Source: hl7 docs/ccs_java_hl7.md @ 8fc4c64 — curated 2026-08-24; edits land in the source first.

b3u.dev HL7 — The Good, the Bad, and the Ugly

A technical report on maintaining HL7 software as a product family on the CCS platform — one grammar, one parse, five runtimes, and an obfuscation layer.

Revision 1.0 — 2026-08-24 — BET THREE LLC — b3u.dev

---

1. Why this report

If you build or maintain HL7 v2 software — an interface engine, a lab bridge, a claims pipeline, an integration practice — your stack is polyglot whether you chose that or not:

```
┌──────────────────────────────────────────────────────────┐
│  interface engine / channel logic      JAVA              │
│  analytics / ETL over the same data    PYTHON            │
│  operator consoles, viewers            JAVASCRIPT        │
│  hot paths, instruments, edge devices  C++ / RUST        │
└──────────────────────────────────────────────────────────┘
     every language boundary re-parses the same message today
```

Each layer parses the same message again, into its own object model, with its own copy of the dialect's quirks, maintained by different people on different schedules. That duplication is invisible because it has always been there — and it is where interface drift, version skew, and a large share of integration maintenance cost live.

CCS removes the duplication at the architecture level:

```
ONE PARSE. FIVE RUNTIMES. NO RE-PARSING AT ANY LANGUAGE BOUNDARY.
```

A message is parsed **once**, by a parser generated from a grammar, into an SCB — a *Syntax-Controlled Binary*: the parsed structure itself, persisted as bytes. Java, Python, JavaScript, Rust, and C++ each read that same artifact through a native SCB API and reconstruct the message **byte-**

identically. The same artifact passes through an obfuscation layer (XTEA and ECIES/AES-GCM) for storage and transport.

This report is not a concept paper. Every claim in it is backed by code that runs today, and the code is printed here: a working HL7 ORU^R01 pipeline — grammar, generated parser, in-process compilation from Java, five independent readers, a byte-identity oracle, and the full obfuscation cycle — plus the measurements behind the performance statements. The final chapter is the honest part: what is good, what is currently limited, and what is genuinely hard.

Two ground rules the demo inherits from our engineering policy: all messages are synthetic (no PHI has ever touched this system), and CCS components are batch/data-layer components — we do not sit inline on a clinical path.

---

2. CCS fundamentals

2.1 Grammar → parser → binary

CCS is a grammar-driven data platform, protected by US Patent 8,464,232. Its center of gravity is a compiler-compiler: you write a grammar in SGDL (a compact EBNF-class grammar language), and the b3u.dev toolchain generates from it:

- a **parser/compiler frontend** (C++) for the format the grammar describes — deliverable as a CLI tool or as a shared library with a small C ABI, callable in-process from any language;
- the format's **keyword table in five languages** (C++, Python, JavaScript, Rust, Java) — the per-grammar data that lets each runtime's SCB API resolve the grammar's symbols by name;
- the **SCB layout** for that grammar: every message the frontend accepts becomes a Syntax-Controlled Binary.

An SCB is not a serialization format the layers agree on; it is the parse result itself. It exists in two on-disk forms with identical content:

```
┌─────────┬──────────────────────────────┬──────────────────────────────┐
│ form    │ what it is                   │ when to use it               │
├─────────┼──────────────────────────────┼──────────────────────────────┤
│ .fgr    │ decimal-text record stream   │ portable interchange,        │
│         │ (one word per line)          │ diffable, human-auditable    │
```

```
| .bfgr   | raw binary image, native      | production load path —    |
|         | byte order                    | 20-30× faster to load     |
└─────────┴───────────────────────────────┴───────────────────────────┘
```

The `.bfgr` form is deliberately native-byte-order: you ship the portable `.fgr` and populate the `.bfgr` on the target machine with a one-command tool, so endianness is correct by construction.

2.2 The five SCB APIs

Each runtime has a native SCB API — a reader that loads `.fgr`/`.bfgr` images (or raw SCB bytes straight from memory), navigates the parsed structure, and decompiles it back to text. The five implementations are validated against each other by a standing byte-identity oracle: for the same input, all five produce byte-identical output.

```
┌─────────────┬─────────────────────────────────┬──────────────────────────┐
| runtime     | SCB API                         | notes                    |
├─────────────┼─────────────────────────────────┼──────────────────────────┤
| C++         | the reference implementation    | plus a standalone        |
|             |                                 | SCB-only library for     |
|             |                                 | pure processors          |
| Python      | ccs_scb package                 | pure Python              |
| JavaScript  | ccs_js module                   | Node, ESM                |
| Rust        | ccs_scb_rs crate                | safe Rust                |
| Java        | ccs.scb package                 | pure Java, stdlib only,  |
|             |                                 | ~1.4k LOC; developed and |
|             |                                 | tested on JDK 25 (§3.1)  |
└─────────────┴─────────────────────────────────┴──────────────────────────┘
```

The APIs are **consumers**. Parser generation itself never ships to a client runtime: language bindings and shared libraries expose SCB consumption and compilation *frontends* only. That boundary is a product invariant, not a habit — the generation machinery stays under b3u.dev.

2.3 What "generated" means, concretely

Here is the actual generated Python keyword table for the HL7 demo grammar of §4 (the `.java`, `.mjs`, `.rs`, and `.cc/.h` siblings carry the same lists in their native idiom):

```
# Generated by the b3u.dev toolchain from hl7oru.sgr -- do not edit.
GRAMMAR = "hl7oru"
```

```
PREDEFINED = [
    "Wrong_Key_Word", "identifier", "stringToken", "integerToken",
    "floatToken", "textToken", "termToken", "terminalTokenOfRule",
    "mettaSymbol", "mettaVariable", "mettaSpaceRef", "mettaComment",
    "hl7Data",
]

TOKENS = [
    ("\"", "quoteTermToken"), ("'", "apostropheTermToken"),
    ("(", "leftParenthesisTermToken"), (")", "rightParenthesisTermToken"),
    ("<", "lessThenTermToken"), ("=", "equalTermToken"),
    (">", "greaterThenTermToken"), ("[", "leftSquareBracketTermToken"),
    ("]", "rightSquareBracketTermToken"), ("|", "verticalBarTermToken"),
    ("{", "leftBraceTermToken"), ("}", "rightBraceTermToken"),
    ("::=", "defisTermToken"), (":", "colonMarkTermToken"),
    ("?", "questionMarkTermToken"), ("/", "slashMarkTermToken"),
    ("</", "xmlEndTermToken"), (".", "dotMarkTermToken"),
    (",", "commaTermToken"), (";", "semicolonTermToken"),
    ("->", "edgeOpTermToken"),
]

DECLARED_TERMINALS = [
    "&", "MSH", "NTE", "OBR", "OBX", "PID", "PV1", "^", "~",
]

NON_TERMINALS = [
    "oru", "patientResult", "patientGrp", "orderObs", "obsList",
    "obsUnit", "mshSeg", "pidSeg", "pv1Seg", "obrSeg", "obxSeg",
    "nteSeg", "part", "sep",
]
```

Every reader in this report resolves the grammar's symbols **by name** from this table — no magic numbers, no per-language drift. When the grammar evolves, the tables regenerate; the readers keep working or fail loudly at load, never silently misread.

2.4 The obfuscation layer

The obfuscator transforms an SCB image into an opaque blob and back — in memory, buffer-to-buffer, no disk required:

```
┌────────┬──────────────────────────────┬──────────────────────────────┐
│ algo   │ scheme                       │ properties                   │
```

1	XTEA, seed-keyed	symmetric, lightweight
2	ECIES (secp256k1) + AES-GCM	public-key: encrypt with the public key, decrypt with the private key; wrong key or tampered blob is REFUSED at the GCM tag
3	wire-compatible modern-C++ implementation of algo 2	interoperates with algo 2 in both directions

The security posture is defense in depth for the data layer: a leaked database dump or an intercepted feed of obfuscated SCBs is not readable without keys, and any tampering is detected. It complements — never replaces — transport security and access control (§6).

---

3. Java on CCS

Java is not an afterthought binding. The HL7 integration world is JVM end to end, so Java was built as the **fifth full member** of the SCB API family and validated to the same oracle as the other four. This chapter is the evidence.

3.1 `ccs.scb` — the fifth SCB API

Pure Java, standard library only, about 1,400 lines. Scope: load `.bfgr` (raw image, word size inferred) and `.fgr` (decimal text), navigate the parsed structure, and decompile — byte-identical to the producing compiler's own decompilation.

class	role
Tag	64/32-bit tag words: split/join, little-endian unpack, real handling, C strings
MetaKeywords	the layered keyword table + the meta rows
KeywordModule	loads a generated keyword table (.java or .py sibling) as TEXT — never compiled
SyntaxControlledBinary	.fgr/.bfgr reader; head/ids/kwn/edp navigation; position sections tolerated
FormatReal	shortest-round-trip real formatting via exact decimal search (not Double.toString)

```
| Decompiler                | SCB → text, byte-identical to the C++     |
|                           | compiler's decompilation                  |
└──────────────────────────┴──────────────────────────────────────────┘
```

Validation: on a corpus spanning grammar files, JSON, and symbolic- expression data, the Java decompilation is byte-identical to the C++, Python, and JavaScript decompilations and to the compiler's own reference output; the real-number formatter passes a 39-case golden suite shared with the other runtimes.

3.2 `ccs.shell` — compiling in-process from Java

Generated frontends are also delivered as shared libraries exporting a small C ABI. The Java binding uses the JDK's Foreign Function & Memory API — no JNI glue, no subprocess:

```
try (Shell sh = new Shell("json")) {
    sh.setKeepScb(true);
    int rc = sh.compileString(text, "/work/data.json");  // 0 ok, 1 diagnostics
    byte[] scb = sh.scbBytes();   // the SCB image, no disk in between
    // hand scb straight to ccs.scb.SyntaxControlledBinary
}
```

Two contract points worth noting, because they are the difference between a demo binding and a production one:

- **A syntax error is an outcome, not an exception.** `compile*` returns 0 or 1; diagnostics land in listing/log files exactly as from the CLI. Exceptions are reserved for API misuse.
- **Thread affinity is explicit.** A `Shell` belongs to its creating thread; hand-off to another thread is an explicit `rebindToCurrentThread()` by the receiver.

The binding's suite proves CLI equivalence: bytes produced through the in-process path are identical to the CLI tool's output files.

3.3 `ccs.obf` — the obfuscation cycle from Java

The same pattern over the obfuscator's C ABI:

```
try (Obf o = new Obf("app")) {
    byte[] wire = o.obfuscateBuffer(1, scbBytes, "seed");  // XTEA, no disk
    byte[] back = o.deobfuscateBuffer(1, wire, "seed");    // == scbBytes
```

```
    String[] kp = Obf.eciesKeypair();                       // [private, public]
    byte[] enc = o.obfuscateBuffer(2, scbBytes, kp[1]);     // ECIES
    byte[] dec = o.deobfuscateBuffer(2, enc, kp[0]);
}
```

Its suite covers round trips for all algorithms, cross-decode between the two ECIES implementations in both directions, refusal of wrong keys and tampered blobs at the GCM tag, and **co-residence**: the compile binding and the obfuscator binding loaded in one JVM, feeding each other byte-exactly.

3.4 dag5 — the five-runtime showcase

Before HL7, the five-runtime architecture was proven end to end on a neutral subject: a DAG (directed-acyclic-graph) model grammar. The pipeline compiles a DAG description in-process from Java, and all five runtimes read the same compiled SCB:

runtime	substrate	render == Java	obf
Java	ccs.scb	(reference)	1/2/3
Python	ccs_scb	byte-identical	1/2
JavaScript	ccs_js	byte-identical	1/2
Rust	ccs_scb_rs	byte-identical	—
C++	C++ SCB runtime	byte-identical	—

All five renders are byte-identical; the SCB is a fixpoint under compile → render → recompile; Java, Python, and JavaScript carry it through the full obfuscation cycle. dag5 is the template the HL7 demo of §4 instantiates on a real-world format.

3.5 JSON benchmarking on the JVM

To place the Java SCB API against the ecosystem it must live in, we mirrored our C++ JSON benchmark on the JVM: Jackson databind 2.18.2, Gson 2.11.0, and fastjson2 2.0.53 versus the CCS Java reader, on the standard public corpus (twitter.json 568 KB, citm_catalog.json 1.7 MB, canada.json 2.25 MB), mean µs over 100 iterations with a fixed 10-iteration JIT warmup, OpenJDK 25, default flags, one dev box.

Correctness gates the numbers. Before any timing is reported, all four Java walkers tally every leaf of the parsed tree into six counts (strings / ints / reals / true / false / null) and the run fails unless all four agree on every cell *and* match the C++ benchmark's recorded reference cells. They do, on every sample.

Parse / load (mean µs, lower is better; text parsers get the payload string, CCS loads the binary file per iteration):

corpus	jackson	gson	fastjson2	CCS.bfgr	CCS.fgr
twitter.json	1581.8	1805.7	1170.9	1586.8	18559.4
citm_catalog.json	2773.8	2952.8	2461.7	3575.9	53604.9
canada.json	21750.8	8407.9	5093.3	8980.8	147538.7

Counts walk (parse once, walk per iteration):

corpus	jackson	gson	fastjson2	CCS walk
twitter.json	359.7	232.4	179.4	257.9
citm_catalog.json	653.5	692.6	229.6	799.1
canada.json	2958.4	1013.1	338.8	2117.1

The C++ reference on the same box and corpus (g++ -O2, µs):

corpus	nlohmann	simdjson	CCS .bfgr	CCS .fgr
twitter.json	4299	134	300	8749
citm_catalog.json	8542	362	814	17997
canada.json	26187	1643	2362	46081

How to read this honestly:

1. **The parse-once thesis holds on the JVM.** Loading the parsed-once binary costs about what ONE Jackson parse costs (twitter: 1587 vs 1582 µs) — and on canada it is 2.4× faster

than Jackson's parse. Every consumer after the first is a load, not a parse. The binary-vs-text load ratio (~12–16× on the JVM, 20–30× in C++) survives the runtime change.
2. **The walk is competitive out of the box** — between Gson and Jackson on two of three samples, unoptimized. canada (111k float leaves) favors the JSON DOMs' packed primitives; on the C++ side that query class is answered by a grammar-aware O(1) derivation rather than a tree walk, which we have not ported to Java.
3. **The Java loader trails the C++ loader ~5×** on the load phase. The cause is located (a scalar byte-to-word conversion loop that a bulk `ByteBuffer` copy would remove); it is an optimization TODO, and no correctness result depends on it.
4. Methodology caveats: one machine, mean-of-100 with fixed warmup (the C++ benchmark's methodology, not JMH), default JVM flags. Read the numbers **relatively**, against the same-box C++ table.

The core of the measured CCS walk, to show there is no trick in it — symbols resolved by name, one recursive descent over the binary:

```
void count(long tagWord, Counts c) {
    if (tagWord == 0) return;
    long kkk = tag.splitT(tagWord);     // node kind
    long nnn = tag.splitD(tagWord);     // node payload
    if (kkk == MetaKeywords.PLS_STRING_TOKEN)  { c.s++; return; }
    if (kkk == MetaKeywords.PLS_INTEGER_TOKEN) { c.i++; return; }
    if (kkk == MetaKeywords.PLS_FLOAT_TOKEN)   { c.r++; return; }
    if (kkk == MetaKeywords.PLS_TERMINAL_TOKEN) {
        if (nnn == kwTrue) c.t++;
        else if (nnn == kwFalse) c.f++;
        else if (nnn == kwNull) c.n++;
        return;
    }
    EdpirType e = edps[(int) kkk][(int) nnn];
    long fStart = (kkk == kwPair) ? 1 : 0;   // object KEYS are not values
    for (long i = fStart; i < e.fl; i++) count(mem[(int)(e.d + i)], c);
    for (long i = 0; i < e.dl; i++)      count(mem[(int)(e.d + e.fl + i)], c);
}
```

---

4. The HL7 demonstration

Everything in §2–§3 now runs on HL7 content. The demonstration is a complete, gated pipeline: an HL7 v2 ER7 tokenizer in the compiler, an ORU^R01 (lab results) subset grammar, a generated parser delivered as a shared library, a Java pipeline that compiles in-process and carries the message through the obfuscation cycle, and four more readers — Python, JavaScript, Rust, C++ — that all load the *same* compiled SCB and render it byte-identically. One command runs every gate.

4.1 The HL7 tokenizer

HL7 v2's ER7 encoding is not a token stream a general-purpose lexer can guess: `|`, `^`, `~`, `&` separate; everything between them — spaces, backslashes, dots, dashes, timestamps like `2.3.1` and `19800101` — is data; a segment identifier such as `OBX` is a keyword **only at the start of a line** and plain data anywhere else.

So the compiler carries a dedicated HL7 tokenizer as a first-class language mode (alongside its meta-grammar, XML, and symbolic- expression modes), selected explicitly or simply by the `.hl7` file extension. Its contract:

- one segment per line; the line-leading identifier is matched against the grammar's segment keywords;
- field payload is a dedicated data token: runs of bytes that stop **only** at the delimiters the grammar itself references — never at characters that merely happen to be special in other language modes;
- no comment syntax, no whitespace normalization inside fields: what is in the message is in the token, byte for byte.

That last property is what makes everything downstream measurable: the canonical render of a parsed message equals the input bytes, so fidelity is a `diff`, not an argument.

Demo scope, stated plainly: this tokenizer bakes in the conventional delimiter set `|^~\&`. Real HL7 allows MSH-1/MSH-2 to re-declare delimiters per message; the MSH-driven settable-delimiter surface is designed and on the roadmap, not in the demo (§6.2).

4.2 The grammar

The entire format definition that drives everything in this chapter — parser generation, the five keyword tables, the SCB layout — is this one page. Segment-level message structure per the

ORU^R01 skeleton (patient → orders → observations, with optional PV1 and NTE); fields stay a flat, verbatim token stream:

```
(hl7oru
  (oru ::= mshSeg patientResult { patientResult } )

  (patientResult ::= patientGrp orderObs { orderObs } )
  (patientGrp    ::= pidSeg [ pv1Seg ] )
  (orderObs      ::= obrSeg [ obsList ] )
  (obsList       ::= obsUnit { obsUnit } )
  (obsUnit       ::= obxSeg [ nteSeg ] )

  (mshSeg ::= 'MSH' { part } )
  (pidSeg ::= 'PID' { part } )
  (pv1Seg ::= 'PV1' { part } )
  (obrSeg ::= 'OBR' { part } )
  (obxSeg ::= 'OBX' { part } )
  (nteSeg ::= 'NTE' { part } )

  (part ::= sep | hl7Data )
  (sep  ::= ( '|' | '^' | '~' | '&' ) )
)
```

(The production grammar file carries two bracketing action hooks on the axiom rule; they are elided here for readability.)

Because the structure is in the grammar, the **parser enforces it**. The gate proves this with red witnesses: a message whose OBR precedes any PID is refused; a message that does not start with MSH is refused. Structural validation is not a library you remember to call — it is what parsing means.

4.3 The sample message

One synthetic two-patient ORU^R01 lab report (no real data of any kind; the LOINC/SNOMED codes are the standard public vocabulary):

```
MSH|^~\&|CCSLAB^1.2.3^ISO|LAB|CCSEHR|CLINIC|20260823120000||ORU^R01|MSG0001|P|2.3.1
PID|1||PATID1234^^^MRN||DOE^JANE Q||19800101|F
PV1|1|O
OBR|1|ORD123||94500-6^SARS-CoV-2 RNA^LN|||20260822
OBX|1|CV|94500-6^SARS-CoV-2^LN||260415000^Not detected^SCT||||||F
```

```
NTE|1|L|Result verified--auto release
OBX|2|NM|GLU^Glucose^LN||95|mg/dL|70-105|N|||F
PID|2||PATID5678^^^MRN||ROE^RICHARD||19751115|M
OBR|1|ORD456||94500-6^SARS-CoV-2 RNA^LN|||20260822
OBX|1|ST|PID^Comment field^L||OBX and PID here are data not segments||||||F
```

The sample is deliberately adversarial: patient 1 has a PV1, patient 2 does not; one OBX carries an NTE, its sibling does not; `2.3.1`, `94500-6`, `70-105`, and `--` must survive as single data runs; and the last OBX's value field contains the words "OBX and PID" — which must parse as *data*, not as segment starts. Every reader asserts all of these.

4.4 The pipeline at a glance

```
   oru1.hl7 (ER7 text)
       │  in-process compile: Java → FFM → libccshl7oru.so (C ABI)
       ▼
   SCB image (oru1.bfgr, 7,872 bytes for this sample)
       │
       ├── Java    ccs.hl7  ──►  canonical render == INPUT BYTES
       ├── Python  reader   ──►  render == Java render (byte-identical)
       ├── JS      reader   ──►  render == Java render (byte-identical)
       ├── Rust    reader   ──►  render == Java render (byte-identical)
       └── C++     reader   ──►  render == Java render (byte-identical)
       │
       └── obfuscate / deobfuscate (algo 1 XTEA, algo 2/3 ECIES,
           cross-decode both ways, wrong-key REFUSED) — proven from
           Java, Python, and JavaScript
```

The generated frontend is delivered two ways from the same build: a CLI executable, and `libccshl7oru.so` — a shared library whose entire export surface is the small `ccs_*` C ABI (the gate verifies with `nm` that nothing else is exported). The Java pipeline uses the library: message text goes in as bytes, the SCB image comes back as bytes, and nothing touches disk in between.

4.5 The Java pipeline

Four small classes and a facade. First the domain model — typed at the group level (the navigation story), verbatim at the field level (the fidelity story):

```
public final class Entities {
    /** One token of a segment: a delimiter or a field-data run. */
    public static final class Part {
```

```
        public final boolean delimiter;
        public final String text;
        public Part(boolean delimiter, String text) { … }
    }
    /** One ER7 segment: its ID and its token stream. */
    public static final class Segment {
        public String id = "";
        public final List<Part> parts = new ArrayList<>();
        public String line() {           // the verbatim segment line
            StringBuilder sb = new StringBuilder(id);
            for (Part p : parts) sb.append(p.text);
            return sb.toString();
        }
    }
    public static final class ObsUnit { public Segment obx;
                                        public Segment nte; }
    public static final class Order   { public Segment obr;
                                        public final List<ObsUnit> obsUnits; }
    public static final class Patient { public Segment pid; public Segment pv1;
                                        public final List<Order> orders; }
    public static final class Message { public Segment msh;
                                        public final List<Patient> patients; }
}
```

The loader is the one piece of real logic in the pipeline — a grammar-aware walk over the SCB. It is worth reading in full once, because the Python, JavaScript, Rust, and C++ readers in the next sections are *this walk*, transposed. The encodings it relies on:

- a terminal (segment ID or delimiter) is a leaf tagged with the terminal's keyword value; its text comes from the keyword table;
- field data is a leaf pointing into the SCB's interned-string store;
- each grammar rule instance carries a fixed child list and a dynamic (iteration) list; optionals shift positions, so group children are dispatched **by kind, never by position**.

```
public final class Loader {

    private final SyntaxControlledBinary b;
    private final KwnirType[] byK;          // rule table indexed by KW value
    private final KeywordTable kw;

    // grammar KW values resolved BY NAME from the generated table
    private final long oru, patientResult, patientGrp, orderObs, obsList,
```

```
        obsUnit, mshSeg, pidSeg, pv1Seg, obrSeg, obxSeg, nteSeg,
        sep, hl7Data;

    /** Load a raw SCB image into the message model. */
    public static Message fromBytes(byte[] scb, KeywordTable kw) {
        SyntaxControlledBinary b = new SyntaxControlledBinary();
        b.readBinaryRawBytes(scb, null);
        return new Loader(b, kw).run();
    }

    /** A terminal's source text: literal or declared spelling. */
    private String termText(long kwValue) {
        Keyword k = kw.get(kwValue);
        return (k.literal != null) ? k.literal : k.name;
    }

    private Segment readSegment(long segKw, long idx) {
        EdpirType e = edp(segKw, idx);
        Segment s = new Segment();
        for (long i = 0; i < e.fl; i++) {            // fixed children
            long w = b.fixedAt(e, i);
            if (kind(w) == KW_TERMTOKEN) s.id = termText(num(w));
        }
        for (long j = 0; j < e.dl; j++) {            // the { part } stream
            long w = b.dynamicAt(e, j);
            long k = kind(w);
            if (k == sep) {                          // a delimiter node
                EdpirType se = edp(sep, num(w));
                s.parts.add(new Part(true, termText(num(b.fixedAt(se, 0)))));
            } else if (k == hl7Data) {               // a field-data run
                s.parts.add(new Part(false, dataText(num(w))));
            } else if (k == KW_TERMTOKEN) {
                s.parts.add(new Part(true, termText(num(w))));
            } else {
                throw new Hl7Exception("segment " + s.id
                    + ": unexpected part kind " + k);
            }
        }
        return s;
    }

    private ObsUnit readObsUnit(long idx) {
        EdpirType e = edp(obsUnit, idx);
        ObsUnit u = new ObsUnit();
```

```
        for (long i = 0; i < e.fl; i++) {            // dispatch BY KIND
            long w = b.fixedAt(e, i);
            long k = kind(w);
            if (k == obxSeg)      u.obx = readSegment(obxSeg, num(w));
            else if (k == nteSeg) u.nte = readSegment(nteSeg, num(w));
        }
        if (u.obx == null) throw new Hl7Exception("obsUnit has no OBX");
        return u;
    }

    // readOrder / readPatient / readPatientResult follow the same
    // by-kind pattern up the group spine …

    private Message run() {
        long axiom = b.head().cntbeg;                // the parse root
        if (kind(axiom) != oru)
            throw new Hl7Exception("axiom is not oru");
        EdpirType e = edp(oru, num(axiom));
        Message m = new Message();
        for (long i = 0; i < e.fl; i++) {
            long w = b.fixedAt(e, i);
            if (kind(w) == mshSeg) m.msh = readSegment(mshSeg, num(w));
            else if (kind(w) == patientResult)
                m.patients.add(readPatientResult(num(w)));
        }
        for (long j = 0; j < e.dl; j++) {            // { patientResult }
            long w = b.dynamicAt(e, j);
            if (kind(w) == patientResult)
                m.patients.add(readPatientResult(num(w)));
        }
        if (m.msh == null) throw new Hl7Exception("message has no MSH");
        return m;
    }
}
```

The renderer is almost nothing — which is the point. Because parts are verbatim, canonical rendering is concatenation in grammar order:

```
public static byte[] toEr7Bytes(Message msg) {
    StringBuilder sb = new StringBuilder();
    line(sb, msg.msh);
    for (Patient p : msg.patients) {
        line(sb, p.pid);
```

```
        if (p.pv1 != null) line(sb, p.pv1);
        for (Order o : p.orders) {
            line(sb, o.obr);
            for (ObsUnit u : o.obsUnits) {
                line(sb, u.obx);
                if (u.nte != null) line(sb, u.nte);
            }
        }
    }
    return sb.toString().getBytes(StandardCharsets.ISO_8859_1);
}
```

The facade ties compile, load, render, and the obfuscation cycle into five methods:

```
public final class Hl7Oru {
    public byte[] compileText(String text)     { … } // ER7 → SCB, in-proc
    public Entities.Message load(byte[] scb)   { … } // SCB → model
    public byte[] render(Entities.Message msg) { … } // model → canonical
    public byte[] canonical(String text) {
        return render(load(compileText(text)));
    }

    public byte[] obfuscate(int algo, String keySeed, byte[] scb)    { … }
    public byte[] deobfuscate(int algo, String keySeed, byte[] wire) { … }
    public static String[] eciesKeypair()          { … }
}
```

The Java test battery (21 checks, all green) asserts, in order: the structure of §4.3 in every detail (two patients, PV1 present/ absent, NTE present/absent, the "OBX and PID" value loaded as data); then the two properties that carry the whole report:

```
byte[] canon = oru.render(m);
check(Arrays.equals(canon, text.getBytes(ISO_8859_1)),
      "canonical render == input bytes");
byte[] scb2 = oru.compileText(new String(canon, ISO_8859_1));
check(Arrays.equals(scb1, scb2),
      "SCB fixpoint: compile(render) == compile(input)");
```

and finally the obfuscation cycle: the XTEA round trip; ECIES round trips under both implementations; cross-decode between them in both directions; a **wrong private key refused** at the authentication tag; and model → obfuscate → deobfuscate → model still loading correctly.

4.6 The Python runtime

One file: entities, loader, renderer — the Java walk transposed onto the Python SCB API. The keyword table section shows the whole by-name discipline in a dozen lines:

```
from ccs_scb.binary import SyntaxControlledBinary

def _load_kwd():
    g = {}
    exec(compile(_KWD.read_text(), str(_KWD), "exec"), g)
    names = (g["PREDEFINED"]
             + [t[1] for t in g["TOKENS"]]
             + g["DECLARED_TERMINALS"]
             + g["NON_TERMINALS"])
    text = {}                       # terminal value -> source text
    base = len(g["PREDEFINED"])
    for i, (lit, _n) in enumerate(g["TOKENS"]):
        text[base + i] = lit
    base += len(g["TOKENS"])
    for i, spelling in enumerate(g["DECLARED_TERMINALS"]):
        text[base + i] = spelling
    return names, text

_NAMES, _TERMTEXT = _load_kwd()

def _kw(name: str) -> int:
    return _NAMES.index(name)
```

The segment walk — compare it line for line with the Java `readSegment` above:

```
def _read_segment(w, seg_kw, idx):
    e = w.edp(seg_kw, idx)
    s = Segment()
    for i in range(e.fl):
        c = w.child(e, i)
        if w.kind(c) == _KW_TERMTOKEN:
            s.id = w.term_text(w.num(c))
    for j in range(e.dl):
        c = w.dyn(e, j)
        k = w.kind(c)
        if k == _KW_SEP:
            se = w.edp(_KW_SEP, w.num(c))
```

```
            s.parts.append(Part(True, w.term_text(w.num(w.child(se, 0)))))
        elif k == _KW_HL7DATA:
            s.parts.append(Part(False, w.data_text(w.num(c))))
        elif k == _KW_TERMTOKEN:
            s.parts.append(Part(True, w.term_text(w.num(c))))
        else:
            raise RuntimeError(f"segment {s.id}: unexpected part kind {k}")
    return s
```

And the renderer:

```
def to_er7_text(m: Message) -> str:
    lines = [m.msh.line()]
    for p in m.patients:
        lines.append(p.pid.line())
        if p.pv1 is not None:
            lines.append(p.pv1.line())
        for o in p.orders:
            lines.append(o.obr.line())
            for u in o.obs_units:
                lines.append(u.obx.line())
                if u.nte is not None:
                    lines.append(u.nte.line())
    return "".join(x + "\n" for x in lines)
```

The Python parity test loads **the Java-compiled** SCB — not its own compile — renders, and requires byte identity with the Java render; then re-asserts the structural checks; then runs the obfuscation round trips through the Python obfuscator binding:

```
m = from_bfgr(work / "oru1.bfgr")            # the JAVA-produced binary
py_render = to_er7_text(m).encode("latin-1")
check(py_render == java_render,
      "python render(java .bfgr) == java canonical render (byte-identical)")

with obf.Obfuscator("hl7demo-py") as o:
    w1 = o.obfuscate_bytes(1, scb, key_seed="hl7-seed")
    check(o.deobfuscate_bytes(1, w1, key_seed="hl7-seed") == scb,
          "python algo-1 round-trip recovers SCB")
    priv, pub = obf.ecies_keypair()
    w2 = o.obfuscate_bytes(2, scb, key_seed=pub)
```

```
    check(o.deobfuscate_bytes(2, w2, key_seed=priv) == scb,
          "python algo-2 (ECIES) round-trip recovers SCB")
```

4.7 The JavaScript runtime

Same shape, ESM module over the JavaScript SCB API. The keyword table loads from the generated `.mjs` sibling; the walk is character-for- character the same logic:

```
const { SyntaxControlledBinary } =
  await import(join(_CCS_JS, "src", "binary.js"));
const _KWD = await import(join(_DEMO, "cppcc", "srcloc",
                               "hl7oruKeyWordDefinition.mjs"));

const _names = [
  ..._KWD.PREDEFINED,
  ..._KWD.TOKENS.map(([_lit, n]) => n),
  ..._KWD.DECLARED_TERMINALS,
  ..._KWD.NON_TERMINALS,
];

function readSegment(w, segKw, idx) {
  const e = w.edp(segKw, idx);
  const s = new Segment();
  for (let i = 0; i < e.fl; i++) {
    const c = w.child(e, i);
    if (w.kind(c) === KW.TERMTOKEN) s.id = w.termText(w.num(c));
  }
  for (let j = 0; j < e.dl; j++) {
    const c = w.dyn(e, j);
    const k = w.kind(c);
    if (k === KW.SEP) {
      const se = w.edp(KW.SEP, w.num(c));
      s.parts.push(new Part(true, w.termText(w.num(w.child(se, 0)))));
    } else if (k === KW.HL7DATA) {
      s.parts.push(new Part(false, w.dataText(w.num(c))));
    } else if (k === KW.TERMTOKEN) {
      s.parts.push(new Part(true, w.termText(w.num(c))));
    } else {
      throw new Error(`segment ${s.id}: unexpected part kind ${k}`);
    }
  }
```

```
  return s;
}
```

Its parity test mirrors the Python one: load the Java-compiled binary, render byte-identically, re-assert structure, round-trip through the JavaScript obfuscator binding.

4.8 The Rust runtime

A crate over the Rust SCB API. The generated `.rs` keyword table is included as a module, so symbol resolution is checked at *compile time* against the same generated lists the other runtimes read at run time:

```
#[path = "…/hl7oruKeyWordDefinition.rs"]
mod kwd;

fn kw_table() -> &'static HashMap<&'static str, i64> {
    static TABLE: OnceLock<HashMap<&'static str, i64>> = OnceLock::new();
    TABLE.get_or_init(|| {
        let mut m = HashMap::new();
        let mut i: i64 = 0;
        for name in kwd::PREDEFINED          { m.insert(*name, i); i += 1; }
        for (_lit, name) in kwd::TOKENS      { m.insert(*name, i); i += 1; }
        for name in kwd::DECLARED_TERMINALS  { m.insert(*name, i); i += 1; }
        for name in kwd::NON_TERMINALS       { m.insert(*name, i); i += 1; }
        m
    })
}

fn read_segment(w: &Walk, seg_kw: i64, idx: i64) -> Result<Segment, Error> {
    let e = w.edp(seg_kw, idx)?;
    let mut s = Segment::default();
    for i in 0..e.fl {
        let c = w.child(&e, i);
        if w.kind(c) == kw("termToken") {
            s.id = w.term_text(w.num(c))?;
        }
    }
    for j in 0..e.dl {
        let c = w.dyn_child(&e, j);
        let k = w.kind(c);
        if k == kw("sep") {
            let se = w.edp(kw("sep"), w.num(c))?;
```

```
            let tw = w.child(&se, 0);
            s.parts.push(Part { delimiter: true,
                                text: w.term_text(w.num(tw))? });
        } else if k == kw("hl7Data") {
            s.parts.push(Part { delimiter: false,
                                text: w.data_text(w.num(c)) });
        } else if k == kw("termToken") {
            s.parts.push(Part { delimiter: true,
                                text: w.term_text(w.num(c))? });
        } else {
            return Err(Error::Walk(format!("unexpected part kind {k}")));
        }
    }
    Ok(s)
}
```

Errors are `Result`-typed throughout — a malformed or truncated SCB is a value, not a panic. Its parity example loads the Java-compiled binary and asserts byte identity plus structure.

4.9 The C++ runtime

A standalone reader over the C++ SCB runtime. It uses the generated C++ keyword **enum header** instead of a runtime table, so here the compiler itself pins the grammar symbols:

```
#include "SyntaxControlledBinary.h"
#include "hl7oruKeyWordDefinition.h"     // the generated KW_* enum
namespace kw = cppcc::hl7oru;

static std::string termText(Long v) {
    switch (v) {
    case kw::KW_VERTICALBARTERMTOKEN: return "|";
    case kw::KW_TERMINAL_CARET:       return "^";
    case kw::KW_TERMINAL_TILDE:       return "~";
    case kw::KW_TERMINAL_AMPERSAND:   return "&";
    case kw::KW_TERMINAL_MSH:         return "MSH";
    case kw::KW_TERMINAL_PID:         return "PID";
    case kw::KW_TERMINAL_PV1:         return "PV1";
    case kw::KW_TERMINAL_OBR:         return "OBR";
    case kw::KW_TERMINAL_OBX:         return "OBX";
    case kw::KW_TERMINAL_NTE:         return "NTE";
    default: throw std::runtime_error("terminal KW not in table");
    }
```

```
}

static Segment readSegment(Walk& w, Long segKw, Long idx) {
    Bin::edpirType* e = w.edp(segKw, idx);
    Segment s;
    for (Long i = 0; i < e->fl; i++) {
        Long c = w.child(e, i);
        if (w.kind(c) == KW_TERMTOKEN) s.id = termText(w.num(c));
    }
    for (Long j = 0; j < e->dl; j++) {
        Long c = w.dyn(e, j);
        Long k = w.kind(c);
        if (k == kw::KW_SEP) {
            Bin::edpirType* se = w.edp(kw::KW_SEP, w.num(c));
            s.parts.push_back({true, termText(w.num(w.child(se, 0)))});
        } else if (k == KW_HL7DATA) {
            s.parts.push_back({false, w.dataText(w.num(c))});
        } else if (k == KW_TERMTOKEN) {
            s.parts.push_back({true, termText(w.num(c))});
        } else {
            throw std::runtime_error("unexpected part kind");
        }
    }
    return s;
}
```

Same parity contract: load the Java-compiled binary, render, byte identity with the Java render, structural asserts.

4.10 The five-way oracle — and the red that earned its place

One script runs the whole demonstration: grammar generation, frontend link, structured parse with nine content witnesses, shared-library build with export-surface verification, the in-process compile smoke test, the two structural red witnesses, the 21-check Java battery, and the four parity legs. Its final section deserves quoting because it shows the engineering culture the numbers come from:

> The first corrupt-a-byte probe (flip the byte at the middle of the file) came back **silently identical** — the flipped byte landed in parser tables the readers never consume. "We corrupt a byte and the comparator fails" is only evidence if the corruption lands on the read path. The shipped check flips a byte *inside an interned patient-name string* — data

> every runtime must read — and every loader now refuses it. A verification that cannot fail is not a verification; we test our oracles the way we test our code.

The measured result, end to end:

runtime	substrate	render == Java	obf
Java	ccs.scb + ccs.obf	(reference — == INPUT)	1/2/3
Python	ccs_scb + obfuscator binding	byte-identical	1/2
JavaScript	ccs_js + obfuscator binding	byte-identical	1/2
Rust	ccs_scb_rs	byte-identical	—
C++	C++ SCB runtime	byte-identical	—

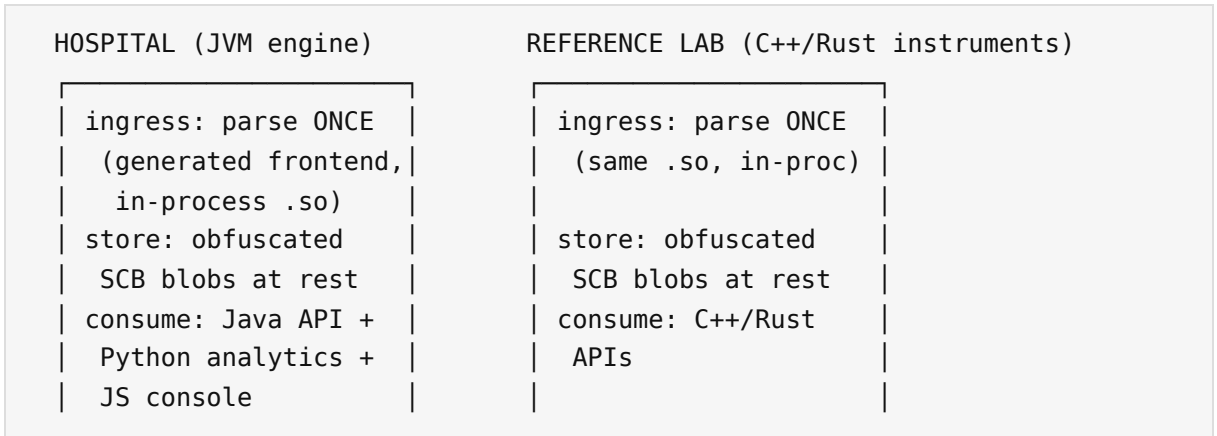
One parse. Five runtimes. No re-parsing at any language boundary. On HL7 content, this is now a measured fact, not a slide.

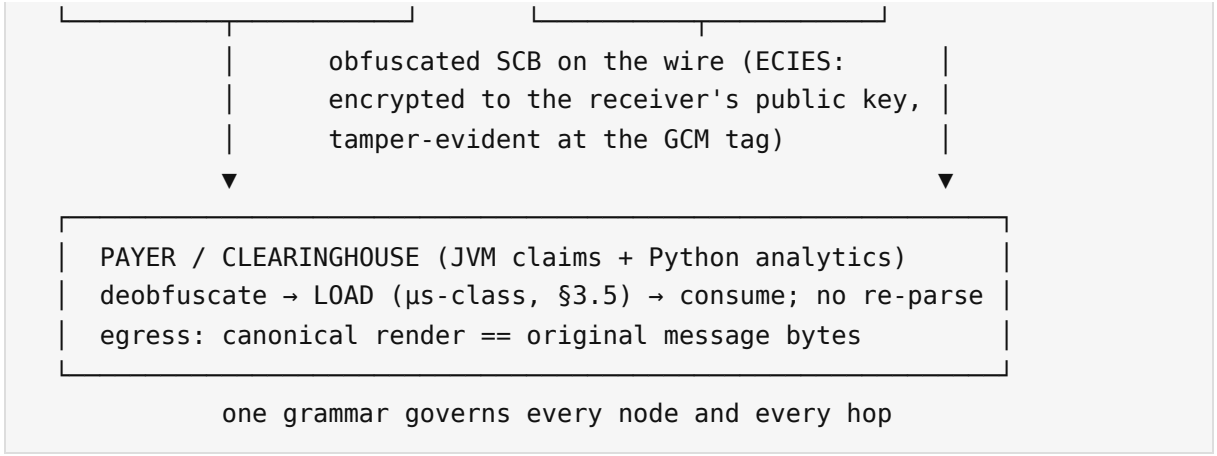
---

5. A distributed HL7 network on CCS

A real HL7 estate is a network: hospitals, reference labs, payers, clearinghouses, and client-facing applications, each with its own stack, exchanging the same messages. Today every node re-parses at ingress, stores text or a proprietary object dump, and re-serializes at egress — and the security of the data layer is whatever each node's database happens to provide.

The CCS deployment pattern places the same two components at every node, regardless of the node's language:





What each layer of the pattern buys:

At the node (storage). Messages persist as SCB — already parsed, loadable in microseconds by any local consumer in its own language. Sensitive stores follow the key-columns-plus-obfuscated-blob pattern: the queryable keys stay plain, the payload is an ECIES-obfuscated SCB. A leaked dump is unreadable without the private key; a tampered blob refuses to decrypt. The application keeps working because the obfuscation boundary is beneath its API.

On the wire (communication). Nodes exchange obfuscated SCB instead of raw ER7. The receiver deobfuscates and *loads* — it does not parse, so it cannot parse *differently*: sender and receiver interoperate by construction because they share the grammar, not because their two hand-written parsers happen to agree. Dual-key operation separates producer from consumer: a node that can encrypt to you cannot read your store.

At the boundary (fidelity and audit). The canonical render equals the original message bytes — measured, not assumed (§4.5). Any node can re-emit the exact bytes it ingested, which is what an audit trail or a downstream legacy consumer actually requires.

What this pattern is not. It is not an interface engine and does not replace one — routing, transformation workflows, and clinical alerting remain the engine's job, and CCS components slot underneath as the parse/persist/exchange substrate. It is also not a transport security replacement: TLS, authentication, and access control remain in place; the obfuscation layer is defense in depth for the data itself. And per our standing policy, CCS components do not sit inline on a clinical path.

---

6. The good, the bad, and the ugly

The title chapter. Everything above is evidence; this is the assessment we would want to read if we were the buyer.

6.1 The good

Interoperability and maintenance: grammar → API → application. The entire format definition is one page of grammar (§4.2). From it, the toolchain generates the parser and the five per-language keyword tables; the five SCB APIs then give every team grammar-aware access to the same parsed artifact. The maintenance consequence is structural: when the format evolves, **the grammar changes in one place**, regeneration propagates it, and the byte-identity oracle tells you — mechanically, in one command — whether every runtime still agrees. Compare that with the status quo: N hand-written parsers in N languages, each patched separately, with no oracle that they agree beyond production incidents. Structural validation comes free (the red witnesses of §4.2: wrong segment order is refused at parse), and the five demo readers were written against the SCB encodings once and ran green on the first attempt in all four ported languages — the transposition is mechanical, which is exactly what makes it cheap to maintain.

Security based on obfuscation. The SCB is already a binary an attacker cannot casually read; the obfuscation layer makes that a guarantee rather than an inconvenience: XTEA for lightweight symmetric cases, ECIES over secp256k1 with AES-GCM authentication for the real ones — wrong key refused, tampered blob refused, cross-implementation wire compatibility proven in both directions, and the whole cycle exercised from Java, Python, and JavaScript in the demo you just read. In a sector where the breach headline is a board-level event, "the leaked data layer was not readable" is a defense-in-depth statement your CISO can actually make.

Scalability and performance. The architectural property: parse cost is paid once per message, at one node, and every subsequent consumer — same process, same node, or across the network — pays a binary load instead. The measured anchors: on the JVM, loading the parsed binary costs about one Jackson parse, and the binary form loads 12–16× faster than its text form (20–30× in C++, where the load lands within ~2× of simdjson's DOM parse). Readers are stateless over an immutable artifact, which is the shape that scales horizontally. What we have *not* measured is a distributed deployment under production load — §6.2 keeps that honest.

6.2 The bad (current limits — real, and fixable)

1. **Demo-scope delimiters.** The tokenizer bakes in the conventional `|^~\&`. Real-world HL7 lets MSH re-declare delimiters per message. The MSH-driven surface is designed (the

MSH segment self-declares, the tokenizer reconfigures) but not yet productized. Until it is, nonstandard-delimiter feeds are out of scope.

2. **A subset grammar.** `hl7oru.sgr` covers the ORU^R01 skeleton with six segment types. A production engagement needs the segment catalog and message types *your* interfaces use — which is exactly the per-engagement work the grammar layer makes cheap, but it is work, and we won't pretend otherwise.
3. **The entity-model layer is hand-cloned per runtime.** The keyword tables are generated in five languages; the typed domain models (Loader/Renderer per language) are currently written by hand from the SCB encodings. The demo shows the transposition is mechanical (four first-attempt-green ports), and generating this layer is on the roadmap — but today it is disciplined cloning, not generation.
4. **Datatype fidelity is asserted at the byte level only.** Canonical render == input bytes is a strong guarantee, but we have not yet separately certified semantic datatype handling (TS time zones, NM precision, escape sequences) against an HL7 conformance suite. It is a named, scheduled residue — not a discovered surprise.
5. **The Java loader trails C++ by ~5×** on binary load, with the cause located and a straightforward fix identified (§3.5). The walk performance is already competitive.
6. **`.bfgr` is native-byte-order by design.** Ship the portable text form, populate the binary on the target. The one-command tool exists; it is an operational step you must know about.
7. **Grammar authoring is a discipline.** SGDL grammars are LL(1); writing them well takes the kind of care §4.2's grammar shows (e.g., delimiter alternatives behind a wrapper rule). In the b3u.dev model, grammar authoring is work we deliver, so this cost sits with us — but it is real.

6.3 The ugly (the hard truths — for us and for you)

1. **HL7 v2 in the wild is not HL7 v2 in the spec.** Z-segments, site-specific field reuse, dialect drift per sender, decades of "the interface works, don't touch it." No parser technology makes that disappear. What CCS changes is where the mess lives: in a reviewable grammar variant per feed, instead of in conditional code scattered across N codebases. The mess remains yours to specify; it becomes ours to hold still.
2. **The demo proves architecture, not coverage.** Five runtimes, byte identity, obfuscation, structural enforcement — proven. The full ORU catalog, ADT, ORM, batch protocols, MLLP framing, acknowledgment semantics — not in this demo, and a buyer should ask for exactly that scoping conversation.

3. **Certification, support, and the clinical relationship stay with the provider.** We sell components to HL7 software providers; we are not HITRUST-certified, we do not run a 24/7 clinical support desk, and we will not sit inline on a clinical path. If your procurement requires the component vendor to be those things, we are the wrong vendor — deliberately.
4. **Pilots run on synthetic data only.** No PHI reaches b3u.dev systems, ever — your conformance corpus, de-identified or synthetic, is the oracle. This protects both parties, and it is non-negotiable even when it slows a pilot down.
5. **CCS is proprietary and patented.** One vendor, commercial terms, not an OSS parser you can fork. We think the report you just read is the honest case for why that trade is worth it — a maintained single source of truth with a cross-runtime oracle — but it is a trade, and you should weigh it as one.

---

7. Working with b3u.dev

The engagement template is deliberately small:

1. **Scope one interface** — one message family from your estate, your segment usage, your quirks — and we author the grammar variant.
2. **You provide the oracle**: your existing conformance/regression corpus (synthetic or de-identified). Acceptance is your suite passing, plus the byte-identity oracle across every runtime you care about.
3. **You get the component set**: the generated frontend (CLI + C-ABI shared library), the five SCB APIs' keyword tables for your grammar, and reference readers in your stack's languages — the same deliverables §4 printed.

Licensing is subscription-based through b3u.dev, with production and OEM/embedding terms under a separate written agreement. The demo in this report, including the five-way oracle harness, runs end to end in one command on delivered artifacts — ask for a walkthrough.

Contact: support@b3u.dev

---

message content shown is synthetic; no PHI was used anywhere in this work. HL7® is a registered trademark of Health Level Seven International.