
           owed MONEY(10,2)

    -- Would this order put the customer past their credit limit?
    -- The rule lives here and nowhere else: one function, called from
    -- the order entry screen and from the nightly batch alike.

    # What the customer already owes us: every unpaid order, summed.
    # Unpaid means paid_date IS NULL - there is no status column.
    SELECT SUM(i.total_price) INTO owed
        FROM orders o, items i
        WHERE o.customer_num = cust_num
          AND o.order_num = i.order_num
          AND o.paid_date IS NULL

    -- SUM over no rows is NULL, not zero. A customer with no unpaid
    -- orders owes nothing, so the NULL has to become zero here or the
    -- comparison below silently fails.
    IF owed IS NULL THEN
        LET owed = 0
    END IF

    { The rule itself. Note it is >, not >=: landing exactly on the limit is allowed. }
    IF owed + order_total > credit_limit THEN
        RETURN "BLOCKED"
    END IF

    RETURN "OK"
END FUNCTION
```

And the emitted Java:

```
/**
 * Translated from {@code FUNCTION check_credit_limit}.
 *
 * <p>Source: {@code credit.4gl} line 9.
 *
 * @param cust_num 4GL INTEGER
 * @param order_total 4GL MONEY(10,2)
 * @param credit_limit 4GL MONEY(10,2)
 * @return the value of the 4GL RETURN statement, or null if it returns nothing
 */
public String check_credit_limit(Integer cust_num, BigDecimal order_total$in,
                                 BigDecimal credit_limit$in) {
    // DEFINE
    FglDecimal order_total  = FglDecimal.money(10, 2).set(order_total$in);
    FglDecimal credit_limit = FglDecimal.money(10, 2).set(credit_limit$in);
    FglDecimal owed         = FglDecimal.money(10, 2);
    // END DEFINE

    // Would this order put the customer past their credit limit?
    // The rule lives here and nowhere else: one function, called from
    // the order entry screen and from the nightly batch alike.

    // What the customer already owes us: every unpaid order, summed.
    // Unpaid means paid_date IS NULL - there is no status column.
    fgl.query("""
            SELECT SUM(i.total_price)
                    FROM orders o, items i
                    WHERE o.customer_num = ?
                      AND o.order_num = i.order_num
                      AND o.paid_date IS NULL
            """, cust_num).into(owed);
    ...
}
```

Six properties of that output are worth naming, because together they are what migration mode buys.

The signature is typed. `cust_num` is an `Integer` and the two money parameters are decimals with their precision and scale restored — recovered from the `DEFINE` block, which is the only place 4GL states them.

Every comment survives, in place. All three 4GL comment forms appear as Java comments at the same points in the method. On a codebase whose comments are frequently the only surviving documentation, discarding them would destroy more knowledge than the migration transfers.

The provenance is in the file, down to the line. The Javadoc names the 4GL function, the source file and the line number it was declared on — `credit.4gl` line 9 here. That is what makes a side-by-side reading exact rather than approximate: a reviewer holding the 4GL open does not search for the corresponding function, they go to the line. Combined with structure that did not move and comments re-emitted in place, the generated file can be read against the original the way two revisions of the same file are read against each other.

The SQL crosses over intact. The statement is recognisably the statement that was written, with the host variable become a bind parameter. Nobody has to reverse-engineer a query builder to see what runs against the database.

Non-idiomatic choices are deliberate. `MONEY(10,2)` does not become `BigDecimal` at the point of use, and arithmetic, comparison and `NULL` semantics resolve to named runtime calls — because Java does not mean the same things by `+`, `>` and `==` on a money value that may be null. The parameters are received as `BigDecimal` so that ordinary Java can call the method, and immediately bound into decimals that carry the declared scale.

The `NULL` handling is preserved rather than optimised away `SUM` over no rows is `NULL`, not zero. The 4GL author knew that and wrote the guard; the translation keeps it, including the comment explaining why it is there.

7. The terminal surface, reproduced

A 4GL application is not only its logic. It is menus, forms, field-level validation, array input, query-by-example and reports — a character-mode surface described by the language itself, in `OPEN WINDOW` at a row and column, `DISPLAY AT`, and `.per` form files laid out in fixed character cells.

Migration mode requires that surface to keep working. An operator who has driven the same screen for fifteen years should not be retrained as a side effect of a language change, and the business should not have to fund a user-interface project in order to fund a migration. The screens are therefore reproduced, not redesigned: the `.per` forms are compiled with their geometry recovered from the `SCREEN` block, and the interactive dialogs — `MENU`, `INPUT`, `INPUT ARRAY`, `DISPLAY ARRAY`, `CONSTRUCT` — are implemented in the runtime over a terminal library.

The screens below are two moments in one run of the same translated application, captured against a live Informix instance, taken headless and rendered as images. The first is its ring menu — the entry point of a six-module demonstration application. The second is a few keystrokes later: the operator has chosen *Customer*, then *Find-cust*, and typed a surname into the form as the search criterion. The `CONSTRUCT` statement turned that into a `WHERE` clause, the query returned customer 118, and the record is displayed on the compiled `.per` form beneath the *BROWSE* menu the program opens over a result set. Every field on it was read from Informix over JDBC by the translated program, and placed by the geometry recovered from the form source.

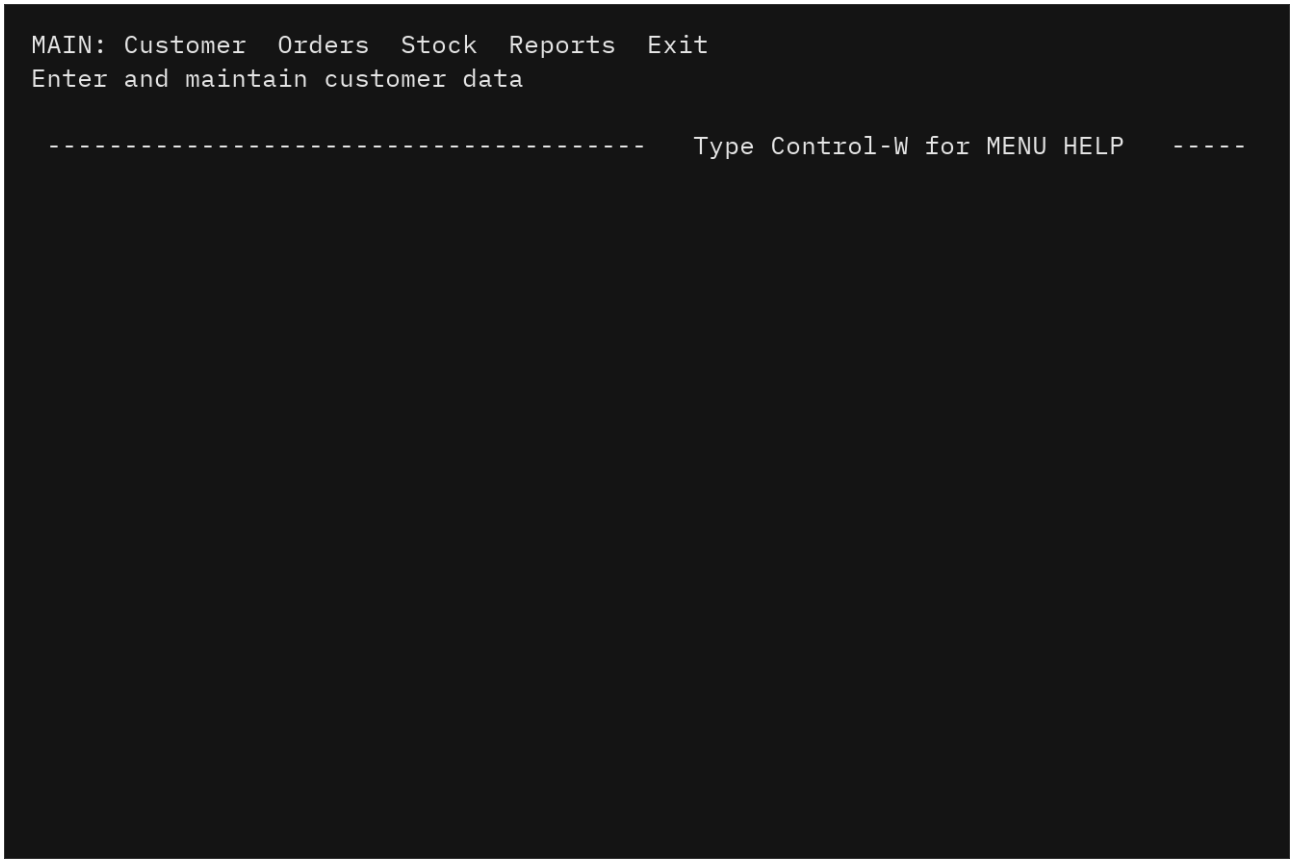

Figure 2 — A translated 4GL ring menu, running on the JVM

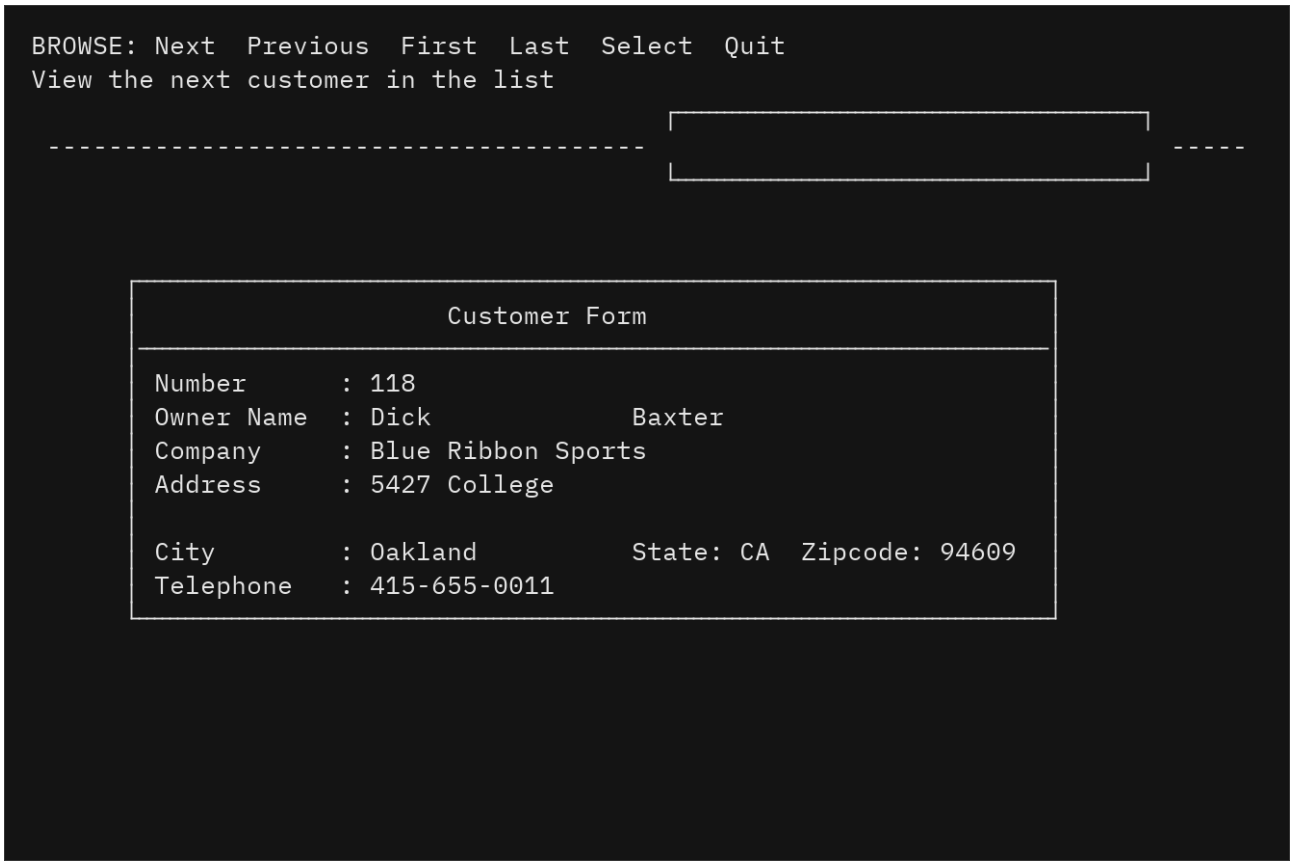

Figure 3 — The same application after a search: a customer on a compiled .per form, under the BROWSE menu

Two details in those captures matter to an architect assessing fidelity. The menu line, the description line under it and the help prompt are placed where 4GL places them, because the runtime implements the `MENU` statement's own layout rules rather than approximating them. And the form's field labels, column positions and box borders come from the compiled `.per` file — the geometry is recovered from the form source, not re-authored by hand during the migration.

> **Reproducing the terminal is a deliberate scope boundary, not a limitation of the compiler.** This is a language migration: Informix 4GL becomes Java, and the application otherwise stays as it is. If what the business wants is a genuinely modern interface, no language migration delivers it, because the terminal assumptions are in the 4GL statements themselves and travel with them into any target language. That is a separate exercise with a separate justification, and it is addressed by running the preserved business logic under a polyglot runtime with a new interface in front of it.

8. What the build produces

For a single program the compiler works the way a 4GL compiler always has: point it at a module containing `MAIN`, and it links what that module needs and produces classes that run. An application, however, is a different object — it has an entry point, modules that must be linked, forms that travel with it, a runtime dependency, a version, and somewhere to be deployed from. Those are the concerns a Maven project exists to hold, so the project is not ceremony on top of the compiler; it is what turns translated modules into an application.

```
$ j4glc project stores          # scaffold a project around existing 4GL
  10 4GL source(s) moved into src/main/4gl

stores/
├── pom.xml                     the translator plugin, the runtime, packaging
├── README.md
└── src/main/4gl/               the source — edit these
    ├── d4_main.4gl             MAIN, the ring menu
    ├── d4_cust.4gl             customer maintenance
    ├── d4_orders.4gl           order entry
    ├── custform.per            the forms, beside the code that opens them
    └── orderform.per

$ cd stores && mvn package

target/
├── generated-sources/j4gl/     the translated Java, registered as a
│   └── fgl/*.java                source root so an IDE indexes it
├── classes/
└── stores-0.1.0-SNAPSHOT.jar   one versioned artifact
```

The translator runs as a build step, so the 4GL under `src/main/4gl` remains the single source while the programme is running: a developer edits 4GL, builds, and the Java in `target/generated-sources/j4gl` is regenerated. Because that directory is registered as a source root, the generated Java is navigable in an IDE and breakpoints set in it are live — which is how a team learns the translated patterns while the migration is still in progress.

For an organisation whose 4GL currently ships as binaries built on one machine by one person, the packaged artifact changes the operational risk profile on its own. It is a versioned jar that deploys the way every other Java artifact in the estate deploys: a repository, a pipeline, a container image, and a version number that can be rolled back.

8.1. Adopting the Java — available from day one, and never obligatory

There is a defined moment at which the 4GL can stop being the source. Copy the generated Java from `target/generated-sources/j4gl` into `src/main/java`, remove the translator plugin from the POM, and the project becomes an ordinary Java project maintained by hand. Nothing else changes: the same runtime dependency, the same artifact, the same deployment. Before that step the compiler is in the loop on every build; after it, the compiler is not required to build, ship or run the application at all.

That step is available from the first day, and it is not a destination. An estate can take it in year one, in year five, for one module, or never — and "never" is a legitimate engineering answer rather than an unfinished migration. 4GL is a smaller, more direct language for the work these programs do: a `DECIMAL` that behaves the way an accountant expects without a library, cursors and reports in the language rather than bolted onto it, and a despatch rule in ten lines that Java needs a page for. A team whose domain experts write 4GL fluently and correctly has a good reason to go on writing it.

What the migration removes is the reason 4GL had become a liability — the proprietary runtime, the toolchain licence, the dependency on a language the market no longer teaches. It does not require giving up the

language itself. Once the build translates on every compile, the 4GL underneath runs on a JVM, deploys as an ordinary artifact and is maintained by whoever knows the rules best.

And the two mix, deliberately. The generated Java is ordinary Java in an ordinary project, so a team can add Java beside it: a REST endpoint over an existing rule, a scheduled job, a message consumer, a library the 4GL never had access to — while the rules themselves stay in 4GL and keep being edited there. New capability arrives in the language best suited to it; the business logic stays in the language that expresses it most directly. That is not a transitional state to be got through. It is a working arrangement, and for many estates it is the right permanent one.

9. The 4GL is still being edited while all of this happens

A migration of any size is not a stop-the-world event. The business keeps asking for changes, and for as long as the migration lasts those changes are made in 4GL — so the 4GL is edited every day of it, by the same people, in whatever editor they already have open. The loop they work in therefore matters as much as the code that comes out the other end: how the language reads on screen, how a mistake is reported, and how long a saved file takes to become a running program.

Three things follow from taking that seriously, and each is worth describing because each was built to answer a specific complaint from people doing the work.

9.1. An editor that knows the language, and knows exactly the language

The distribution carries syntax colouring for the editors an enterprise team is already standardised on. One file serves Eclipse and Visual Studio Code, another serves the JetBrains IDEs, and neither asks the organisation to adopt a new tool to keep maintaining code it has maintained for twenty years.

File	Editor	What it is
`informix4gl.tmLanguage.json`	Eclipse, Visual Studio Code	A TextMate grammar. Eclipse reads it through its TextMate engine (TM4E, included in recent packages); Visual Studio Code reads the same file directly.
`Informix4GL.xml`	IntelliJ IDEA and the other JetBrains IDEs	A file type definition, in the format the IDE keeps its own in.

Both are generated from the compiler's own lexer, not written by hand. That is the whole of the design, and it is the part worth the paragraph. A hand-maintained keyword list is a second, informal statement of what the language is — one that starts slightly wrong and drifts further with every grammar change, and whose errors are invisible because a word that should be coloured and is not looks merely like a word. Deriving the files from the lexer makes the words an editor colours exactly the words the compiler accepts, and a test in the build fails when the grammar gains a keyword the editor files do not know. Highlighting that disagrees with the compiler is worse than no highlighting, because it teaches the wrong language to whoever is learning the estate.

What is derived is the vocabulary; what is a judgement is which of four groups a word reads as. The partition is required to be total and disjoint, which turns "somebody added a keyword to the grammar" into a failing test rather than a word that quietly stays black on screen. Four groups and no more, because the JetBrains file-type format offers exactly four keyword lists — TextMate could carry more, but two different classifications of one language would drift the way the keyword list did.

Group	Count	Reads as
Statements, control flow and the word-operators	131	`MAIN` , `IF` , `FOREACH` , `DISPLAY` , `WHENEVER`
Types and the type-forming words	40	`CHAR` , `DECIMAL` , `DATETIME` , `RECORD` , `LIKE`
Embedded SQL	74	`SELECT` , `WHERE` , `DECLARE` , `COMMIT`
Built-ins, display attributes, colours and constants	51	`TODAY` , `LENGTH` , `REVERSE` , `RED` , `INT_FLAG`

296 keywords in total, plus comments in all three of the language's forms, both kinds of string literal with their escapes, numbers, and the name in a `FUNCTION` or `REPORT` header. A few words are honestly two things at once — `SET` is a statement and a SQL clause, `LIKE` both declares a variable from a column and matches a string — and each is coloured as whichever a reader meets more often. A word in the wrong group is a cosmetic bug; a word in no group is a build failure.

> **Colouring is not checking.** Neither file parses 4GL. They colour it, which is a lexical judgement made a token at a time, and they will happily colour a program that is not valid. The question "is this program correct" has exactly one answer worth having, and it is what the compiler reports — the same front end that produces the translation. The JetBrains format carries a second, smaller limit of its own: it allows one line-comment prefix, so `#` is declared and `--` comments are not coloured.

9.2. A syntax error on the line that caused it

The compiler knows the file, the line and the column of every syntax error it finds. For a while it flattened them into a message string, and the consequence was that a mistake in a 4GL program arrived in the IDE as a stack trace attached to the project's build file, with a page of parser expectations and nothing to click. Every piece of information needed to place the error correctly existed; none of it reached the one component with somewhere to put it.

It is now carried structurally through to the build, which means a syntax error is reported the way a Java syntax error is: as a marker on the statement, in the editor, in the file that contains it. The estate is edited with the same feedback the rest of the organisation's code has had for fifteen years.

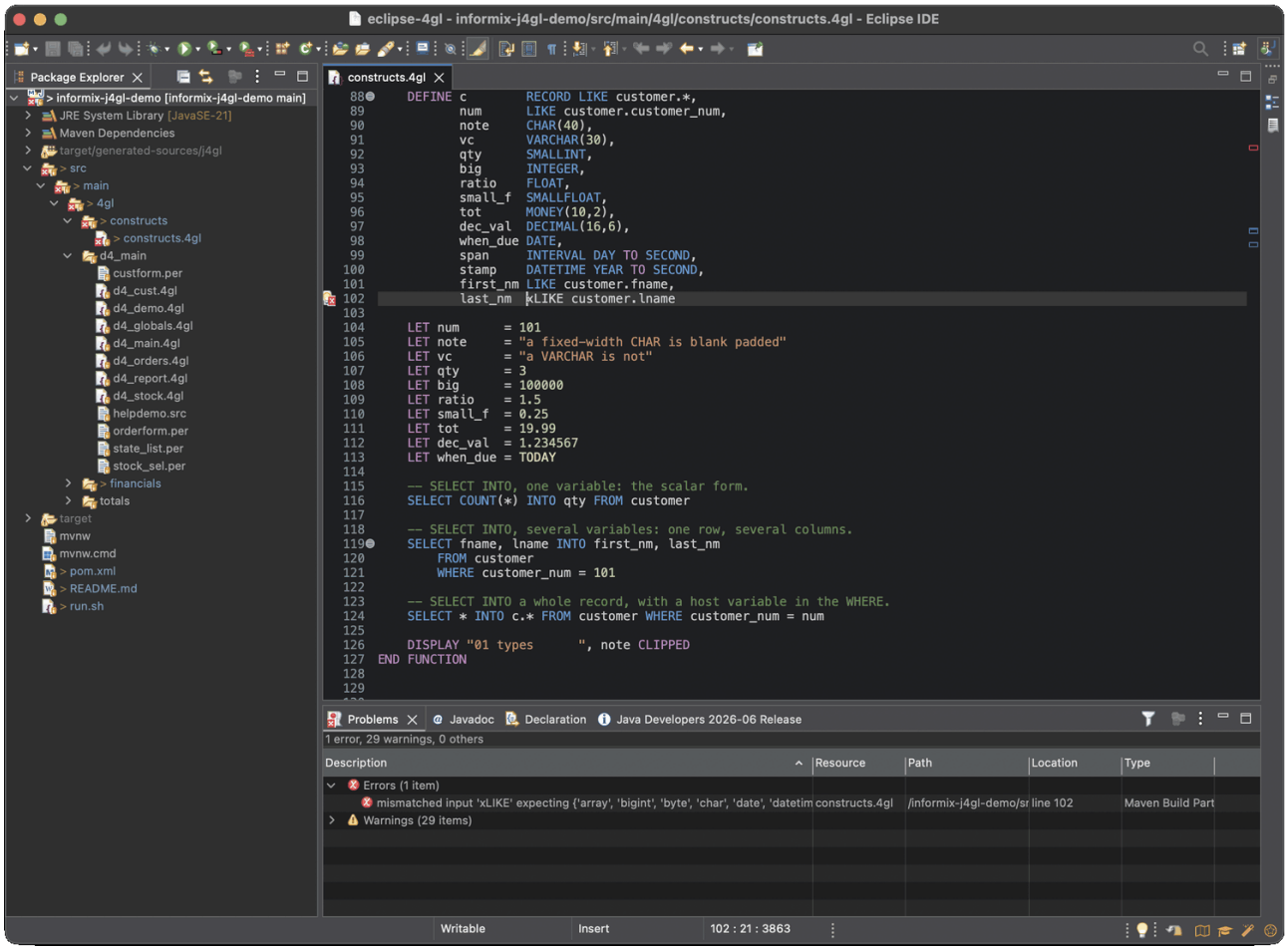

Figure 4 — A 4GL syntax error in Eclipse — on the line, in the Problems view

The capture is one screen of an ordinary working session, and four things in it are the subject of this chapter. The marker sits in the ruler beside line 102, where a deliberate typo has turned `LIKE` into `xLIKE` in a declaration whose neighbour on line 101 is correct. The Problems view carries the parser's own message — the token it did not expect and the set it did — against the resource `constructs.4gl` and the location `line 102`, so it is a row to double-click rather than a trace to read. Its Type column says the report came from the Maven build participant, which is the point: this is the project's real build speaking through the IDE's problem model, not a second tool with a second opinion. And the error is counted beside everything else the project has to say about itself — one error, twenty-nine warnings — rather than in a place of its own.

The colouring described above is doing its work in the same window: types, embedded SQL, statements and comments each in their own group, on a `.4gl` file sitting under `src/main/4gl` in a conventional Maven layout, with the translated Java in `target/generated-sources` beside it. To the developer, and to the build, 4GL is simply another source language in the project.

Placing a marker is the easy half. Removing it is the half that is usually wrong, and the rule is worth stating because it explains behaviour a user would otherwise have to guess at:

What the build does	What happens to the markers
Translates	Clears what its last run reported, then reports what it finds now — so a fixed error disappears and an unfixed one stays.
Decides it is up to date	Clears nothing. Nothing has changed, so everything already reported is still true.

There is a policy question underneath, and it has two right answers depending on where the estate is. For a finished application, one program that will not compile should stop the build: a program that does not compile is not a program. For an estate in the middle of a migration — which is the state every migration starts in — the opposite is right, and the build can be told so. The programs that translate are translated, each one that does not reports against the line that stopped it, and the work continues on the rest while that program is dealt with.

9.3. What a build does when nothing has changed

The translation runs on every build, before the Java compiler, and the first thing it does is decide whether to translate at all. That decision is the difference between an IDE that responds to a saved file and one that appears to hang on it, so it is worth being precise about what is asked — and about what the answer is `not`.

Four questions, in order. The first that answers "yes, work is needed" ends the sequence.

Question	Why it is asked
Has a program been deleted?	A source that is gone must take its generated Java with it, and timestamps cannot see this — removing a program leaves every surviving file untouched. Generated code for a program that no longer exists is dead code that looks live, which during a migration is the expensive kind.
Does every program have output, or a recorded reason for having none?	The existence of an output directory is not enough. An empty one used to mean "up to date" indefinitely, over a program that had never been generated at all, and the symptom surfaced layers away as a missing entry point.
Did the editor say a `.4gl` file changed?	Asked only inside an IDE, which rebuilds a whole project when one file is saved. The workspace already knows what was touched, so asking it turns "translate on every save" into "translate when a 4GL file was saved".
Is any source newer than the oldest generated file?	The command-line equivalent, and the question that decides a build outside an IDE. Form specifications count as sources here, not only programs.

> **The unit of decision is the source tree, not the individual program.** When the answer is that work is needed, every program under that source directory is translated again — not only the one that changed. This is a property of the language rather than a shortcut: a 4GL program is linked from its call graph, and a module reached through `GLOBALS` can belong to several programs at once, so determining which programs a single edited module belongs to means parsing all of them. The analysis that would avoid the work costs the same as the work. What the build reliably avoids is translating when `nothing` has changed, which is the case that dominates a working day.

One consequence of the second question is worth following, because it is the kind of detail that decides whether a feature is usable in practice. A program that fails to translate produces no Java, so "every program has output" can never become true while it fails — and a source tree containing one such program would therefore be fully translated on every build, for ever. Allowing a broken program to be reported rather than fatal would, on its own, have traded a stopped build for a build that repeats all of its work every time: a worse deal, and not an obvious one. So the failures are recorded, which lets them count as accounted-for. The record

is rewritten on every build, including when it is empty — otherwise a program that started translating would be counted broken indefinitely, and the build would stop noticing the very change that fixed it.

The database catalogue is the other cost in the loop, because translation resolves declarations made against table columns. It is read **once per database per build**, not once per program: a source tree of forty programs naming one database opens one connection and reads one catalogue. Database names are compared without regard to case, because Informix does not distinguish them and two programs naming the same database in different case name one database — an estate with both `PIA` and `pia` in its sources used to read the same catalogue twice. And a build that decides no translation is needed does not read the catalogue at all, which is what makes a no-op build independent of whether the server is reachable.

> **The catalogue can be held for a whole editing session, and it is off by default.** With it on, saving a file never contacts the server. With it off — the default — the catalogue is read afresh by each translating build, so an `ALTER TABLE` is picked up by the next build with nothing to remember and nothing to invalidate. Turning it on means a schema change is not seen until the IDE is restarted or the project cleaned. That is a promise about the state of a running server, and it belongs to whoever runs that server rather than to a build tool that would have to make it silently.

9.4. The rest of the loop, which arrives with the language rather than from us

Colouring and a marker on the line are the modest part of the story, and it is worth being explicit about where the rest comes from — because it is a genuine architectural difference rather than a feature comparison.

This compiler does not ship a 4GL debugger, and there is no plan to build one. An environment whose answer is to keep the program in 4GL has to: if 4GL is what executes, then someone has to write a debugger for 4GL, a profiler for 4GL, a coverage tool for 4GL, and keep all of them current. Each is a tool the vendor maintains alone, understood by the people who use that product and nobody else.

Translating the program to Java answers the same question by not asking it. What executes is Java, so what debugs it is the debugger the organisation already has, and the same is true of every other tool in the loop:

What the team gets	What it replaces
Breakpoints and stepping, into the runtime call that implements a 4GL construct and back up the stack to the statement that caused it	A `DISPLAY` statement and a rerun
Conditional and exception breakpoints — stop when a customer number matches, on the two-hundredth iteration, or the moment a particular exception is constructed anywhere in the program	A day of narrowing by bisection
Watch expressions and live evaluation, including calling a translated function to see what it returns for the case in front of you	Recompiling with more `DISPLAY` statements
Hot code replace — change a method body and continue the session rather than restarting and re-reaching the state through ten screens	Restarting
Sampling profilers and heap analysis, the ones already used across the estate	Inference from timings

What the team gets	What it replaces
Unit tests, coverage, static analysis and CI, on the infrastructure the organisation already runs	Nothing comparable

None of that was built for this compiler. It arrives because the program is Java, which is also why it will still be there in ten years, maintained by people who have never heard of 4GL. That is the difference between a tool an organisation depends on a vendor for and a tool the market maintains on its behalf.

The obvious objection is that a debugger stepping generated Java is stepping the wrong text. Two things answer it. The generated code is written to be read — that is the property this paper opens with — and every generated routine carries a Javadoc naming the 4GL function it came from and the source file and line it was declared on, with the original comments preserved in place beside the statements they explained. A breakpoint in Java is therefore locatable in the 4GL. And for the many teams who adopt the Java as the source at some point, the objection dissolves entirely: the text under the debugger becomes the text they maintain.

None of this is visible in the translated Java, which is the point. It is what a team notices in the first week and stops noticing in the second — and the difference between a migration that people work inside comfortably and one they work around.

10. SQL, cursors and the shape of embedded statements

Embedded SQL is where a 4GL program spends most of its wall-clock time and where a translation most easily changes behaviour without changing output — so the treatment is deliberately conservative: the statement that reaches Informix is the statement the 4GL contained.

10.1. Host variables become bind parameters, in position

A 4GL statement references program variables directly inside the SQL text. Those references are replaced by parameter markers, and the values are bound in the order the markers appear. The component that performs that placement is shared with the GraalVM backend, precisely because parameter placement is easy to get subtly different twice — and a transposed pair of parameters is a defect that produces plausible rows rather than an error.

Two properties follow, and both matter to a DBA. The statement text is stable across executions, so the engine's statement cache behaves as it did before. And nothing in the translation concatenates a value into SQL text, so a class of injection risk that hand-written migrations reintroduce is absent by construction.

10.2. SELECT … INTO, and the singleton rule

`SELECT … INTO` in 4GL is a singleton select: it expects at most one row, and returning more is an error the program can observe through its status. That behaviour is preserved rather than relaxed into "take the first row", because a program that silently took the first row of an unexpectedly ambiguous result would produce a different answer from the original on exactly the data that revealed the ambiguity.

Selecting into a record — `INTO p_customer.*` — binds positionally against the record's members, whose names, order, widths and scales came from the catalogue at compile time. This is why `LIKE` resolution against a stale catalogue is treated as a defect rather than an inconvenience: a record built in a different order puts the city in the postcode, and nothing raises an error.

10.3. Cursors, FOREACH and transactions

Declared cursors, `FOREACH`, and hand-managed `OPEN` / `FETCH` / `CLOSE` map onto JDBC result sets with their 4GL semantics intact — including `NOTFOUND` as the status that ends a loop, and including the fact that a cursor holds its position across other statements on the same connection.

Transaction statements — `BEGIN WORK`, `COMMIT WORK`, `ROLLBACK WORK` — are issued on the same connection, so the database's own transaction and locking behaviour is unchanged. A migrated program holds locks for exactly as long as the original did, because it issues the same statements in the same order. Isolation level and lock-wait remain properties of the database and the connection rather than decisions the translation makes.

`WHENEVER ERROR` and the status variable are preserved as control flow rather than being converted into Java exceptions at the point of use. A 4GL program that checks `status` after a statement continues to work the way it was written; one that declared `WHENEVER ERROR STOP` — or declared nothing, since stopping on error is the language's default — continues to stop.

11. Forms and reports — the two subsystems that are not code

Two parts of a 4GL application are not procedural code at all, and a migration that treats them as code produces the worst of the output. Both are compiled rather than transliterated.

11.1. The .per form is a canvas, and the canvas is the specification

A form file is a character-cell drawing of a screen plus a set of declarations binding its fields to columns. Its geometry is not decoration: field positions, widths and the tab order derived from them are what operators have in muscle memory after fifteen years.

Below is `customer.per` from the Informix tutorial — the form behind the screenshot earlier in this paper. The `SCREEN` block is the canvas; a bracketed tag such as `[f001 ]` is a field, and its width is the width of the bracket. The `ATTRIBUTES` section binds each tag to a column, and `SCREEN RECORD` declares a record over a run of them.

```
DATABASE stores

SCREEN
{
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                         CUSTOMER  FORM

       Number:  [f000       ]

   First Name:  [f001           ]     Last Name:  [f002           ]

      Company:  [f003               ]

      Address:  [f004               ]
                [f005               ]

         City:  [f006           ]

        State:  [a0]   Zipcode:  [f007 ]

    Telephone:  [f008              ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
}
END

TABLES
customer

ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0   = customer.state;
f007 = customer.zipcode;
f008 = customer.phone;
END

INSTRUCTIONS
SCREEN RECORD sc_cust (customer.fname THRU customer.phone)
END
```

And this is the compiler's output for it — a static factory returning a form, written against a fluent builder in the runtime. It is verbatim, including the Javadoc that records which `.per` it came from.

```
/** Compiled from {@code customer.per}. */
private static Form customer() {
    return FormBuilder.named("customer")
            .database("stores")
            .table("customer")
            .canvas("""
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                         CUSTOMER  FORM

       Number:  [f000       ]

   First Name:  [f001           ]     Last Name:  [f002           ]

      Company:  [f003               ]

      Address:  [f004               ]
                [f005               ]

         City:  [f006           ]

        State:  [a0]   Zipcode:  [f007 ]

    Telephone:  [f008              ]
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
""")
            .field("f000").bound("customer", "customer_num").add()
            .field("f001").bound("customer", "fname").add()
            .field("f002").bound("customer", "lname").add()
            .field("f003").bound("customer", "company").add()
            .field("f004").bound("customer", "address1").add()
            .field("f005").bound("customer", "address2").add()
            .field("f006").bound("customer", "city").add()
            .field("a0").bound("customer", "state").add()
            .field("f007").bound("customer", "zipcode").add()
            .field("f008").bound("customer", "phone").add()
            .build();
}
```

The canvas survives as a canvas. This is the decision that matters, and it is worth dwelling on because the alternative is so tempting. A compiler could compute each field's row and column and emit a list of coordinates — and the output would then be unreviewable, because nobody can look at `at(9, 17).width(11)` and see a screen. Instead the drawing is carried across into a Java text block, unchanged, and the builder scans it to recover geometry. A developer reading the generated Java sees the form.

It is a normal Java API, not a private format. `FormBuilder` is part of the runtime and documented for direct use, so a team that later wants a screen the original estate never had can write one — or compute one, since coordinates remain available for a caller that would rather place fields than draw them. Whichever is written by hand wins over what the picture said. The migration therefore leaves behind a form layer that can be extended by hand rather than only regenerated.

Binding is by name, resolved against the catalogue. `f001` becomes `customer.fname`, and the field's type and width come from that column. This is the same `LIKE` resolution the rest of the compiler performs, which is why a form and the record it populates cannot disagree about what a column is.

Forms are emitted into the program's entry point, one static factory each, and registered by name at start-up. `OPEN FORM cust_form FROM "customer"` therefore resolves without reading a file at run time: the form travels inside the artifact as code, which is what allows the packaged jar to carry an application's screens with it.

11.2. Reports are a language within the language

4GL's report writer is declarative: a `REPORT` block declares sections — `FIRST PAGE HEADER`, `PAGE HEADER`, `PAGE TRAILER`, `BEFORE GROUP OF`, `ON EVERY ROW`, `AFTER GROUP OF`, `ON LAST ROW` — and the runtime drives them as rows arrive, maintaining page geometry, group breaks and aggregates.

Emitting that as ordinary imperative Java would produce something no reviewer could follow, so it is lowered into a report definition the runtime executes with the same engine both backends use. Margins, page length, pagination, `SKIP` and `NEED`, `WORDWRAP`, column alignment and printer output are runtime behaviour rather than generated code.

This has a useful consequence for verification, described below: a report is deterministic text that both the original toolchain and the migrated program produce without a terminal, which makes it the ideal medium for byte-for-byte comparison.

12. What a reviewer should look for in the output

Migration mode is only meaningful if someone checks it, so this is the checklist an architect can apply to a sample of generated code from their own estate before committing to a programme.

Check	What passing looks like	Why it matters
Structure	One module, one class. One `FUNCTION`, one method, in source order.	You can diff the Java against the 4GL and follow both.
Names	Function, variable and parameter names unchanged.	Institutional vocabulary survives; a domain expert can still read the code.
Comments	All three 4GL comment forms present, in position.	On most estates the comments are the only surviving documentation.
Types	Typed signatures — no sea of `Object`.	Untyped output is unreadable and unmaintainable regardless of correctness.
Decimals	No `double` anywhere near money. Declared precision and scale visible.	The defect that reconciles wrong months after go-live.
SQL	The statement recognisable, with bind parameters.	A DBA can review what reaches the engine without reading Java.
Provenance	Javadoc naming the 4GL function, file and line.	Every method answers "where did this come from".
Determinism	Regenerating twice produces identical files.	Without it, review drowns in spurious diffs.

13. The DATABASE statement, at compile time and at run time

`DATABASE stores` is the first line of most 4GL modules, and it does two quite different jobs at two different moments. Conflating them is the single most common source of confusion when an architect first looks at a migrated build, so they are worth separating precisely.

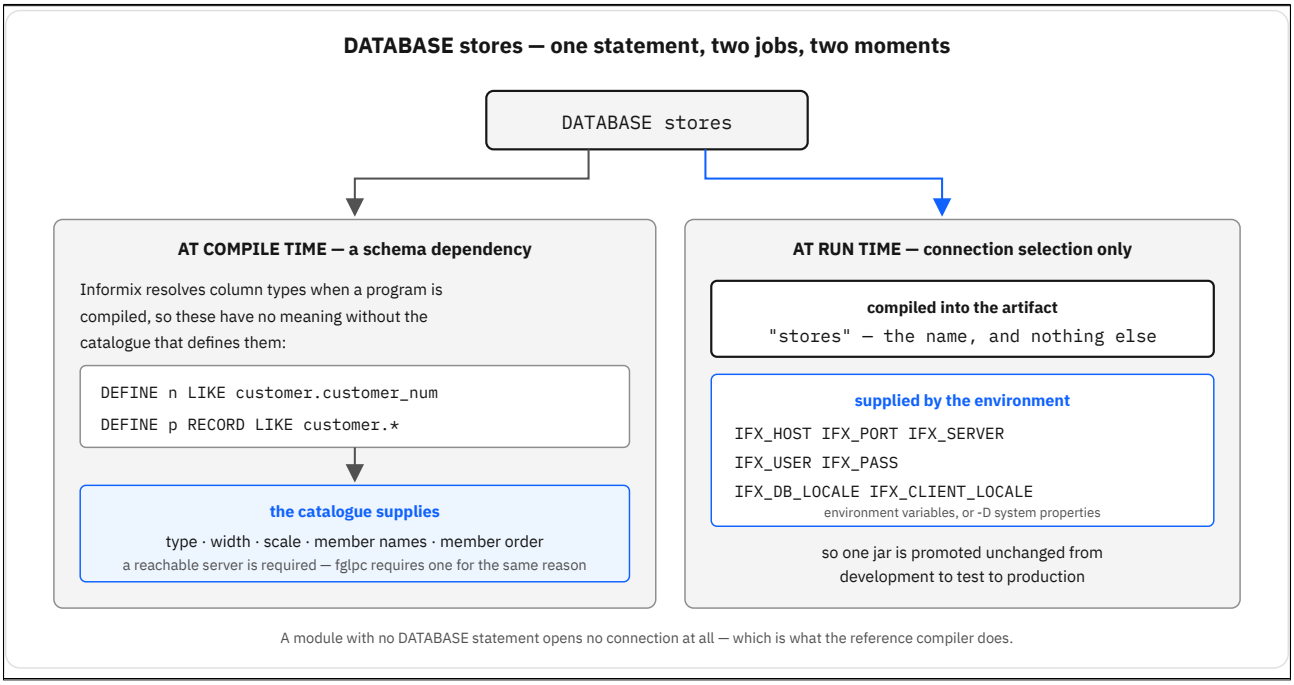

Figure 5 — DATABASE stores — one statement, two jobs, two moments

13.1. At compile time it is a schema dependency

Informix resolves column types when the program is compiled, not when it runs. A `DEFINE n LIKE customer.customer_num` has no meaning without the catalogue that defines `customer`, and neither does `DEFINE p RECORD LIKE customer.*`, whose member names, order, widths and scales all come from the database. This compiler needs a reachable server for the same reason the original toolchain did: the `DATABASE` statement names the catalogue it must read.

Two consequences follow. A build needs the database the program declares to be reachable — the same requirement the estate already lives with. And the type a column has in `that` catalogue is the type that reaches the Java, which is why resolving against a stale copy is a defect rather than a convenience: a column silently becomes the wrong Java type and the error surfaces as a truncation or a rounding difference much later.

The database a module was compiled against is recorded in the generated file's header comment, so the provenance of a translated class is visible without consulting a build log.

13.2. At run time it selects the connection, and nothing else

The declared database name is compiled into the generated entry point as a constant. At start-up the program opens `that` database if connection settings are present in its environment, and otherwise runs without a connection. A module with no `DATABASE` statement opens nothing at all — which is what the reference compiler does, and what a program that never touches SQL expects.

What is **not** compiled in is equally important: no host, no port, no server name, no user and no password. Only the database's own name travels inside the artifact. The same jar therefore runs against development, test and production by changing its environment, which is the property a deployment pipeline requires.

> **Multi-module programs.** A 4GL application is normally several modules linked into one program, and the `DATABASE` statement belongs to the module that declares it. Linking resolves this the way the original toolchain does — the program takes the database of the module that carries `MAIN`, and the other modules' `LIKE` references were already resolved, at compile time, against the catalogue each of them declared.

14. Driving a screen program without a person at the terminal

A 4GL application assumes a human. `MENU` waits for a keystroke, `INPUT` waits for a field, `PROMPT` waits for an answer, and none of it proceeds without a terminal. That assumption is why most 4GL estates have no automated tests of their screen programs, no way to load data through the paths a clerk uses, and no evidence of what a screen showed on a given day beyond somebody's memory.

The generated application takes two arguments that remove the assumption. `--headless` runs on an in-memory terminal and prints the screen; `--keys` supplies the keystrokes. Both are emitted into the entry point of every translated program, so this is a property of the artifact you deploy rather than a facility of a development tool.

```
# Select Customer, then Find, then accept the query (ESC), unattended
java -jar target/stores-0.1.0-SNAPSHOT.jar --headless --keys $'CF\x1b'
```

The important property is `where` the keystrokes go. They are queued into the same input path a user's keyboard feeds, so the dialog runs exactly as it runs for a person: field by field, with every `BEFORE FIELD` and `AFTER FIELD` block firing, every validation applied, every message displayed. Nothing is bypassed and nothing is simulated — which is what makes the results mean anything.

14.1. Automated data entry, through the rules rather than around them

This is the case with real commercial weight. In a mature 4GL estate a substantial part of the business logic lives `inside the dialogs` : the check that fires when a field is left, the lookup that fills three others, the rule that refuses a despatch when the credit check comes back short. None of it is in the database, and much of it is in no other program.

So when the business needs to load data — a bulk import, an acquisition's customer file, a nightly feed from an upstream system — there are normally two options and both are bad. Write `INSERT` statements and you bypass every rule that lives in the screen, which is how a load ends up creating records the application itself would have rejected. Or re-implement the validation in the loader, which means maintaining a second copy of rules that were never written down once.

Driving the screen is the third option, and it is the only one where the data goes in through the same path a clerk would use. The loader supplies keystrokes rather than SQL; the program validates, looks up, calculates and refuses exactly as it does at a desk. A record that a human could not have entered is not entered.

14.2. Regression testing that a 4GL estate could not previously have

A scripted dialog is a repeatable dialog, and a printed screen is an assertable result. That is enough to put screen programs under test in continuous integration — for most estates, for the first time.

It is also how this compiler proves itself: the parity harness runs the same programs through the original Informix toolchain and through the generated Java, driving both with the same keystrokes, and compares what they printed. The mechanism a customer would use for their own acceptance tests is the mechanism the compiler is verified with, which is a reasonable thing to ask of a tool before trusting it.

The practical shape during a migration is a golden-screen suite: capture what the original produced for a set of scripted sessions, run the same scripts against the translated application, and compare. Differences are found by a build rather than by a user, and they are found in the week the change was made.

14.3. And in JUnit, in the same suite as the rest of your Java

The command line is the coarse form. Underneath it the runtime publishes a small API for driving a program without a terminal, which is what turns a migrated screen into something an ordinary unit test can exercise — no subprocess, no separate harness, and no reason for 4GL-derived code to sit outside the test suite the rest of the estate already runs.

It is not a separate product and there is nothing to install. These methods are public API in the same runtime library the migrated application already depends on to execute — the one that provides the 4GL types, the screen and the SQL session. A scaffolded project has them on its compile classpath from the first build, and a customer adds only whichever test framework they already use.

Three calls are all a test needs.

Call	What it does
`FglScreen.headless()`	An 80×24 screen in memory. No terminal is opened.
`screen.onRefresh(...)`	Called with the screen's rows every time the program repaints. Collecting them gives everything the program displayed during the run — which matters because a dialog usually clears the screen when it ends.
`screen.typeAhead(...)` / `typeLine(...)`	Queue keystrokes, exactly as a keyboard would deliver them. `typeLine` adds the Enter.
`FglProgram.builder()`	Builds the `environment` a 4GL program runs in — the screen, the database connection, the program arguments. Despite the name it is not your program ; it is what your program is given.

One name is worth pausing on, because it reads backwards. `FglProgram` is the `runtime` — the screen, the connection and the arguments that a 4GL program is handed. **Your** program is the entry class the compiler generated from your `MAIN` : `OrdersmenuMain` below, from `ordersmenu.4gl` . It takes the runtime as its constructor argument, and `run()` executes what `MAIN` did. The generated launcher does exactly this — the test is not a special path.

So a test is four steps: build the runtime with an in-memory screen, queue the keystrokes a user would type, construct and run your translated program, and assert on what appeared. Both examples below start with the 4GL, because that is the thing under test.

A menu. Three commands, one of which leaves:

```
MAIN
    DEFINE chosen CHAR(20)

    MENU "MAIN"
        COMMAND "Customer" "Enter and maintain customer data"
            LET chosen = "customer"
            DISPLAY "chose: ", chosen CLIPPED AT 5, 1
        COMMAND "Orders" "Enter and maintain orders"
            LET chosen = "orders"
            DISPLAY "chose: ", chosen CLIPPED AT 5, 1
        COMMAND "Exit" "Leave the program"
            EXIT MENU
    END MENU
END MAIN
```

Translated, this becomes the entry class `OrdersmenuMain`, and the test drives it:

```
@Test
@DisplayName("pressing an option's initial letter runs that command")
void initialLetterSelects() throws Exception {
    // Every row the program ever displays, in order.
    List<String> displayed = new ArrayList<>();

    FglScreen screen = FglScreen.headless();
    screen.onRefresh(displayed::addAll);          // one call per repaint

    // The runtime: a screen, and no database because this program needs none.
    try (FglProgram fgl = FglProgram.builder().screen(screen).build()) {
        screen.typeAhead("OE");                   // Orders, then Exit
        new OrdersmenuMain(fgl).run();           // your program, from ordersmenu.4gl
    }

    assertThat(displayed).anyMatch(row -> row.contains("chose: orders"));
}
```

A rule behind a screen. The same shape, now with a database. This is `ordercheck.4gl`: a menu that collects two values and calls `check_credit_limit` — the function shown verbatim earlier in this paper, in `credit.4gl`, linked into the same program.

```
DATABASE stores

MAIN
    DEFINE cust_num     INTEGER,
           order_total  MONEY(10,2),
           verdict      CHAR(7)

    MENU "ORDERS"
        COMMAND "Check" "Check an order against the customer credit limit"
            PROMPT "Customer number: " FOR cust_num
            PROMPT "Order total: "     FOR order_total

            LET verdict = check_credit_limit(cust_num, order_total, 3000.00)

            DISPLAY "Verdict: ", verdict AT 5, 1
        COMMAND "Exit" "Leave"
            EXIT MENU
    END MENU
END MAIN
```

Compiled, it links into the entry class `OrdercheckMain`. Driven from the command line it behaves as a person would — `C` selects Check, the two answers go to the prompts, `E` leaves. Both of these are real runs against the demonstration database:

```
$ java -jar target/ordercheck.jar --headless --keys $'C104\n999999.00\nE'
Verdict: BLOCKED

$ java -jar target/ordercheck.jar --headless --keys $'C104\n25.00\nE'
Verdict: OK
```

And as a test, asserting instead of reading:

```
@Test
@DisplayName("an order past the credit limit is refused at the screen")
void overTheLimitIsBlocked() throws Exception {
    List<String> displayed = new ArrayList<>();

    FglScreen screen = FglScreen.headless();
    screen.onRefresh(displayed::addAll);

    // The runtime: a screen and a connection, which this program does need.
    try (FglProgram fgl = FglProgram.builder()
            .screen(screen)
            .connection(testDatabase())           // your fixture, your transaction
            .build()) {
        screen.typeAhead("C");                    // Check
        screen.typeLine("104");                   // customer number
        screen.typeLine("999999.00");             // an order past the limit
        screen.typeAhead("E");                    // Exit
        new OrdercheckMain(fgl).run();           // your program, from ordercheck.4gl
    }

    assertThat(displayed).anyMatch(row -> row.contains("Verdict: BLOCKED"));
}
```

A session, and every branch of it. Both tests above drive one path through one operation, which makes the model look narrower than it is. Keystrokes are queued `ahead` of the run, the program executes to completion, and the assertions are made afterwards on the ordered record of what it displayed. A test can therefore drive a whole session — and a suite of them can cover the branches a program actually has.

The last example is the shape that matters commercially: a decision with more than one answer. It is `orderdecision.4gl`, printed here in full so that every keystroke in the test below can be traced to the statement that consumes it. First the dialog — the whole of the program's input surface:

```
DATABASE stores

MAIN
    DEFINE cust_num    INTEGER,
           qty         INTEGER,
           unit_price  MONEY(10,2)

    MENU "ORDERS"
        COMMAND "Price" "Price an order and decide whether it needs approval"
            PROMPT "Customer number: " FOR cust_num
            PROMPT "Quantity: "        FOR qty
            PROMPT "Unit price: "      FOR unit_price

            CALL decide(cust_num, qty, unit_price)

        COMMAND "Exit" "Leave"
            EXIT MENU
    END MENU
END MAIN
```

Two commands, and 4GL selects one by its initial letter — so `P` runs Price and `E` leaves. Choosing Price runs three `PROMPT` statements in written order, each reading one line into one variable. There is no other way into this program: five keystroke groups, in that sequence, and it has everything it needs.

Then the two decisions, in the order the dialog calls them. The band is arithmetic on the quantity; the approval needs the database, because it depends on what the customer already owes:

```
FUNCTION decide(cust_num, qty, unit_price)
    DEFINE cust_num   INTEGER,
           qty        INTEGER,
           unit_price MONEY(10,2),
           band       CHAR(6),
           rate       DECIMAL(5,4),
           net        MONEY(10,2),
           total      MONEY(10,2)

    --   The volume band. Ordinary business rule, no database:
    --   the boundaries are inclusive and the order matters.
    CASE
        WHEN qty >= 100
            LET band = "BULK"
            LET rate = 0.1500
        WHEN qty >= 25
            LET band = "VOLUME"
            LET rate = 0.0750
        WHEN qty >= 10
            LET band = "TRADE"
            LET rate = 0.0250
        OTHERWISE
            LET band = "LIST"
            LET rate = 0.0000
    END CASE

    LET net   = qty * unit_price
    LET total = net - (net * rate)

    CALL approve(cust_num, band, total)
END FUNCTION
```

```
FUNCTION approve(cust_num, band, total)
    DEFINE cust_num  INTEGER,
           band      CHAR(6),
           total     MONEY(10,2),
           cust_name CHAR(15),
           owed      MONEY(10,2),
           verdict   CHAR(8)

    --   Who the customer number refers to. Reading the name back is
    --   what lets the operator check the number he just typed.
    SELECT lname INTO cust_name
        FROM customer
        WHERE customer_num = cust_num

    --   What that customer already owes: every unpaid order, summed.
    SELECT SUM(i.total_price) INTO owed
        FROM orders o, items i
        WHERE o.customer_num = cust_num
          AND o.order_num = i.order_num
          AND o.paid_date IS NULL

    --   SUM over no rows is NULL, not zero.
    IF owed IS NULL THEN
        LET owed = 0
    END IF

    IF owed + total > 3000.00 THEN
        LET verdict = "REFERRED"
    ELSE
        LET verdict = "APPROVED"
    END IF

    DISPLAY "customer ", cust_num USING "<<<<&", " ", cust_name CLIPPED AT 5, 1
    DISPLAY "band ", band CLIPPED, "  total ", total, "  owed ", owed AT 6, 1
    DISPLAY "verdict ", verdict CLIPPED AT 7, 1
END FUNCTION
```

`104` is nothing more than a customer number in the demonstration database that ships with Informix. The program reads the surname back for exactly the reason an operator needs it read back — to confirm the number he typed is the customer he meant — and that makes the identifier visible on screen, so a test can assert on it rather than assume it.

Translated and run with the dialog scripted, twenty-five units at ten dollars produces this. It is a real run against a live instance, not a mock-up:

```
$ java -jar target/orderdecision.jar --headless --keys $'P104\n25\n10.00\nE'

ORDERS: Price  Exit
Price an order and decide whether it needs approval

customer 104 Higgins
band VOLUME  total       $231.25  owed         $0.00
verdict APPROVED
```

Every keystroke in that script has one destination, and the test that follows sends the same five:

Keystroke	What consumes it	Result
`P`	the `MENU`	runs `COMMAND "Price"`

Keystroke	What consumes it	Result
`104` then Enter	1st `PROMPT`	`cust_num = 104`
`25` then Enter	2nd `PROMPT`	`qty = 25`
`10.00` then Enter	3rd `PROMPT`	`unit_price = $10.00`
`E`	the `MENU`, on return	`EXIT MENU`, program ends

Only the middle two vary across the branches, so the test is one method with a table of cases. Each row is a complete session — menu, three prompts, decision, exit — driven exactly as an operator would drive it:

```
@ParameterizedTest(name = "{0} units at {1} -> {2} {3}, {4}")
@CsvSource({
    //  qty   price     band     total     verdict
    "     5,  10.00,    LIST,    $50.00,   APPROVED",
    "    10,  10.00,    TRADE,   $97.50,   APPROVED",
    "    25,  10.00,    VOLUME,  $231.25,  APPROVED",
    "   100,  10.00,    BULK,    $850.00,  APPROVED",
    "   100, 100.00,    BULK,    $8500.00, REFERRED"
})
void everyBandAndBothDecisions(String qty, String price, String band,
                               String total, String verdict) throws Exception {
    List<String> displayed = new ArrayList<>();

    FglScreen screen = FglScreen.headless();
    screen.onRefresh(displayed::addAll);

    try (FglProgram fgl = FglProgram.builder()
            .screen(screen)
            .connection(testDatabase())
            .build()) {
        screen.typeAhead("P");             // MENU  -> COMMAND "Price"
        screen.typeLine("104");            // PROMPT "Customer number: "
        screen.typeLine(qty);              // PROMPT "Quantity: "
        screen.typeLine(price);            // PROMPT "Unit price: "
        screen.typeAhead("E");             // MENU  -> COMMAND "Exit"

        new OrderdecisionMain(fgl).run();  // your program, from orderdecision.4gl
    }

    //   The customer the number resolves to, and the decision made about him.
    assertThat(displayed).anyMatch(row -> row.contains("customer 104 Higgins"));
    assertThat(displayed).anyMatch(row -> row.contains("band " + band)
                                       && row.contains(total));
    assertThat(displayed).anyMatch(row -> row.contains("verdict " + verdict));
}
```

That suite is worth more than its five assertions. It pins the band boundaries — the reason the quantities are 5, 10, 25 and 100 is that those are the values either side of which the answer changes, and a translation that turned `>=` into `>` would fail two rows. It pins the arithmetic, because the totals are exact to the cent and a decimal mapped to floating point would not produce `$231.25`. It pins the customer lookup, because the surname on screen has to be the one that number resolves to. And it pins the independence of the two decisions: the last row differs from the fourth only in unit price, which is enough to change the verdict while leaving the band alone.

What a test cannot do is stop halfway and choose what to type next from what it sees; the script is fixed before the run begins. In exchange the run is deterministic and repeatable, which is what makes it usable in a build.

All three programs in this section — the menu, the credit check and this one — are published with the paper, with the commands to compile and run them and the output each should produce.

Three things follow that are worth more than they first appear.

The rule is exercised where it lives. The second test does not check a re-implementation of the credit rule; it drives the dialog that calls it, so the field validation, the prompt parsing and the rule are covered together — the path a clerk takes.

The test is ordinary Java. It runs under the build the organisation already has, reports where its other tests report, and can be written by a Java developer who has never read 4GL. A team can therefore start writing tests for a 4GL application before anyone on it knows 4GL, which is often the real constraint.

It makes a migration reviewable statement by statement. Characterisation tests written against the original behaviour become the specification the translation is held to — and because they drive screens rather than call functions, they cover exactly the parts that were never covered before.

14.4. Why this matters at board level

The preceding pages are engineering. The consequence is not, and it is worth stating in the terms an executive committee weighs, because it is one of the larger returns in a migration of this kind and it is easily overlooked beside the licence saving.

A historical point, made without criticism. Informix 4GL was designed in the mid-1980s and matured through the 1990s, and it was very good at what it was asked to do: put transactional business logic close to the data, on the hardware of the day, with a productivity no general-purpose language of the period matched. What did not exist then was the practice that has since become ordinary — a suite of automated tests, run on every change, by a machine. The frameworks that made that normal arrived at the end of that decade and the discipline spread through the 2000s. Systems written before it were built to a standard of care that was entirely reasonable at the time, and they were verified the way everything was verified then: by a person, at a terminal, working through a checklist.

What that costs an organisation today is not the testing, it is the caution. When the only way to know whether a change broke something is to have somebody exercise the screens, three things follow. Change becomes expensive, so less of it is attempted. Changes are batched into large releases, which makes each release riskier and the next one later. And the people who can judge whether a change is safe are the few who remember how the system behaves — so knowledge concentrates rather than spreads, and every departure takes some of it away. None of this is a failure of the team. It is the arithmetic of testing by hand.

The migration changes the arithmetic. Once the application is a Java program and its screens can be driven programmatically, the estate's core business logic can be put under automated test for the first time — using the same frameworks, the same build servers and the same practices the organisation already applies to everything written this decade. Not a parallel toolchain for the legacy system, and not a testing product to buy: the suite the rest of engineering already runs, extended to cover the application the business actually depends on.

Four consequences follow, and they compound.

What changes	Why it matters commercially
Regression risk moves from people to machines	A change is checked in minutes, on every commit, against the behaviour the business relies on — rather than in a test window, by whoever is available, against a checklist written from memory.
Release cadence stops being governed by fear	Smaller, more frequent releases become defensible, because each is verified. The cost of a change falls, which means changes the business wanted but could not justify become affordable.

What changes	Why it matters commercially
Knowledge leaves people and enters the repository	A characterisation test is a written statement of what the system does, in a form that fails when it stops being true. It is the documentation these estates were always missing, and it cannot go stale silently.
The hiring constraint eases at both ends	Tests are ordinary Java, so a developer who has never seen 4GL can write them — and can then change the application with evidence rather than folklore. New people become productive on the legacy system rather than only on the new work.

There is a governance dimension as well, and in regulated sectors it is often the argument that carries. An automated suite produces evidence: a record, per release, that the pricing, credit and approval rules behaved as specified, generated by the build rather than attested by a person. Auditors are accustomed to asking for that of modern systems and accustomed to being told it does not exist for the older ones.

None of it requires the 4GL to be abandoned. Tests written this way exercise the rules wherever they live — in 4GL that the build translates, in Java the team has adopted, or in both at once — so the testing discipline can be established first and the language decisions taken later, on their own merits. For most organisations that is the right order: make the system safe to change, then decide what to change.

14.5. Capture, for evidence as well as documentation

Because the screen buffer is printed rather than drawn, a run leaves a record. Every frame is retained and `--frame` selects one — negative counts back from the end, and `--frame 0` lists what was rendered, which matters because a program that ends in `CLEAR SCREEN` or returns to a menu leaves the interesting screen in the middle.

The screenshots in this paper were produced that way, from programs run against a live Informix instance. The same mechanism serves an audit trail — what the operator saw when the limit was overridden — and documentation that cannot drift from the software, because it is generated from a run rather than drawn by hand.

14.6. What it is not, and where the limits are

It is not an integration API. When the goal is to call a rule from another system, the better answer is to call the rule: the translated Java exposes it as a method, and the GraalVM backend exposes it as a routine inside the server. Driving a screen is right when the logic exists `only` inside the dialog, which in these estates is often, but it should be a deliberate choice rather than the default.

A script is a contract with the dialog. Keys are consumed in order, so a screen that gains a field breaks the scripts that drive it — the same brittleness every user-interface automation has. Scripts belong under version control beside the program they drive.

Running out of keys accepts. End of input is treated as acceptance, so an incomplete script does not hang — it completes the dialog with what it has. That is the right behaviour for an unattended run and a trap for an author who expected a failure, so a test should assert on the screen rather than on the exit status alone.

An automated run is a real run. The writes are real writes in a real transaction against the database named in the environment. Everything said earlier about which database a program opens applies with more force when nobody is watching it.

15. The entry point — what a 4GL MAIN becomes

4GL has no `main`. It has a `MAIN` block in one module, and a runtime that arranges everything around it: the screen, the database, the program arguments, the exit status. Reproducing that arrangement is the job of a generated entry class, and it is worth describing in full because it is the piece an operations team actually runs.

The entry class is emitted **separately from the module that holds `MAIN`**, and that separation is deliberate. It carries no translated logic — only wiring, the connection and the column catalogue, which for a six-module program is over a hundred lines the compiler invented. Keeping it out means every `.java` named after a `.4gl` contains that module's code and nothing else, which is what makes a side-by-side review possible.

15.1. Arguments: the program's, and the launcher's

A 4GL program reads its arguments with `NUM_ARGS()` and `ARG_VAL(n)`, and `fglgo` passes everything after the compiled program straight through. A generated `main` has its own options to keep apart from the program's, so the two are separated by a bare `--`, following the getopt convention.

```
# Options for the launcher, then the program's own arguments
java -jar target/stores-0.1.0-SNAPSHOT.jar --headless -- 2026-08-31 EUR
```

Everything after `--` becomes the program's argument vector: `NUM_ARGS()` answers 2, `ARG_VAL(1)` is `2026-08-31`. The launcher's own options are recognised before the separator.

Option	Effect
`--headless`	Render to memory and print the screen at the end instead of driving a terminal. Assumed automatically when there is no console, because a piped or IDE-launched run cannot open a terminal and failing with a terminal error would be worse than rendering.
`--keys`	Feed keystrokes to a dialog, with `\n` for Enter. This is what makes an interactive program scriptable, and therefore comparable against the original toolchain — the parity harness drives the generated entry point exactly this way.
`--`	Everything after it belongs to the 4GL program.

`EXIT PROGRAM n` becomes the process exit status, so a batch scheduler sees what it saw before.

15.2. Nothing is read from disk that was not compiled in

This is the property that makes the artifact deployable, and it is worth being explicit because 4GL practice is the opposite.

A 4GL installation ships compiled forms and message files alongside the program, and the runtime finds them by path at execution time. Here they are **not** deployment artefacts. Forms are emitted as Java — one static factory each, as shown earlier — and registered by name when the program starts, so `OPEN FORM ... FROM "customer"` resolves in memory. The column catalogue is read from `syscolumns` `while the compiler was connected` and baked into the entry class, so no type is looked up at run time.

The consequence for an operator is that the jar is the deployment. There is no forms directory to ship beside it, no message-file path to configure, no 4GL installation on the target host and no licence server to reach. The `.per` files remain in the repository because they are source — edited during development, recompiled by the build — and they take no part in the execution path.

16. Connecting to Informix

Connection settings come from the environment, or equivalently from Java system properties, so that one artifact is promoted unchanged between environments. Only the database `name` is compiled in.

Setting	Meaning	Default
`IFX_HOST`	Host running the Informix instance	required
`IFX_PORT`	Listener port	`9088`
`IFX_SERVER`	The instance name — `DBSERVERNAME`	required
`IFX_USER`	Database user	required
`IFX_PASS`	Password, plain or `OBF:`	see below
`IFX_PASS_COMMAND`	A command whose first line of output is the password	see below
`IFX_DB_LOCALE`	Database locale	discovered
`IFX_CLIENT_LOCALE`	Client locale	discovered

```
export IFX_HOST=informix.internal
export IFX_SERVER=ol_informix
export IFX_USER=appuser
export IFX_PASS=...
java -jar target/stores-0.1.0-SNAPSHOT.jar
```

A login timeout is set before connecting, because an unreachable or wedged server would otherwise block start-up with no output at all — indistinguishable, to whoever is watching, from a hang.

16.1. Locales are discovered, not assumed

This is the setting most likely to stop a migration on its first day, and it now resolves itself.

The locale pair used to default to `en_US.819`, which is a working default only for the databases that happen to be ISO 8859-1. A UTF-8 database **refuses the connection outright** with `"Database locale information mismatch"` — at run time, and equally at compile time, since resolving column types needs the same connection. An estate whose databases are not all one encoding would meet that error immediately, and the message does not obviously point at a client setting.

So both locales are now left unset and resolved **per database, from the server**. The database's own locale is read from `sysmaster` once per database per process and used for the connection. When the server cannot say, no locale properties are sent at all — deliberately, so that the connection fails with the server's own message naming the real problem, rather than a guess reporting a mismatch against a database that may not even exist.

Setting `IFX_CLIENT_LOCALE` turns discovery off and passes what you gave straight to the driver. That is the escape hatch for a database the server reports wrongly, or for reproducing a specific client's behaviour — and it means an explicit setting is never quietly overridden by a lookup. The same applies to `IFX_DB_LOCALE`.

16.2. Where the password comes from

`IFX_PASS` holds it directly. If it is unset, `IFX_PASS_COMMAND` is run and the first line of its output is taken as the password. An explicit `IFX_PASS` is never second-guessed.

The command is a hole, not an integration. It is whatever prints the secret on standard output, and the compiler and runtime never learn where it came from — which is exactly the property that makes it work with a store this project has never heard of.

```
export IFX_PASS_COMMAND='security find-generic-password -w -s informix'
export IFX_PASS_COMMAND='pass show informix/prod'
export IFX_PASS_COMMAND='vault read -field=password secret/informix'
export IFX_PASS_COMMAND='op read op://prod/informix/password'
export IFX_PASS_COMMAND='aws secretsmanager get-secret-value --secret-id informix \
                         --query SecretString --output text'
```

This is the only arrangement in which the secret never exists at rest in the application's world: no ciphertext in a config file, no key sitting beside it, nothing to encrypt, because nothing is stored. Rotation, audit and revocation belong to the store, which is where they were always going to be done properly. Standard error from the command is passed through untouched, because that is where a store explains itself — `"could not be found in the keychain"` — and without it an operator would get an exit code and nothing else. Standard output stays piped, because that is the secret.

Its limit is honest and worth stating: anyone who can run that command as your user gets the password. On a laptop the keychain prompts; in continuous integration it does not. That is the same limit every option on this ladder has — it is simply moved somewhere better staffed.

Either source may carry an `OBF:` value instead of a plain one. That is **not** a security boundary — anyone with the jar can reverse it — but it stops a password being legible in a config file or a `git diff`, which is the leak that actually happens. A store may hold the obfuscated form too, so a configuration that moves from `IFX_PASS` to a password manager does not have to be un-obfuscated first.

The connection's own `toString` never includes the password, so it cannot reach a log or an exception message by accident.

16.3. What the artifact does not need

No 4GL runtime on the target host, no licence server to reach, no environment variable pointing at a 4GL installation, no forms or message files beside the jar, and no compiler present at run time. The jar and a JVM are the deployment. That is the practical difference between a translated application and a recompiled one, and it shows up on the first host the artifact is copied to.

17. Asking the compiler about the estate

An estate of 4GL is the part of a migration nobody can see. How much of it can be read, what a module calls, what nothing reaches, which programs touch a table, what breaks if a column changes, what a column becomes in Java — every one of those has an exact answer this compiler already computes on its way to producing your build, and until recently the only way to get one was to open files.

The compiler answers them over the **Model Context Protocol**, so that an organisation's own assistant asks the compiler instead of guessing from the text. It is a fourth module built the same way as the alternative backend: it depends on the compiler, and nothing depends on it, so a migrated application never sees it. The protocol is an open standard rather than a private interface — any assistant that speaks it connects to the same tools and receives the same answers, so nothing here rests on a single AI vendor.

> **The governing property: the assistant does not answer the question.** It decides which tool answers it, calls that tool, and reads the result back. Every tool is a thin wrapper over a compiler entry point this build already tests. The consequence is the one that matters: the assistant's answer and your build cannot disagree, because the same code produces both.

17.1. What it is, and what it is not

The word `server` misleads in every respect that matters to whoever owns the infrastructure, so it is worth stating plainly what this is not:

- **There is no service to run.** No daemon, no port, no URL, nothing to start at boot, nothing to monitor.
- **There is nothing listening.** The assistant starts it as a child process when a conversation needs it and talks to it over a pipe.
- **It stops when the conversation stops.** Between conversations nothing of it runs.
- **It sends the source nowhere.** The tools read files on the machine they run on. What travels is the answer, and only as far as the assistant the organisation has chosen.

It ships in the distribution beside the compiler and installs with the same command, so the client configuration names a command on `PATH` and nothing else:

```
{"mcpServers": {"j4gl": {"command": "/usr/local/bin/j4gl-mcp"}}}
```

17.2. Every tool is a wrapper, and that is the whole design

Each tool delegates to an entry point the build already covers: the parser for diagnostics, the catalogue reader for column types, the translator for Java, the headless driver for a running screen.

> **A tool that computes something itself is a tool that can disagree with the compiler** — and then there are two compilers to keep in step, one of which nobody is testing. That is the single rule this module is held to, and it is what makes the answers admissible: the number an assistant reports is the number the build would produce.

The rule is enforced structurally too. Six tools had each grown a private copy of the same loop — collect the files, parse each, skip what failed, remember which — and the duplication was not the problem. Drift was,

because each copy decided for itself what to do with a file it could not read. **A tool that quietly omits what it could not parse reports "nothing calls this" with the same confidence it reports a fact.** That loop is now written once, so every tool names what it could not read instead of privately dropping it.

An MCP server is normally the one program nobody can try by hand: no command line, started by somebody else's application, speaking a protocol down a pipe. This one also runs from a shell, using the `same` tool definitions the protocol path uses, and a test asserts the two return byte-identical answers:

```
$ j4gl-mcp --tools                              # what there is, and each argument
$ j4gl-mcp estate_survey path=/opt/estate/src
$ j4gl-mcp who_calls path=/opt/estate/src function=ring_menu
$ j4gl-mcp describe_table database=stores_demo table=customer
```

For an evaluation this is the property to test first: any figure an assistant reports can be reproduced from a shell, by a person, with no model in the loop at all.

17.3. The tool catalogue

Tool	Answers	Database	Writes
`estate_survey`	how much of a tree parses, the failures grouped by cause, its size in lines, and which character sets its files are in	no	no
`outline`	one file's `DATABASE`, `MAIN`, functions, reports, calls and tables	no	no
`who_calls`	every call site of a function, and where it is defined	no	no
`where_used`	every place a table or column is named in SQL	no	no
`call_graph`	what calls what, what nothing reaches, what is defined twice, what no 4GL defines, where the graph loops	no	no
`schema_impact`	what breaks if a column changes	no	no
`sql_inventory`	every database statement, and the defects that are not syntax errors	no	no
`complexity`	which routines are hard, so a migration can be ordered	no	no
`globals_usage`	who reads and who writes each global	no	no
`review_program`	one ranked list to act on, over a program or a whole tree	no	no
`describe_table`	a table's columns, each with its 4GL type and its Java type	yes	no
`translate`	the generated Java, plus what did not translate	yes	yes
`compiles_check`	whether the generated Java actually compiles	yes	yes
`run`	the program, headless, with scripted keystrokes	yes	yes

The tools needing a database need one for the reason set out in `Recovering types that 4GL never wrote down`: Informix resolves column types at compile time, so an answer without a catalogue is not one that skipped a step, it is one that could not have been right.

17.4. What the analysis makes knowable

The figures below describe **one financials application** — not a company's whole codebase. A module is one `.4gl` source file; a program is the group of modules that share an entry point. Each answer returns in about five seconds over the whole application.

Question	On this application	Why it decides something
How large is it?	227,936 lines across 613 modules	The figure every owner quotes, and the one the rest are read against
How big is the call graph?	3,528 routines, 17,612 calls	The unit of work in any migration plan
What does no 4GL define?	83 names	The native surface a migration must supply — the classic late cost
What does nothing reach?	298 routines	Maintained and migrated for nothing unless found first
Can it be split up?	245 of 388 globals written from more than one module	Why that application cannot be lifted apart a piece at a time

A survey is a migration plan. The parse gate is all-or-nothing, so one construct the front end cannot read costs a whole program. Grouping the failures by cause turns a number nobody can act on into a queue — on one reference tree, forty of the fifty-one unreadable files were the same single construct. That is not a report of a problem; it is a day's work, identified.

Asking the tree beats searching the text, in both directions. For one function in the demonstration corpus, `grep -rn 'CALL ring_menu'` finds 22 lines and `who_calls` finds 21 — and the tool is the one that is right. Six of its hits are `call ring_menu ()`, lower case with a space, which 4GL accepts and that grep misses; seven of grep's hits are in a file that is not 4GL source at all.

Two findings here were wrong before they were right, both caught by reading the output against the source rather than trusting it. `SELECT COUNT(*)` parses as a function call — syntactically that is what it is — so `COUNT` joined the list of names no 4GL defines, which is the list a migration estimate is drawn from and where a phantom entry is a phantom cost. And a cursor opened in one routine and closed in another is ordinary 4GL; a routine-local check called it a leak. Cursors closed anywhere in the program are now gathered before anything is judged, and the two cases are reported as what they are: closed nowhere is a defect, closed elsewhere is a pair that has to move together.

17.5. The dependency that is invisible in the text

`schema_impact` is the tool to keep, because the dependency that breaks a 4GL shop is not written down anywhere:

```
DEFINE r RECORD LIKE customer.*
```

That declaration names no column, and yet every column of `customer` belongs to it. Change one column's width and this module changes shape — while a text search for that column will never mention it. So the two classes are reported separately: the modules that name the **table**, which change shape, first; then the

modules that name the **column**. Those references come from the compiler's own reader of `LIKE` declarations — the same one that resolves those types during translation — rather than from a search written for the tool.

17.6. One ranked list, banded by what a finding costs

Each analysis tool answers one question well, and an estate produces more findings than a reader can act on. `review_program` runs the measuring tools, keeps what is worth acting on, and returns a single list — **banded rather than scored**.

Band	Means
Blocking	The compiler cannot read it, so nothing downstream is true
Structural	It decides how the migration is organised
Housekeeping	Real, and safe to leave

A single number ranking a parse failure against a deep routine invents a unit neither has and hides which of the two put a file at the top. It measures nothing of its own, reading each tool's answer through the result that tool returns, so it cannot disagree with them.

> **The cap is per kind, and that was learned the hard way.** A flat cap of 300 findings looked reasonable and was not: the shared globals and duplicated names filled every place before the deep-routine findings were reached, so a review of an application containing hundreds of routines too deep to review by reading reported none of them — the most useful structural finding, deleted by an arithmetic detail, with nothing to say it had happened. Capping per kind means a class can be shortened and never disappear.

17.7. Encoding: the defect that ships silently

A 4GL estate written before this century is frequently not UTF-8. It is ISO-8859-1, because that is what the terminals, editors and operating systems of the period used. On the reference application, well over a third of the modules are Latin-1 and the rest are UTF-8 — an estate that does not agree with itself, which is the normal condition rather than an unlucky one.

Read as though it were UTF-8, every byte that is not valid UTF-8 becomes a replacement character. Dozens of those sit inside quoted string literals, which is the text a user reads. The replacement character is then written faithfully into the generated Java, and nothing fails: the program compiles, runs, passes a smoke test, and displays rubbish where the original displayed a customer's name.

> **This is the defect class that decides whether a migration can be trusted.** A missing feature announces itself; a silent mistranslation does not, and it is found by a customer rather than by a test.

The compiler decodes strictly as UTF-8 and falls back to ISO-8859-1, which cannot fail and cannot lose a byte — UTF-8 is self-validating, so this needs no configuration and no guess. It is deliberately not the platform default: since JDK 18 that is UTF-8 on every machine whatever the locale, so it describes the machine rather than the file, and a compiler decoding by it would translate the same source into two different programs on

two different hosts. The decoder lives in the runtime module because the form compiler needs it too — a form's labels are the text a user reads, so mojibake there is visible on every screen.

Decoding correctly is only half of it, and the missing half is the one worth having. **A fallback nobody is told about is a guess dressed as a fact.** `estate_survey` counts what each file actually is and reports when an estate disagrees with itself, and `review_program` raises a mixed encoding as a structural finding — because every module still parses, so nothing else in the answer would hint at it.

The same work produced the line counts, and their discipline is worth stating: they are reported and never ranked on. A thousand lines of `DISPLAY` statements is a long read and an easy migration; forty lines nested five deep is neither. They exist so a reader can convert between the unit every estate is quoted in and the units that predict the work. Comment lines are separated rather than dropped, because in an old estate they are a large and uneven share and a quote drawn from the wrong total is wrong by that much.

17.8. Where this sits in the edit, build and test loop

The sections above on editing, building and regression testing describe a loop this completes at both ends. Before a change, `who_calls`, `where_used`, `schema_impact` and `globals_usage` give its blast radius across the whole estate rather than across the files someone thought to open — the step a person cannot do reliably over millions of lines, and the one whose absence makes changes feel dangerous.

After it, `compiles_check` answers whether the generated Java actually builds, which `translate` does not claim: a translation that returns Java, reports nothing outstanding and then fails `javac` looks like success everywhere except where it matters. It executes nothing, so it is safe against a program that writes to the database, which makes it the check to put in front of every commit. `run` and the JUnit harness cover behaviour, and `review_program` reports what the change introduced.

> **The assistant drives this loop; it does not underwrite it.** Every step is a command a build server runs unattended, so a team that never connects an assistant gets the same guarantees. Nothing an assistant proposes is accepted on trust — the compiler and the runtime check it exactly as they check a person's work.

17.9. What the tools may change, and the limits

All but two of the tools only read. `translate` writes generated Java, and `run` executes a program against the database that program declares — and a 4GL program can `INSERT`. MCP clients ask for approval before each call, so an execution happens because a person allowed it. For an evaluation, point the connection at a copy; the read-only analysis tools need no database at all.

- **Not a resolved semantic model.** `who_calls` and `where_used` read one file at a time with no link step: an alias is not followed back to its table, and a name is matched as written.
- **Not silent about what it could not read.** A file that fails to parse is listed as unsearched rather than counted as clean, because "no uses" over an estate where some files never parsed is a different answer from "no uses".
- **Line totals cover what parsed.** A file the compiler cannot read is not one whose size it can vouch for, so the totals exclude it.
- **Not a replacement for the migration.** These tools describe an estate and translate parts of it. What to do about it remains an engineering and commercial judgement.

18. Proving fidelity rather than asserting it

The principal risk in any source migration is not that the translator crashes. It is that the translation runs, produces plausible output, and disagrees with the original somewhere nobody looks. A compiler that claims a percentage of automation is answering the wrong question; the question an architect should ask is `how do you know the translated program computes what the original computed`.

The method is differential testing against the original toolchain. A corpus of 4GL programs — arithmetic, cursors, dates and intervals, string handling, `NULL` semantics, pattern operators, defaults, records, arrays, reports — is compiled twice: once by the original Informix 4GL compiler, and once by this one. Both are run against the same database, and their output is compared byte for byte. A difference is a defect, and it is a defect stated in the only currency that matters: the original binaries disagree with the translation on real data.

Reports are a particularly useful medium for this, because a report is deterministic textual output that both implementations produce without a terminal — margins, pagination, group headers, aggregates and column alignment all land in a file that can be compared exactly.

Interactive programs are compared the same way, through the scripted-dialog mechanism described earlier: both compilations of a program are driven headless with the same keystrokes, and the screens they print are compared frame by frame. Between reports and screens, the comparison covers both surfaces on which a translated program could disagree with the original — what it computes, and what it shows.

The architecture strengthens the method. The front end that parses, types and links the 4GL is shared with a second backend — the GraalVM runtime that executes 4GL directly, the subject of the companion paper — and both backends call the same runtime library for decimals, SQL and reports. Two implementations of a language are two answers to the question of what it means, and both are held to the same reference: the original Informix toolchain, on the same programs, over the same data.

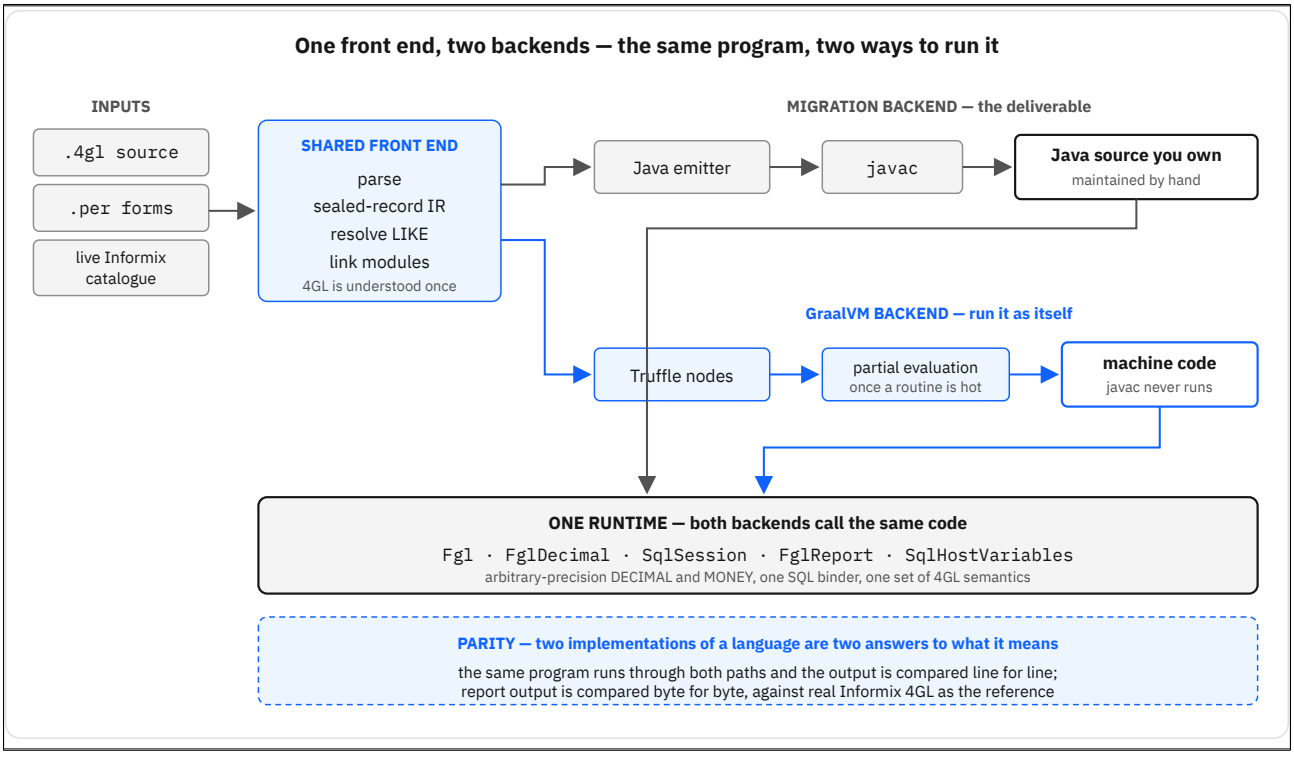

Figure 6 — One front end, two backends — the same program, two ways to run it

What this means for an assessment. Scope is not an estimate. Every construct the compiler cannot yet translate is marked in place in the generated Java and counted per program, so the work in front of a programme is a figure that can be planned and priced from the first day. Files translate and compile around each marker, so progress is continuous and visible rather than arriving at the end.

19. Where the compiler sits in a wider modernisation

Translating 4GL to Java answers one question: what language the rules are written in. It is complete on its own — after cut-over there is no proprietary runtime, no toolchain licence and no dependency on a shrinking specialist market — and it is also a starting point rather than a terminus.

The reason is that a JVM application can do things a 4GL program could not, and those things are usually what the business was asking for when it asked about modernisation. The same rules, unchanged, can be exposed as an HTTP API, published to a message queue, driven by a web or mobile front end, packaged into a container image and observed with whatever the estate already runs. Integration stops being a bespoke extract built for each request. A modern interface, when it is wanted, becomes a project in front of a codebase that can already answer it — rather than a rewrite of the logic underneath.

None of that requires abandoning 4GL, and this is the point most easily lost. What made 4GL a liability was the proprietary runtime, the hiring market and a screen model welded to a character terminal — not its arithmetic or its data handling.

So the routes are meant to be mixed, and the same compiler front end serves all of them. Keep authoring 4GL and let the build translate it, for as long as that suits the people who know the rules. Take one module over in Java and leave the rest. Write new work in Java beside rules still maintained in 4GL. Move everything and stop thinking about 4GL. Or keep the rules in 4GL permanently and run them on a polyglot runtime as a registered language, compiled to native instructions, callable from Java, JavaScript and Python in the same process — the subject of a companion paper. An organisation that picks one of these and changes its mind two years later has lost nothing, because the 4GL remained readable source throughout.

20. Glossary

Term	Meaning
4GL	Informix 4GL, a procedural language with embedded SQL, screen dialogs and a report writer, compiled by `fglpc` against a live catalogue.
`.per` form	A screen layout file: a character-cell canvas plus attribute declarations binding fields to columns.
`CONSTRUCT`	The dialog statement that turns criteria typed into form fields into a SQL `WHERE` clause — query-by-example as a language construct.
`GLOBALS`	A block of variable declarations shared by every module of a program, usually held in a module of its own and named in a `GLOBALS "file"` statement.
Readable output	The design commitment that the emitted Java is reviewable, and therefore adoptable as the source whenever a team chooses — the constraint from which this compiler's other decisions follow.
IR	Intermediate representation: the sealed-record tree every pass rewrites and the emitter walks.
`LIKE` resolution	Replacing `DEFINE x LIKE table.column` with the column's real type, width and scale, read from the catalogue at compile time.
`MONEY(p,s)`	A fixed-point decimal with declared precision and scale; exact within it, and the type most often lost by a naive translation.
Differential testing	Running the same program through the original compiler and this one and comparing output byte for byte.
Host variable	A program variable referenced inside embedded SQL; becomes a bind parameter in the translation.
`WHENEVER`	The statement declaring what a program does when a later statement raises an error or a warning, or finds no rows — stop, continue, or transfer control. Stopping on error is the default.

21. License, Trademarks and Disclaimer

The migration compiler described in this document is 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** is a registered trademark 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.