

Source: cppcc docs/AI_Limits_For_CCS_Parsing_Model.md @ 6d4c0df — curated 2026-09-06; edits land in the source first.

AI Limits for the CCS Parsing Model

A Plain-English Account of a Clean-Room Experiment

Experiment designed and run by hand by A. F. Urakhchin. This account written by Claude Code (Claude Fable 5.1) in the cppcc working session, from the experiment's own files, at the author's request. September 4, 2026.

If you can program a little, you can follow this. The question is simple: give an AI a list of requirements, and does it come back with the invention those requirements were written from? We measured it. The answer is no, and the interesting part is why.

1. What Is This Really?

1.1 The Question We Were Trying to Answer

The Compiler Compiler System, CCS, is a tool that reads a grammar and produces a compiler. Its first version was written in C in 1991. Its C++ version was written in 2010, alongside the patent for its parsing model, which was granted in 2013. Since May 2026 it has been extended with Claude Code, an AI coding agent.

That last fact prompted an accusation on a programming forum: that the whole thing, code and text alike, was made by AI. The honest reply was a dated history. But the accusation raised a question worth answering properly, and not with a denial: **could an AI have invented the CCS Parsing Model?**

You cannot answer that by asking the AI, because the AI that works on CCS every day has read all of it. So we built a clean room.

1.2 What the CCS Parsing Model Is, in One Paragraph

Almost every parser generator you have heard of, from YACC to ANTLR, hands you a **parse tree**: a nested structure, one node per grammar rule that matched, with the matched text hanging off the leaves. Your program then walks the tree. CCS does something different. While parsing, it fills in a small set of tables, indexed the way the grammar itself is organized: every grammar symbol keeps a list of all the places it was recognized, every identifier in the source keeps an index, and each recognition records the values it saw and which other recognitions it invoked. There is no tree to walk. You look things up. This in-memory form is called the runtime; a flat binary copy of it, called the binary, can be read from any programming language; and the source text can be regenerated from either. The published overview paper lists six ways this departs from the YACC lineage. We will come back to those six.

1.3 Why the Answer Matters

If a fresh AI, given only the requirements CCS satisfies, produces the CCS model, then the model is a consequence of its requirements and anyone could have derived it. If the AI produces something else that meets the same requirements, then the model is a choice that the requirements do not force, and the choice is the invention. Either answer is worth having, and either one is a better reply to "AI built it" than any promise.

2. The Experiment, Step by Step

2.1 The Brief

The only input to every run was one file, `brief.md`, about 350 words. It asks for an architecture-level design of a parser generator for data formats, and it sets seven hard requirements. In plain words:

1. The parse output must be usable from five languages, C++, Java, Python, JavaScript and Rust, without parsing the text again.
2. From that output, every one of the five languages must be able to reproduce the original text byte for byte. Nothing the grammar saw may be lost.
3. A consumer works from the parse output alone. No access to the parser process, no hand-written knowledge of the format beyond the grammar.
4. The five consumers are checked against each other with a plain `diff` of their outputs, not with a suite of test cases.

5. The generator is built in its own framework: its grammar language has a grammar, the parser for it is generated by the tool, and regenerating it must give an identical result.
6. No semantic actions in the grammar. One grammar serves all five languages.
7. Parsing is predictive with one token of lookahead. Reading the output, not parsing, is the performance-critical path.

The brief also carries one process rule: nobody will answer questions. If something is unclear, decide, state the assumption, and continue. The full text is in Appendix A. It contains no CCS vocabulary, no product names, no reference to any existing tool.

2.2 What "Clean" Means for the AI

Claude Code keeps a persistent memory for each project, and it decides which project you are in by the folder you start it from. Start it inside the CCS repository and it loads years of context: the project instructions file, the memory directory, the whole history of the work. Start it in an empty folder somewhere else and it has none of that. It is the same model, but it knows nothing about this project.

So the experiment lives in a folder that is not under any CCS repository and has no instructions file above it:

```
~/experiments/pm_probe/
```

Each run was started from its own sub-folder, which also means each run had its own empty memory. After the first run we checked its memory directory: it was empty. The runs could not leak into each other.

2.3 What "Clean" Means for the Inventor

The harder problem was the human. The person running the experiment invented the model, and one hint in a follow-up question would contaminate everything. The fix was to remove the inventor from the loop entirely:

- One message per run, always the same: Read `brief.md` and produce the deliverable.
- No follow-up questions answered, ever. The brief says so, and the runs obeyed it.
- Nobody read any run's output until all five existed.

- The scoring was done by a sixth fresh session, not by a person, and not by the session that works on CCS.
- Only after scoring did the outputs come into the CCS working session for comparison. That is where this document was written.

One honest residue remains and is recorded here: the seven requirements were written by the CCS working session, from the design. The experiment measures whether those requirements suffice. It cannot measure whether they were the right ones to write down. Section 6 returns to this.

2.4 The Directory

```
~/experiments/pm_probe/
├─ brief.md           the seven requirements, the only input
├─ PROTOCOL.md        read by the human only, never given to a run
├─ run_1/
│   ├─ brief.md       identical copy
│   └─ design.md       what the first fresh session produced
├─ run_2/ ... run_5/   same shape
├─ score/
│   ├─ brief.md
│   ├─ design_1.md ... design_5.md  copies of the five outputs
│   ├─ task.md         the scoring instructions
│   └─ scores.md        the sixth session's verdict
└─ COMPARISON.md       the informed comparison, written last
```

Nothing in this folder is under version control.

2.5 The Five Runs

Five times, in a plain terminal:

```
cd ~/experiments/pm_probe/run_N
claude
> Read brief.md and produce the deliverable.
```

Each session read the brief, wrote a design, and saved it as `design.md`. Some wrote the file on their own; that was the only variation in procedure. The first run happened to be started from the top-level folder and its output was moved into `run_1/` afterwards, a filing difference and nothing

more. The session logs that Claude Code keeps on its own confirm six separate sessions in six separate project folders.

Each design is a real architecture document: data-structure layouts with byte offsets, a grammar language with examples, a parser algorithm, a bootstrap procedure, a correctness check, and the two required closing sections, "what each requirement forced" and "what I rejected and why." They range from 19 to 38 kilobytes of text. Nobody edited them.

The runs, in order:

run	output at (Sep 4)	size (design)	session log (kept by CC)	host language the design chose
1	07:50	19 KB	186 KB	C++
2	08:35	36 KB	419 KB	Rust
3	08:48	38 KB	184 KB	Rust
4	08:58	30 KB	184 KB	Rust
5	09:11	33 KB	208 KB	Rust
S	09:24	20 KB	677 KB	(the scorer)

2.6 Why Five and Not One

One run cannot tell you whether the requirements forced a design or whether that particular session happened to land there. Five runs from identical input show whether the outcome is stable. If all five agree, the requirements did the work. If they scatter, the brief was underspecified. If one differs in kind, that one is the interesting case and the scoring step tells you which requirement pushed it there.

2.7 The Prediction, Written Down Before Reading

Before any output was read, the CCS session recorded this prediction, so it would count as a prediction and not a rationalization:

Most runs will pick a serialized parse tree, or a tree plus a symbol table, as the central output. Each will satisfy the easy constraints and quietly weaken a hard one. Zero or one run out of five will replace the tree with a flat, position-indexed model that is the same shape as the grammar's own runtime structure.

3. What the Five Designs Looked Like

3.1 The Shape They All Chose

All five designs chose the same central structure, and it is worth understanding it, because it is a good design. Here it is in plain words.

The parser reads the input and writes one output file. That file contains the input, unchanged, byte for byte. In front of the input (or after it, the order varies) sits an array of small fixed-size records, one per grammar rule that matched and one per token. Every record says three things: **what kind** of thing it is (which rule or which token), **which bytes** of the input it covers (a start and a length), and **how far** its sub-structure extends in the array. The records are laid out in the order a predictive parser discovers them, parent before children.

A tiny example. Grammar: a file is a list of key=value lines. Input, six bytes: a = 1 and a newline.

index	kind	start	len	next	covers
0	file	0	6	8	
1	pair	0	6	8	
2	KEY	0	1	3	"a"
3	WS	1	1	4	" " (whitespace kept)
4	EQ	2	1	5	"="
5	WS	3	1	6	" "
6	VAL	4	1	7	"1"
7	NL	5	1	8	newline

To find the children of record 1, start at record 2 and keep jumping by each record's next until you reach 8. To get the text of any record, slice the input at start for len bytes. To reproduce the input, walk the leaves in order and copy their slices. No pointers, no objects, no decoding. In C++ and Rust the array is viewed in place; in Java through a byte buffer; in JavaScript as a typed integer array; in Python as a memory view. That is how one file serves five languages without anyone parsing again.

This is a lossless concrete syntax tree, stored flat. It is the structure behind tools like tree-sitter and the Roslyn compiler. It is the textbook answer for a serious engineer, and all five fresh sessions found it.

3.2 Where They Differed

D	record size	names in the file?	how children are reached	how the lexer handles data
1	16 bytes	kind names only	jump by subtree size	modes declared on rules
2	16 bytes	full schema with labels	jump by subtree size	parser tells lexer what to expect
3	24 bytes, parent link	full schema with labels	jump, plus a parent pointer	parser-directed + parameters
4	16 bytes + a slot table	kind table with labels	a SECOND array of child slots, looked up	modes computed from expected sets
5	16 bytes	nothing but a hash	jump by next index	modes driven by tokens

Design 4 is the one that differs in kind rather than in detail: instead of walking siblings to reach the third child, it keeps a second array that says, for each parent, exactly which record is child number three. We look at it closely below, together with design 3, the one the scorer ranked first.

3.3 Design 3 in Detail

Design 3 calls its grammar language GDL. A grammar file has four kinds of declaration: token (a leaf), skip (whitespace or comments, kept as leaves but ignored by structure), param (a lexical parameter, a few bytes wide, bound during the parse), and rule (an interior node). Its HL7 example shows the idea that makes it stand out:

```
grammar hl7 ;

param FS  : bytes(1) ;           // field separator, bound from MSH-1
param ENC : bytes(4) ;           // component, repetition, escape, sub

token MSH  = "MSH" ;
token CR   = "\x0d" ;
token FSEP = $FS ;
token REP  = $ENC[1] ;
token COMP = $ENC[0] ;
token TEXT = not($FS, $ENC[0], $ENC[1], "\x0d")+ ;
token NAME = alpha alpha (alpha | digit) ;
```

```

rule message = msh:head segment:seg* ;
rule msh     = MSH bind(FS) bind(ENC) (FSEP field:f)* CR ;
rule segment = NAME:name (FSEP field:f)* CR ;
rule field   = rep:r (REP rep:r)* ;
rule rep     = comp:c (COMP comp:c)* ;
rule comp    = TEXT:text? ;

```

Clinical messages declare their own delimiters inside the message. Every classic parser generator handles that with a semantic action that reaches into the lexer, which requirement 6 forbids.

Design 3's answer is `bind(P)`: consume exactly the parameter's width as a leaf, and from then on any token whose definition mentions `$P` uses those bytes. The generator checks statically that every parameter is bound before any token that needs it can be expected. No code runs; the grammar declares, the engine executes.

Its second idea is that the lexer never guesses. Because the parser is LL(1), before each token it knows exactly which tokens may come next, and it hands that set to the lexer. Keywords are contextual for free; a TEXT and a NAME may overlap lexically and never collide, because they are never expected at the same place.

Its record is wider than the others, 24 bytes, and carries a parent index, so a consumer can go up as well as down without keeping a stack. Every output file also carries the full symbol and shape tables copied from the compiled grammar, so a reader with no generated code at all can navigate by name and label and print the canonical dump.

Its meta grammar, the grammar of GDL written in GDL, is shown in full. Tokens omitted, as in the original:

```

grammar meta ;

skip WS      = (" " | "\x09" | "\x0d" | "\x0a")+ ;
skip COMMENT = "//" not("\x0a")* ;
token KW_GRAMMAR = "grammar" ; token KW_TOKEN = "token" ; token KW_SKIP = "skip" ;
token KW_PARAM = "param" ; token KW_RULE = "rule" ; token KW_BIND = "bind" ;
token KW_BYTES = "bytes" ; token KW_NOT = "not" ; token KW_ANY = "any" ;
token CLASS = "alpha" | "digit" | "hex" | "printable" ;
token IDENT = alpha (alpha | digit | "_")* ;
token STRING = "\"" (not("\\"", "\\") | "\\" any)* "\"" ;
token INT = digit+ ;
token LPAR = "(" ; token RPAR = ")" ; token BAR = "|" ; token STAR = "*" ;
token PLUS = "+" ; token QMARK = "?" ; token EQ = "=" ; token SEMI = ";" ;
token COLON = ":" ; token COMMA = "," ; token DOLLAR = "$" ;

```

```

token LBRACK = "[" ; token RBRACK = "]" ;

rule file      = KW_GRAMMAR IDENT:name SEMI decl:d* ;
rule decl      = token_decl | skip_decl | param_decl | rule_decl ;
rule token_decl = KW_TOKEN IDENT:name EQ lalts:body SEMI ;
rule skip_decl  = KW_SKIP  IDENT:name EQ lalts:body SEMI ;
rule param_decl = KW_PARAM IDENT:name COLON KW_BYTES LPAR INT:width RPAR SEMI ;
rule rule_decl  = KW_RULE  IDENT:name EQ alts:body SEMI ;

rule alts      = seq:s (BAR seq:s)* ;
rule seq       = item:i* ;
rule item      = atom:a (COLON IDENT:label)? suffix:card? ;
rule atom      = IDENT:ref | STRING:lit | LPAR alts:group RPAR
                | KW_BIND LPAR IDENT:param RPAR ;
rule suffix    = STAR | PLUS | QMARK ;

rule lalts     = lseq:s (BAR lseq:s)* ;
rule lseq      = litem:i+ ;
rule litem     = latom:a suffix:card? ;
rule latom     = STRING | KW_ANY | CLASS | KW_BYTES LPAR INT RPAR
                | DOLLAR IDENT:param (LBRACK INT:index RBRACK)?
                | KW_NOT LPAR latom:x (COMMA latom:x)* RPAR
                | LPAR lalts:group RPAR ;

```

The design then argues, correctly, that this is LL(1): the set of tokens that can start an `item` is disjoint from the set that can follow a `seq`, and the optional label and the optional suffix are introduced by different tokens.

Its bootstrap is the neatest of the five. Because the generator reads grammars as tree files, the seed it ships is not a parser but a tree file of the meta grammar, produced once by a throwaway hand-written parser that is then deleted. Rebuilding from scratch is: generate tables from the seed tree, parse the meta grammar with those tables, compare with the seed. The fixed-point check in continuous integration is a recursive `diff` of the freshly generated meta artifacts against the shipped ones.

The scorer ranked design 3 first: all seven requirements met, with the strongest treatment of requirements 3 and 5.

3.4 Design 4 in Detail

Design 4 is the one that differs in kind. Its summary sentence tells you why: "the node kinds in the image are the production numbers of the LL(1) table, so the tree is literally the parser's prediction trace." Every time the parser consults its prediction table and picks a production, it appends a

record of that production's number. There is no separate tree-building step, because prediction is the tree-building step.

Its grammar language is small and its rules for tree shape are mechanical. Each alternative of a rule gets its own kind, named R.0, R.1, and so on, so a record's kind tells you both the rule and which alternative was taken. Each element of an alternative is a **slot** at a fixed position. An optional element that is absent is slot reference zero. A repeated element is a list node. Groups never vanish. Whitespace is not a separate leaf: each leaf records where its leading trivia ends and its own text begins.

The output file has two integer arrays over the input bytes. The first is the record array. The second, `refs[]`, is the slot table: for each interior record, one entry per slot naming the child record, and for lists a count followed by the elements. Reading `file.entries[1].value` in any language is three loads and a slice:

```
n = refs[nodes[1].ref_base]           the list node
e = refs[nodes[n].ref_base + 1 + 1]   element 1 (count comes first)
v = refs[nodes[e].ref_base + 2]       slot 2 of that entry
bytes[nodes[v].text_start .. nodes[v].end)
```

No sibling walk, ever. The price is that the writer must fill in the slot table after each subtree closes, and that the record holds a pointer into the second array rather than a subtree count.

Its meta grammar, abridged by its author with the interesting part in full:

```
grammar pg;
trivia WS, COMMENT;
token WS      = [ \t\r\n ]+;          token COMMENT = "#" [^\n]*;
token KW_GRAMMAR = "grammar"; token KW_TOKEN = "token"; token KW_RULE = "rule";
token KW_START = "start"; token KW_TRIVIA = "trivia"; token KW_BYTES = "bytes";
token IDENT    = [A-Za-z_] [A-Za-z0-9_]*; # after keywords: keywords win ties
token STRING   = "\"" ([^"\\] | "\\" [^])* "\"";
token CLASS    = "[" ([^\\] | "\\ [^])* ";
token INT      = [0-9]+;
token LP="("; token RP=")"; token STAR="*"; token PLUS="+"; token QUERY="?";
token BAR="|"; token COLON="."; token EQ="="; token SEMI=","; token COMMA=",";

rule file      = KW_GRAMMAR name:IDENT SEMI decls:decl* EOF;
rule decl      = token_decl | rule_decl | start_decl | trivia_decl;
rule token_decl = KW_TOKEN name:IDENT EQ body:texpr SEMI;
rule rule_decl  = KW_RULE name:IDENT EQ body:alts SEMI;
rule start_decl = KW_START name:IDENT SEMI;
```

```

rule trivia_decl= KW_TRIVIA first:IDENT rest:(COMMA IDENT)* SEMI;
rule alts      = first:seq rest:(BAR seq)*;
rule seq      = items:item*;
rule item     = head:IDENT labeled:(COLON atom)? rep:repeat?
               | lit:STRING rep:repeat?
               | LP group:alts RP rep:repeat?;
rule atom     = IDENT | STRING | LP alts RP;
rule repeat   = STAR | PLUS | QUERY;
rule texpr    = first:tseq rest:(BAR tseq)*;
rule tseq     = items:titem+;
rule titem    = atom:tatom rep:repeat?;
rule tatom    = STRING | CLASS | KW_BYTES LP INT RP | LP texpr RP;
start file;

```

The item rule is a small lesson in what requirement 6 costs. The natural way to write "an element with an optional label" is (label:IDENT COLON)? atom, and that is not LL(1): an identifier could begin either the label or the element. The LL(1) form reads an identifier first, then an optional colon and atom. Whether that identifier was a label or a reference is not decided in the grammar at all. It is decided later, in ordinary host code, by asking whether the optional part is present. The grammar records what was seen; the consumer decides what it means. That is requirement 6 taken seriously, and it is also exactly how CCS treats the boundary between syntax and semantics.

Its bootstrap is a three-stage fixed point. Stage 0 uses a hand-written parser once. Stage 1 regenerates tables and accessors with stage 0's output. Stage 2 regenerates with stage 1's. The check is that stages 1 and 2 are byte-identical, in all five languages' accessor files. Then the hand-written parser is deleted.

The scorer placed design 4 last, on one reading of requirement 2: because trivia is folded into each leaf's span, the boundaries between individual whitespace or comment tokens are not recorded. The scorer itself noted that under the other reading design 4 ranks second. Of the five, it is the one closest to the CCS idea that a record is a recognition of a production with fixed slots.

4. The Judge: A Sixth Fresh Session

4.1 The Scoring Task

The five outputs were copied into score/ as design_1.md through design_5.md, with the brief and a scoring task, task.md. The task was written by the CCS session, but with the same

discipline as the brief: no CCS vocabulary, no hint of what a "right" answer looks like. It asks the scorer to:

- mark each requirement, for each design, as Met, Weakened or Missed, with one sentence of evidence;
- describe each design's central data structure in its own words, without reusing the design's name for it;
- say which requirement cost each design the most;
- group the five by central structure, rank them, and say whether any one differs in kind rather than in detail.

A sixth session was started in `score/`, again with one message, `Read task.md and do what it says`. It wrote `scores.md`.

4.2 What the Scorer Decided

The scorer first stated its own assumptions, five of them. The one that mattered was about requirement 2: it read "no information the grammar saw may be lost" to include the boundaries of whitespace tokens, and it applied that reading consistently.

Its per-requirement verdicts:

requirement	D1	D2	D3	D4	D5
1 five lang	Met	Met	Met	Met	Met
2 byte-for-	Met	Met	Met	Weakened	Met
3 alone	Met	Met	Met	Met	Met
4 diff	Met	Met	Met	Met	Met
5 self-host	Met	Met	Met	Met	Met
6 no action	Met	Met	Met	Met	Met
7 LL(1)/hot	Met	Met	Met	Met	Met

Its grouping: all five belong to one family, "one self-contained binary file holding the verbatim input plus a flat depth-first array of fixed-width integer records, each with a kind and a byte span, with leaves tiling the input." Within the family, two shapes: designs 1, 2, 3 and 5 encode the tree by nesting; design 4 stores structure in a separate child-slot table. Its ranking: 3, 2, 1, 5, 4, with the note that design 4 would be second under the other reading of requirement 2.

4.3 What the Scorer Found That Nobody Asked For

Reading blind, the scorer found a real defect in design 5. That design keeps no names in its output and finds a labelled child by scanning the parent's children for a kind. If a rule has two children of the same kind, the first optional, then when the optional one is absent the accessor for the second returns the wrong thing. That is a valid LL(1) grammar the design's own author had not considered. The scorer did not downgrade the design for it, since no requirement covers it, but it recorded it as a caveat. This is the kind of thing a blind reviewer is for.

5. My Comparison

This section is the work of the CCS working session, the one that knows the model, done only after `scores.md` existed. Its full form is `COMPARISON.md` in the experiment folder.

5.1 What I Compared Against

The overview paper on CCS lists six ways the system departs from the YACC and ANTLR lineage. Stated plainly:

- (i) The parse tree is replaced by a model of four entities: Context, Name, Symbol and Rule.
- (ii) Semantics consume an interface derived from the grammar; no actions in the grammar.
- (iii) Parsing is two-phase: a mutable runtime is built first, and a binary is produced from it afterwards; the two convert both ways.
- (iv) Regenerating the source text from the parsed form is a first-class operation, not a debugging aid.
- (v) The runtime and binary interfaces are derived from the grammar, not hand-written.
- (vi) The grammar language is defined in itself and the tool regenerates itself from that definition.

5.2 The Scorecard

departure	runs	verdict
(i) tree replaced by the four-entity model	0 of 5	NOT forced
(ii) no actions, grammar-derived interface	5 of 5	forced
(iii) mutable runtime, then bidirectional bin	0 of 5	NOT forced

(iv)	regenerate source as a first-class op	5 of 5	forced, WEAKLY
(v)	interfaces derived from the grammar	5 of 5	forced
(vi)	self-hosting, regenerates itself	5 of 5	forced

Three departures fall out of the requirements cleanly. One, (iv), falls out only in a weak form explained in 5.4. The two that do not fall out are the two that define the model, and they are linked.

The prediction from 2.7 held, at its pessimistic end: not one run in five, but zero.

5.3 What All Five Did Without Being Told

The convergence is itself a result. Every one of the five, independently:

- chose one parser and five **readers**, and explicitly rejected generating a parser in each language, on the grounds that five parsers are five ways to disagree and reading is the hot path;
- chose a table-driven interpreter rather than generated parser source (CCS emits a C++ parser; the fixed-point requirement pushed the fresh designs toward tables as data);
- chose a flat, fixed-width, little-endian record array read in place, and rejected JSON, Protocol Buffers, FlatBuffers and Cap'n Proto, with reasons that match the ones CCS gives;
- embedded the input bytes in the output rather than referencing a file;
- made a canonical byte-level dump plus `diff` the whole oracle, with a fixed-point regeneration as its third leg;
- put a grammar hash in the header so a reader refuses a foreign file;
- made LL(1) conflicts errors, never resolved by declaration order.

Three of five copied the grammar-derived symbol tables into every output so the file describes itself. That is the closest any run came to the CCS binary's grammar-driven layout, and it is the schema, not the instances.

5.4 Three Things True of All Five

The second question the author asked was concrete: for each design, what is the parse result, what is the serialized form, what is the interface in each language, and how is the round trip done? The full per-design table is in `COMPARISON.md`, section 8. Three facts hold across all five and answer the question directly.

There is no in-memory parse result. Not one design has a runtime form that the parser builds and later serializes. The parser writes the file records as it goes, filling in extents afterwards. The file is the only artifact. In the paper's terms, the compile phase and the runtime-to-binary phase are one pass. The paper argues that there must be no direct arrow from grammar to binary, because the parser needs a form it can revise. All five draw that arrow. For a data format, that is correct and simpler. That is departure (iii) missing, seen from the other side.

Every interface is a view over the file. Two layers in each language: a hand-written, grammar-independent kernel of three to four hundred lines (open, node fields, children, bytes, validate, dump), and a generated, grammar-specific typed layer of file-and-index wrappers with one accessor per labelled child. This is the analogue of the CCS binary readers plus the generated keyword modules, and it is the closest match to CCS in the whole experiment. The difference: it navigates by position in the record array. The CCS interface looks things up by name, by symbol, and by rule index.

Every "decompile" is a copy, not a generation. All five store the input verbatim inside the output and reproduce it by concatenating leaf spans that point back into that block. Design 5's production version is a single slice of the whole block. Remove the embedded text and every one of the five loses the ability to reproduce the source. CCS regenerates source text from the parsed structure by the grammar; its binary does not carry the source bytes; and its round trip is compile, decompile, compile again, and compare the two outputs. None of the five does anything like that. This is why (iv) is marked "forced, weakly."

5.5 What Is Missing, in Plain Words

The four-entity model's primary operations are: find every identifier in a scope, by text or by number; find every grammar symbol in a scope; find every recognition of a given symbol; all in constant time. None of the seven requirements asks for any of these. Requirement 7, "reading is the hot path," was read by all five as "navigation must be index arithmetic," which a flat tree satisfies. It was never read as "looking up by symbol or by identifier must not require a walk."

6. Why the Requirements Were Not Enough

6.1 The Brief Said "Data Formats"

The brief's title is "a parser generator for data formats," and its examples are clinical messages, EDI interchanges, fixed-layout records. That framing came from the current commercial focus of the project. Data formats have no identifiers, no scopes, no forward references. Name and Context exist in the CCS model because the 1991 and 2010 tools parsed programming languages, where the parser must build a symbol table while parsing and must be free to revise it. That freedom is what the runtime phase is for, and it is exactly why the paper forbids the direct grammar-to-binary arrow.

So the brief carried a second contamination, opposite in sign to the one recorded in 2.3. The first is that the inventor wrote the requirements. The second is that he, and the session that drafted them, wrote them from where the project is now rather than from where the model came from, and that removed the one problem that motivates half the model.

6.2 Requirement 2 Invited the Copy

Requirement 2 says: reproduce the text byte for byte; no information the grammar saw may be lost. The cheapest design that satisfies that sentence is to store the text and lay records over it, and all five found it. The sentence never says "without the source text" and never asks that the regenerated text compile back to the same output. As written, requirement 2 selected *for* a concrete-syntax-tree overlay, the opposite of what the experiment wanted to test.

6.3 Requirements Describe Behavior; the Model Is a Choice of Being

This is the fundamental point, and the answer to the third question the author asked: what is there in the CCS Parsing Model that a list of requirements cannot reach?

Every requirement in the brief, and every one proposed since, is **behavioral**: it says what must hold of the output. The CCS model is **ontological**: it says what the output is made of. Symbol owns all of its recognitions. Name is a first-class entity with two indexes. The derivation is secondary, reachable but not primary. That is the grammar's own shape imposed on the parsed instance, so the program is indexed by the grammar rather than by the text.

A tree with enough side tables satisfies any finite list of behavioral requirements. That is the mechanism, and it answers the author's second half-question, whether it would take an unlimited number of iterations to reach the model by adding requirements. The answer is worse than unlimited. Each added requirement is cheapest to satisfy by attaching one more index to the tree. The process converges, quickly, to "tree with N annexes," which is behaviorally equivalent to CCS and is not CCS. The tree is an attractor, and adding requirements cannot leave it, because the

requirement that would leave it is "the output is not a tree," and that is not a requirement. It is the answer.

Is this an AI limitation? Not specifically. A human engineer given the brief would also have produced the tree. Tree-sitter, Roslyn and the whole lossless-syntax-tree literature did. What the experiment measured is that the design is not compressible to seven behavioral constraints, and would not be compressible to ten. The content of the invention is exactly what the constraints do not carry. One more thing a requirement cannot carry is history: in CCS, compile-and-decompile began in 1991 as a way to test a compiler in a world without test frameworks, and the method hardened into the architecture. Path dependence of that kind has no requirement form either.

6.4 What Would Break the Attractor

Three kinds of input would, and none of them is a requirement:

- **An instance.** One worked example of the four-entity layout for a small grammar. A fresh session would reconstruct the rest, but that is copying, not deriving.
- **A prohibition.** "The primary structure of the output must not be a tree." Cheap, but it smuggles the answer in.
- **The workload.** A brief written from the consumer's queries rather than from the parser: "design the data structure that a semantic pass over a programming language reads, in five languages, without the source text, such that the regenerated source recompiles to an identical structure." The parser becomes whatever fills it.

The third is the legitimate next experiment, because it tests the claim in 6.3 rather than assuming it.

7. What This Means

7.1 For the "AI Built It" Question

Measured, not asserted: given the requirements CCS satisfies, a fresh AI rebuilds the *consequences* of the CCS model, four of six of them, and does not rebuild the model. The model is the part it does not produce. That is a better answer to the accusation than any denial, and it is also a fair account of the last four months: the extensions since May 2026, the tokenizer sets, the test suites, the sibling language runtimes, the documents, are exactly the kind of work that follows from the model once the model exists.

7.2 What Might Come Next

Two follow-up rounds are drafted in `COMPARISON.md`, neither run yet.

Round two keeps the requirement-list method and corrects the brief: requirement 2 becomes "the output must not contain the source text; regenerate the text from the output alone; the regenerated text, compiled again, must give an identical output"; and two requirements are added, constant-time lookup by identifier and by grammar symbol for programming-language grammars, and a parser that may revise what it has recorded before the output is final. Prediction: the tree survives as primary, with indexes beside it, and the decompiler becomes a grammar-driven generator.

Round three abandons the requirement list and gives the workload, as in 6.4. Prediction: a by-symbol, by-name indexed store in most runs, with the tree as an annex or absent. If that prediction fails, the honest public statement becomes: a fresh model reconstructs the consequences of the CCS model from its requirements, and reconstructs the model itself only from the workload it was built for, or from an example of it.

7.3 What Could Go Wrong

The experiment has known weaknesses, listed here so that nobody has to discover them:

- The brief was written by the CCS session from the design. A differently phrased brief might behave differently. Round two tests exactly one such rephrasing.
- The scorer is the same model family as the designers. It was fresh and blind, but it shares their defaults.
- Five samples from one model on one day. Enough to show the outcome is stable, not enough to put a number on it.
- The comparison in section 5 was done by the session that knows the model. Its reading of the five designs is informed, which is the point, and is therefore not blind.

None of these change the headline. All five designs are kept, unedited, and are available on request (support@b3u.dev) to anyone who wants to read them and disagree.

Appendix A. The Brief, Verbatim

```
# Design brief: a parser generator for data formats
```

You are asked to design, at the architecture level, a parser generator. Do not write the implementation yet; produce the design: the central data structures, what the generator emits, how the pieces relate, and what the primary correctness check is. Then explain which parts of the design each requirement forced.

The tool reads a grammar for a data format (examples: a clinical message standard, an EDI interchange, a fixed-layout record description, a small domain-specific language) and generates a parser for that format.

Hard requirements:

1. The output of a parse must be consumable in five programming languages (C++, Java, Python, JavaScript, Rust) without re-parsing the text in each language.
2. That output must reproduce the original text byte for byte, in every one of the five languages. No information the grammar saw may be lost.
3. A consumer in any of the five languages works from the parse output alone. It has no access to the parser process and no hand-written knowledge of the format beyond the grammar.
4. Agreement between the five consumers is verified by a plain `diff` of their outputs, not by a suite of semantic test vectors.
5. The generator is written in the same framework it generates: its own grammar-description language has a grammar, the parser for that grammar is generated by the tool, and regenerating it must give an identical result.
6. No semantic actions in the grammar. One grammar serves all five languages; application logic lives outside it.
7. Parsing is predictive with one token of lookahead. Reading the parse output, not parsing text, is the performance-critical path.

Deliverable: the design, in prose and diagrams, followed by a section "what each requirement forced" and a section "what I rejected and why".

Process rule: nobody will answer questions during this task. If something is ambiguous, decide it yourself, state the assumption explicitly, and continue. Produce the complete deliverable in one pass.

Appendix B. The Protocol, Verbatim

```
# Protocol (read by the human only, never pasted into a run)

1. Five runs. Each: fresh directory run_N/ containing a copy of
   brief.md only; `claude` started there; ONE message:
   "Read brief.md and produce the deliverable." No other input.
   If the session asks anything: "Decide yourself and state the
   assumption."
2. Save each output as run_N/design.md. Do not read any output until
   all five exist.
3. Blind scoring: a sixth fresh session in score/ with brief.md and
   the five design.md files, asked to score each design against the
   seven requirements only, and to classify its central data
   structure in its own words.
4. Only then: bring the five designs and the scores into a cppcc
   session for comparison with the published overview paper.
```

Appendix C. Per-Design Summary

D	parse result	serialized form	round trip as designed
1	none in memory; records written straight to file	.pt: kind names, 16-byte records, then the input	leaf-span concat via accessors, diff input; five dumps diff; meta regenerate diff
2	none; engine writes, back- patches size	.ptree: input, 16-byte records, SYMBOLS, diagnostics	leaf copy, diff input; generic + bindings dumps diff; two-stage fixpoint
3	none; 24-byte records with parent index	input, 24-byte records with parent, SYMTAB copied in	restore = leaf concat, cmp; ten dumps diff; diff -r meta + re-parse
4	none; records + refs[] back-filled per subtree	.pgi: kind table, records, refs[], input	leaf concat, diff; dumps pairwise; typed vs generic; 3-stage fp
5	none; records written	.cst: hash, 16-byte records, input; no names at all	unparse = one slice of the text; cmp; dumps vs Rust; fixpoint cmp

In every row the interface per language is the same shape: a fixed kernel plus a generated typed layer of file-and-index views.