TidyEDI

X12 EDI · DEFENSIVE PARSING

Finding the Elusive ISA Line

A 1979 standard, fixed at 106 bytes, meets senders who strip it, prepend to it, and re-encode it. The byte offsets don't survive that. Neither does the regex you'd reach for next. Here is what does — and how it names every deviation it steps over.

Engineering note x12-tidy / isa ~2,400 words PDF

THE PROBLEM

01 The ISA line is load-bearing

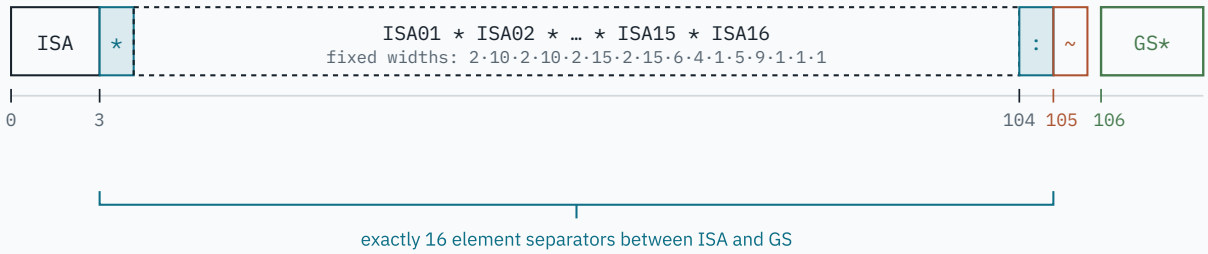
An ANSI X12 interchange is a stack of envelopes: `ISA` wraps one or more `GS` functional groups, each wrapping `ST` transaction sets, closed in reverse by `SE`, `GE`, `IEA`. The very first segment, `ISA`, is the one every parser has to read before it can read anything else: it is a fixed 105-byte record whose byte positions *declare the delimiters* for the entire rest of the file — element separator at byte 3, component separator at byte 104, segment terminator at byte 105.

The standard is unusually strict about this. Every element in the ISA is a fixed width. The segment is exactly 105 bytes plus its terminator. The functional-group header that follows, `GS` + the element separator, therefore begins at byte 106 — always, by rule.

So the naive reader is one line: take `data[106:109]`, check it spells `GS*`, and slice the ISA at fixed offsets. It works on conformant files. It fails on a large fraction of real ones, because the audience for this tool is a developer holding a file their trading partner sent that their parser just rejected, and they need to know *precisely* what is wrong with it so they can push back or handle it. That reframes the job: **parse permissively — locate the envelope even when it is**

malformed — then emit a diagnostic for every deviation rather than failing on the first.

FIG. 1 — ANATOMY OF A CONFORMANT ISA LINE



The ISA segment is positional, not delimited-and-free: **16 fixed-width elements**, a component separator, a one-byte terminator, and then `GS*` at byte 106. x12-tidy returns the run from `ISA` up to (not including) that `GS` — terminator and any trailing bytes included; splitting the run into elements comes later.

THE OBVIOUS ALTERNATIVE

02 Why a regex can't do it

The fixed-offset reader from §1 fails on real files. The next instinct — and it is the right instinct to have — is a pattern: match the identifier, capture whatever byte is acting as the separator, hop over sixteen fields, then find the header.

THE PATTERN YOU WOULD REACH FOR

```
rb"ISA(.)(?:.*?\1){15}.*?GS\1"
#      ^ capture the separator — (.) then \1 matches any byte literally,
#        metacharacters included, so it need not be hard-coded
```

It compiles. It even matches a clean file. It is still the wrong tool, for four reasons — and the first one is fatal on its own:

- **A match cannot carry the reason.** The product of this tool *is* the explanation: the terminator was `\r\n`, forty newlines were appended, the file opened with a BOM, there were seventeen separators because the sender's ID contains a `*`. A regex returns `Match` or `None` — a non-match gives you no position and no cause. You would end up writing one pattern per deviation and branching on which matched: a hand-rolled parser, with worse ergonomics than a real one.

- **"Exactly sixteen" is expressible; the ambiguity under it is not.** You *can* force the count — `(?:(?!\1).)*\1` for "one field, then the separator," sixteen times — and the capture handles any separator byte without hard-coding it. But that assumes every separator is a field boundary, and in the case x12-tidy exists for, one isn't: a sender whose ID holds the separator byte produces a segment that is genuinely ambiguous — sixteen fields or seventeen, indistinguishable from the bytes alone. A regex resolves that silently, matching *some* boundary; the trailing `GS\1` then anchors early, or the match fails, with no signal either way. x12-tidy does the opposite: it counts the separators, sees seventeen, and reports it (`isa.separator-count-high`), naming both possible causes. And `GS\1` matches a `GS` + separator *anywhere* — inside ISA06 or ISA08 data as readily as in a transaction set. A match gives you a boundary; it never gives you "this boundary is suspect, and here is why."

- **Choosing among candidates is a search, not a match.** When the bytes `ISA` turn up in junk, the logic is: try this offset; if the run is not a valid ISA line, anchor on the next `ISA`; if none work, report the *first* candidate's failure. That is a stateful search where every failure is inspected. A regex backtracks inside the engine and hands you nothing to look at.

- **The counted, alternation-heavy construct backtracks catastrophically.** On a large file with a near-miss, the very pattern that expresses "sixteen separated fields" — nested quantifiers over `.*?` or `(?:(?!\1).)*` — is the one that

explodes. And the O(1) shortcut — is `GS` already at byte 106? — has no regex equivalent; the engine scans from the start every time.

The function that replaces the pattern is about forty lines. It never backtracks, it takes the O(1) shortcut when the file is clean, and every branch that tolerates a non-conformance carries the sentence that explains it. The rest of this note is that function.

FIELD CONDITIONS

03 What senders actually send

Every one of the following has been observed in production traffic. Each one breaks a fixed-offset reader, and several break a delimiter-first reader too.

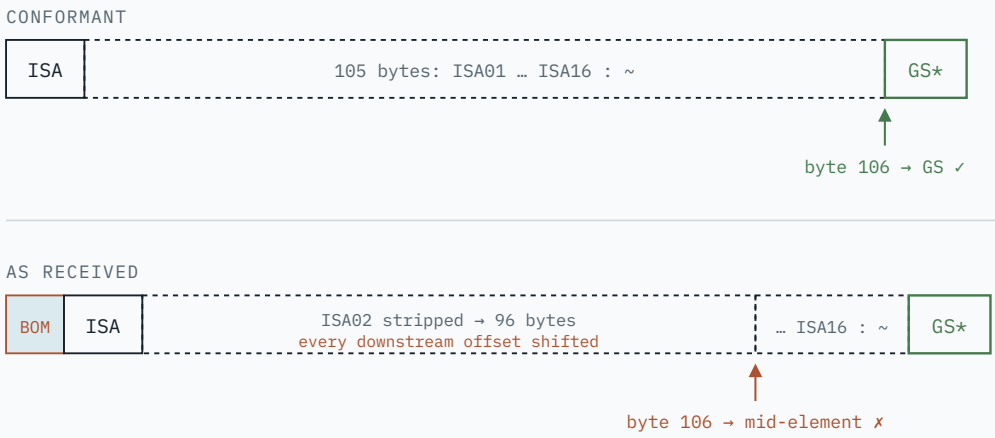
- **Leading bytes before `ISA`.** A UTF-8 byte-order mark (`EF BB BF`), SMTP or HTTP headers left in by a gateway, a one-line "generated by…" banner. Byte 0 is no longer `I`.
- **Stripped empty elements.** A sender that omits blank trailing elements ships an ISA segment shorter than 105 bytes. Every fixed offset after the first gap is now wrong, including the byte-106 assumption for `GS`.
- **Appended newlines.** `CR LF` after *every* segment terminator — a pretty-printer or an FTP transfer in text mode. `GS` is pushed to byte 108.
- **Two-byte terminators.** The segment terminator is `\r\n`, or `~\r`, so the "one byte at 105" model is off-by-one for the whole file.
- **Non-uppercase identifiers.** `isa`, or `Isa` — a system that lower-cased the payload somewhere in transit. A case-sensitive `find(b"ISA")` reports an empty file.
- **Wide encodings.** The file is UTF-16; the identifier is `I 00 S 00 A 00`. Nothing matches `ISA`.

- **The element separator inside element data.** A sender whose ID contains `*` while using `*` as the separator. The segment now has 17 separators and no single correct parse.

- **A `GS` + separator lookalike.** A `REF*GS*…` element deep in a transaction set when the real functional-group envelope is missing — or a sender ID ending in `GS` right inside ISA06. Either matches `b"GS" + sep` before, or instead of, the real header.

- **The bytes `ISA` in junk.** An email subject line reading `SUBJECT: ISA FILE`, a path like `/feeds/ISA/…`. The first match is not the segment.

FIG. 2 — WHY FIXED OFFSETS BREAK



Three prepended bytes and one omitted element are enough. The fixed-offset reader looks at byte 106 and finds element data; it either rejects a valid interchange or, worse, mis-reads the delimiters and corrupts everything downstream. **The byte position is not an invariant. The `GS` header that follows the ISA line is.**

WHY UTF-16 IS FATAL, NOT TRANSCODED

Every other row in that table is a malformed X12 interchange — the bytes are still X12, just not conformant ones, and locating the ISA line tolerates that. A UTF-16-encoded file isn't a malformed interchange; it isn't X12 bytes at all. The standard predates Unicode and is single-byte-per-character throughout — there is no legal X12 interchange that is anything else. Decoding one would mean guessing the variant (LE or BE), trusting or inferring a byte-order mark, and re-encoding the whole file before parsing has even started. That is exactly the kind of guess this tool refuses to make everywhere else in this note — an ambiguous separator count gets reported, not resolved; a truncated element gets padded by rule, not inferred. The interleaved-NUL pattern (`I\x00S\x00A` , or the big-endian mirror) is cheap and unambiguous to detect, so x12-tidy names it and stops (`isa.identifier-utf16` , "re-export the file") rather than attempt a conversion that could silently mis-decode and hand back a diagnosis of the wrong bytes.

SCOPE

04 One job: return the run

Locating the ISA line does exactly one thing: given the raw bytes, return the run that starts with `ISA` and ends immediately before `GS` + the element separator. It does *not* validate the delimiters, the element widths, the terminator, or the element content — that all comes later, and none of it can run until the run has been located.

But a run has to clear a minimum bar to *be* an ISA line at all. Three checks, no more:

- **It begins with `ISA`** (any case).
- **It ends immediately before `GS` + the element separator.**
- **It holds exactly 16 element separators** — `ISA` then ISA01…ISA16.

WHY THE BAR IS HERE AND NOT LATER

A run that fails any of these is not an ISA line, and it is reported *fatal and terminal* — it does not go to recovery. More than 16 separators is unrecoverable by definition: a segment whose separator appears in its own data has no unambiguous parse. And you cannot parse delimiters out of a run that does not have 16 elements — there is nothing coherent to parse.

Everything *past* the bar — is `ISA05` a valid qualifier, are the fixed widths right, is the terminator a legal byte — is decided downstream. Locating the line hands forward a run with the right shape; whether it also has the right meaning is someone else's job.

THE APPROACH

05 Five techniques

5.1 Anchor on `GS`, not on a byte number

The end of the ISA line is wherever the `GS` functional-group header begins. Byte 106 is used only as a shortcut that produces the identical answer when the file happens to be conformant; when it doesn't, the code searches for the header by content.

`find` is not a validator, though. `GS` + the separator can occur *inside* the ISA line — a sender ID (ISA06) or receiver ID (ISA08) ending in `GS`, followed by the element separator, is enough. When the fast-path offset check has already failed and `find` lands on one of those, the anchor is too early. That is caught in 5.2, not here.

X12_TIDY/ENVELOPE/ISA/ISA_LINE.PY · _TRY_CANDIDATE, STEPS B–C

```
# b. the element separator is, by rule, the 4th byte of the ISA segment
element_separator = cleansed[3:4]
gs_identifier     = GS_IDENTIFIER + element_separator

# c. find where the ISA line ends == where the GS segment starts
if hay[STANDARD_GS_OFFSET:STANDARD_GS_OFFSET + 3] == needle:
    gs_pos = STANDARD_GS_OFFSET          # fast path: GS at the standard offset
else:
    gs_pos = hay.find(needle)
    if gs_pos == -1:
        return _Attempt(None, Diagnostic(
            Code.ISA_GS_NOT_FOUND,
            f"no {gs_identifier!r} functional-group header after the ISA "
            f"segment; cannot locate the end of the ISA line.",
            offset=isa_start,
        ))
```

5.2 Require *exactly* 16 separators — not "at least"

Once a candidate `GS` is found, the bytes before it are counted. An early version accepted `>= 16`. That let three false anchors through silently: a stray `GS*` deep in transaction data, leading junk that ended in `ISA*`, and a `GS*` sitting inside the ISA line's own ISA06/ISA08 data. Requiring the count to be *exact* catches all three — and the diagnostic names the structural fault, not the count:

- a `GS*` matched **inside** the ISA line (5.1) cuts the run short → fewer than 16 separators → `isa.separator-count-low`;
- a `GS*` matched **past** the real header (downstream, or a decoy `ISA*` prefix) overshoots → more than 16 separators → `isa.separator-count-high`.

X12_TIDY/ENVELOPE/ISA/ISA_LINE.PY · _TRY_CANDIDATE, STEP D

```
# d. the run must hold exactly 16 element separators
isa_line        = cleansed[:gs_pos]
separator_count = isa_line.count(element_separator)

if separator_count < ISA_ELEMENT_SEPARATORS:
    return _Attempt(None, Diagnostic(Code.ISA_SEPARATOR_COUNT_LOW, ...))
if separator_count > ISA_ELEMENT_SEPARATORS:
    return _Attempt(None, Diagnostic(Code.ISA_NO_FUNCTIONAL_GROUP, ...))

return _Attempt(isa_line, None)          # a real ISA line
```

5.3 Try every `ISA`, keep the first that parses

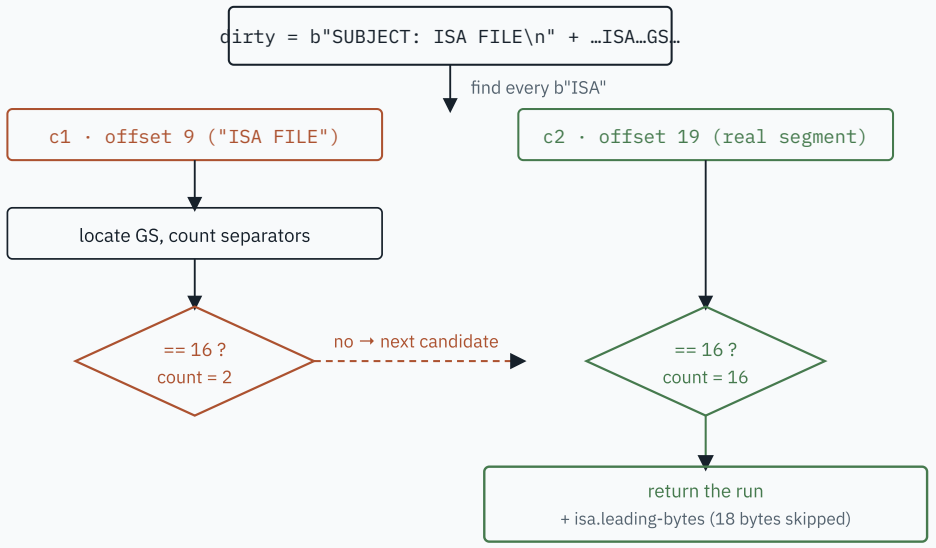
Because `ISA` can appear in leading junk, the code collects *every* occurrence (capped, against a file that is mostly the bytes `ISA`) and tries each. The first candidate whose run clears the bar wins; the bytes before it become an `isa.leading-bytes` warning. If none clear it, the first candidate's failure is what gets reported.

X12_TIDY/ENVELOPE/ISA/ISA_LINE.PY · _ISA_OFFSETS & THE CANDIDATE LOOP

```
def _isa_offsets(haystack: bytes, identifier: bytes = ISA_IDENTIFIER) -> list[int]:
    offsets: list[int] = []
    at = haystack.find(identifier)
    while at != -1 and len(offsets) < MAX_ISA_CANDIDATES:
        offsets.append(at)
        at = haystack.find(identifier, at + 1)
    return offsets

# _try_all: first clean run wins; else the first failure is remembered
for isa_start in offsets:
    attempt = _try_candidate(dirty, isa_start, case_insensitive=case_insensitive)
    if attempt.isa_line is not None:
        return IsaLineResult(attempt.isa_line, isa_start,
                             _context_diagnostics(dirty, isa_start, ...)), None
    if first_failure is None:
        first_failure = (isa_start, attempt.failure)
```

FIG. 3 — MULTI-CANDIDATE ANCHORING



The **exactly-16 gate** is what makes the retry safe: a candidate anchored in junk almost never has 16 separators followed by a `GS`, so it fails and the search moves on. When every candidate fails, the first failure — not a guess — is reported.

5.4 Case-insensitive, but only after the fast path fails

Matching `ISA` case-insensitively means lower-casing the whole buffer — an allocation the size of the file. The common case (an uppercase identifier that parses) must not pay for it. So the structure is two-phase: try the exact-uppercase candidates first, touching nothing; only when all of them fail take one `lower()` copy and retry. This also rescues a valid lowercase segment sitting *behind* junk that contains the literal uppercase word `ISA` — the earlier design missed that, because it only fell back when no `ISA` existed at all.

X12_TIDY/ENVELOPE/ISA/ISA_LINE.PY · EXTRACT_ISA_LINE, THE TWO PHASES

```
def extract_isa_line(dirty: bytes) -> IsaLineResult:
    # Fast path: exact uppercase ISA identifiers. Never copies the buffer.
    upper = _isa_offsets(dirty, ISA_IDENTIFIER)
    result, upper_failure = _try_all(dirty, upper, case_insensitive=False)
    if result is not None:
        return result

    # No uppercase identifier parsed. UTF-16 only matters when there is none at all.
    if not upper and any(m in dirty[:_UTF16_SCAN_LEN] for m in _UTF16_MARKERS):
        return IsaLineResult(None, -1, [Diagnostic(Code.ISA_IDENTIFIER_UTF16, ...)])

    # One full-buffer lower-case copy, only on this already-failed path.
    lowered = dirty.lower()
    if ISA_IDENTIFIER.lower() in lowered:
        ci_offsets = _isa_offsets(lowered, ISA_IDENTIFIER.lower())
        result, ci_failure = _try_all(dirty, ci_offsets, case_insensitive=True)
        if result is not None:
            return result
    ...
```

One guard keeps the fallback honest: the string `isa` occurs inside ordinary words. A lowercase candidate is only reported as a identifier if it sits where a segment could start.

X12_TIDY/ENVELOPE/ISA/ISA_LINE.PY · _LOOKS_LIKE_SEGMENT_START

```
def _looks_like_segment_start(dirty: bytes, offset: int) -> bool:
    """Start of file, or right after a non-alphanumeric byte. Distinguishes a
    real lowercase `isa*` from `isa` buried in a word like "advisable"."""
    return offset == 0 or not dirty[offset - 1 : offset].isalnum()
```

"BUT THIS IS EXACTLY WHAT `RE.IGNORECASE` IS FOR"

A flag makes the *matching* case-insensitive; it does nothing about the rest. A case-insensitive regex over a 2 MB buffer still visits every byte — no cheaper than `lower()`, and with no way to skip the work when the file is already uppercase. It matches `isa` inside *advisable* just as eagerly, with no signal to separate that from a real lowercase identifier. And it still produces a match, not the `isa.identifier-lowercase` diagnostic that tells the developer their partner lower-cased the payload. The flag saves the one line that was never the problem.

5.5 Name every deviation

Nothing above is a silent repair. Each tolerance emits a stable diagnostic code — `isa.leading-bytes`, `isa.identifier-lowercase`, `isa.identifier-utf16`, `isa.separator-count-high`, `isa.gs-not-found`, and the rest — so the developer holding the bad file gets the exact list of what their partner did wrong, not a single exception at the first surprise.

VALIDATION

06 Proving it with hostile input

The step was validated against 103 adversarial inputs across two sweeps — NUL bytes and `0xFF` as separators, a 2 MB junk prefix, UTF-16 LE and BE, a PDF header, 2,000 appended newlines, a file that is nothing but the bytes `ISA`. Zero crashes. Every returned run was checked against a single invariant:

TESTS/TEST_ISA_LINE.PY · _ASSERT_CONTRACT

```
def _assert_contract(dirty: bytes, r: IsaLineResult) -> None:
    if r.isa_line is None:
        return
    assert r.isa_line[:3].upper() == b"ISA"
    cleansed = dirty[r.isa_start:]
    assert cleansed.startswith(r.isa_line)
    assert cleansed[len(r.isa_line):][:2].upper() == b"GS"
    # and, from the caller: r.isa_line.count(sep) == 16
```

INPUT	RESULT	DIAGNOSTICS
UTF-8 BOM, then a clean interchange	106-byte run	`isa.leading-bytes`
`SUBJECT: ISA FILE\n` + lowercase interchange	run returned	`isa.identifier-lowercase`, `isa.leading-bytes`
ISA02 & ISA04 stripped (86-byte segment)	86-byte run	none — still 16 separators
No `GS` envelope; a `REF*GS*` deep in the data	fatal	`isa.separator-count-high`
Two interchanges concatenated, first `GS` missing	2nd interchange	`isa.leading-bytes`
UTF-16 LE encoded file	fatal	`isa.identifier-utf16`
Element separator `*` occurs inside the sender's ID	fatal	`isa.separator-count-high`
2 MB of leading junk, then `ISA`	106-byte run	`isa.leading-bytes`

THE SEAM

07 Where content validation takes over

Locating the ISA line returns a run with the *shape* of an ISA line. It does not know whether the run has the *meaning* of one — whether `ISA01` is a real authorization qualifier, whether the fixed widths line up, whether the delimiters are legal bytes. That comes next, and it is a genuine backstop. Those Pesky Delimiters picks up here, with the four delimiters.

The one place the seam is visible: leading junk shaped *exactly* like an ISA line — the bytes `ISA`, sixteen element separators, then `GS` + that separator, across at least 109 bytes. It clears all three checks, so the run is returned — correctly, by this stage's own contract — and the element-level validation downstream is what rejects it, because the "elements" are the wrong widths and carry nonsense. This is the layering doing its job: shape here, meaning next.

The weaker and far more common form — junk that merely *ends* in `ISA*` — never reaches that seam. It carries no run of sixteen separators, so the exactly-16 gate rejects it and the retry moves to the real segment.

THE PATTERN

Parse permissively and report strictly. Anchor on a structural invariant — the header that must follow — not on a byte position the sender can shift. Validate the anchor with a hard count, so a false match fails loudly instead of passing quietly. Retry from the next candidate rather than giving up. And prove all of it against inputs written specifically to break it.