Source: cppcc docs/ccs_json_and_benchmarking.md @ 78a2674 — curated 2026-08-06; edits land in the source first.

Compiler Compiler System: JSON and benchmarking

Draft 2.0 — August 2026 (successor of the Draft 1.0 docx/pdf; restructured around the b3u.dev delivery flow)

The Compiler Compiler System (CCS) takes grammars written in SGDL (Source Grammar Definition Language) and produces compilers. This article walks the JSON use case end to end from a subscriber's seat: download the compiled deliverables from b3u.dev, run the JSON frontend on real-world inputs, build a real C++ program against the delivered SCB library, and measure it honestly against nlohmann/json and simdjson.

1. The JSON grammar in SGDL

```
(jsonp
  (jsonp ::=
    0 = "  METAACTBEG();" =
    value
    0 = "  METAACTEND();" =
  )

  (value ::=
    object | array | str | num | trueLit | falseLit | nullLit
  )

  (str       ::= stringToken )
  (num       ::= integerToken )
  (trueLit   ::= 'true' )
  (falseLit  ::= 'false' )
  (nullLit   ::= 'null' )

  (object   ::= '{' [ members ] '}' )
  (members  ::= pair { restPair } )
  (restPair ::= ',' pair )
  (pair     ::= str ':' value )

  (array     ::= '[' [ elements ] ']' )
```

```
  (elements  ::= value { restValue } )
  (restValue ::= ',' value )
)
```

The grammar's name is `jsonp`; the file is `json.sgr`. A guided tour of the rules — and of the two SGDL authoring habits they teach — is in the ccs_json User Guide; this article takes the grammar as given and focuses on what you do with its compiled results.

2. Getting the deliverables

Compiling `json.sgr` on b3u.dev (upload it to a workbench unit — here the unit is named `json` — validate, compile) yields the use case's deliverables on the downloads page. Three zips matter:

```
ccs_json-bundles.zip                          the per-language SCB
                                              material + the SCB C++
                                              library
ccs_json-frontend-linux-x86_64.zip            the compiled JSON
                                              frontend
ccs_json_bench-code.zip                       this article's C++
                                              program (scb_counts.cc
                                              + makefile)
```

Download them (they land in `~/Downloads`), then lay out the working directories:

```
mkdir -p ~/ccsUseCases/ccs_json
cd ~/ccsUseCases/ccs_json
mv ~/Downloads/ccs_json-bundles.zip .
mv ~/Downloads/ccs_json-frontend-linux-x86_64.zip .
unzip ccs_json-bundles.zip
unzip ccs_json-frontend-linux-x86_64.zip

mkdir -p ~/ccsUseCases/ccs_json_bench
cd ~/ccsUseCases/ccs_json_bench
mv ~/Downloads/ccs_json_bench-code.zip .
unzip ccs_json_bench-code.zip
```

After unzipping, `~/ccsUseCases/ccs_json` holds:

```
ccsjson              the compiled frontend for the json unit
json.sgr.fgr         the grammar binary (portable decimal-text form)
json.sgr.bfgr        the grammar binary (raw native-byte-order form)
README.md, MANIFEST.json, NOTICE.txt, PROVENANCE.txt
cpp/                 generated C++ sources for the grammar +
                     jsonKeyWordDefinition.h + grammar binaries
python/  js/  rust/  the per-language KeyWordDefinition modules +
                     grammar binaries
scbcpp/              the SCB C++ library: include/ headers +
                     lib/libscbcpp.a
```

and `~/ccsUseCases/ccs_json_bench` holds `scb_counts.cc` and its `makefile`.

3. Running the frontend on real-world JSON

The standard JSON benchmark corpus — `twitter.json` and `citm_catalog.json` from the simdjson project's `jsonexamples/`, `canada.json` from `miloyip/nativejson-benchmark` — makes a good end-to-end check. Download the three files into a run directory:

```
mkdir -p ~/ccsUseCases/ccs_json_bench/run
cd ~/ccsUseCases/ccs_json_bench/run
# fetch twitter.json, citm_catalog.json (simdjson jsonexamples/)
# and canada.json (nativejson-benchmark data/) from those projects'
# public repositories
```

Compile each with the frontend:

```
$ ~/ccsUseCases/ccs_json/ccsjson canada.json
$ ls -la canada.json*
canada.json          2.25 MB   the input
canada.json.bfgr     13.4 MB   the SCB instance, raw form
canada.json.fgr      11.4 MB   the SCB instance, decimal-text form
canada.json.lis                compile listing
canada.json.log                log
canada.json.urt                runtime decompilation (readable)
```

```
Input               Size      .fgr (text)   .bfgr (raw)   errors
twitter.json        632 KB    2.26 MB       2.35 MB       0
```

```
citm_catalog.json   1.7  MB   4.14 MB       5.32 MB       0
canada.json         2.25 MB   11.4 MB       13.4 MB       0
```

The `.urt` file is the runtime decompilation — a textual rendering of what the parser saw. For canada.json it round-trips to JSON-shaped text identical (modulo whitespace) to the input, with full numeric fidelity within double precision.

There are two binary forms. The `.fgr` is decimal-text: one value per line. The `.bfgr` is a raw native-byte-order dump of the same contents; it exists purely so a downstream runtime can load the parsed form fast — 20-30x faster than `.fgr` on the reference box. Ship `.fgr` across machines of unknown byte order and repopulate `.bfgr` locally with the frontend's `-b` flag.

4. What the frontend is for

`ccsjson` plays the same role for JSON that the CCS toolchain plays for SGDL: it takes a textual input matching the grammar and writes a binary representation (.fgr / .bfgr). That binary is the canonical artifact the rest of an application consumes — load the `.bfgr` once, walk it many times, without ever re-tokenising the original JSON. This separates "parse once" from "process many times", which is also the natural granularity for benchmarking against general-purpose JSON libraries.

5. Building scb_counts

`ccs_json_bench-code.zip` delivered two files into `~/ccsUseCases/ccs_json_bench`:

- `scb_counts.cc` — a complete program that loads a `.bfgr` once, then runs three walk strategies over it, timing each and tallying six leaf counts (s/i/r/t/f/n = string, int, float, true, false, null). All three walkers must agree — they oracle each other.
- `makefile` — builds it against the SCB C++ library delivered in `ccs_json-bundles.zip`:

```
CCS_JSON_HOME ?= $(HOME)/ccsUseCases/ccs_json
SCBCPP        ?= $(CCS_JSON_HOME)/scbcpp
CPP_BUNDLE    ?= $(CCS_JSON_HOME)/cpp

CXX      ?= g++
CXXFLAGS ?= -std=c++17 -O2 -Wall -fno-strict-aliasing

scb_counts: scb_counts.cc
```

```
    $(CXX) $(CXXFLAGS) -I$(SCBCPP)/include -I$(CPP_BUNDLE) \
      scb_counts.cc $(SCBCPP)/lib/libscbcpp.a -o scb_counts
```

```
$ cd ~/ccsUseCases/ccs_json_bench
$ make
$ ./scb_counts run/canada.json.bfgr 100
```

A small sanity input shows the shape of the output (and is easy to verify by hand — object keys are deliberately not counted as strings):

```
$ printf '{"hello":"world","n":42,"arr":[1,2,3,true,null,"x"],"nested":
{"a":1.5,"b":[]}}\n' > sample.json
$ ~/ccsUseCases/ccs_json/ccsjson sample.json
$ ./scb_counts sample.json.bfgr 100
loaded: sample.json.bfgr (240 longs), iterations: 100
  Helper           0.46 us/walk  (s=2 i=4 r=1 t=1 f=0 n=1)
  Helper2          0.23 us/walk  (s=2 i=4 r=1 t=1 f=0 n=1)
  Helper3          0.02 us/walk  (s=2 i=4 r=1 t=1 f=0 n=1)
OK: all three walkers agree
```

6. Benchmarking against nlohmann/json and simdjson

The reference measurements loop each parser N times on a sample input and report the mean microseconds per call, in two phases: parse/load (once-per-input cost) and walk+tally (per-iteration cost over the already-parsed structure). Numbers from the reference box (g++ -O2, 100 iterations); the nlohmann/json and simdjson baselines are reproducible from those projects' own public releases.

6.1 Parse / load (us per parse-or-load)

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

Reading these honestly:

- `.fgr` (decimal-text records) is slower than nlohmann's full text parse — the file is 2-5x larger than the source JSON and each value goes through stream extraction. That is the worst end of the spectrum.
- `.bfgr` cuts CCS load time by 20-30x and lands within ~2x of simdjson DOM — which is itself a pure in-memory parse with no file I/O; most of the remaining gap is the file read.
- `.bfgr` is bigger than `.fgr` on disk; the win is I/O and parse cost, not byte count.

6.2 Walk + tally (us per traversal of the already-parsed tree)

The three walk strategies in scb_counts.cc, against the same tallying loops written for nlohmann and simdjson:

- **Helper** — recursive traversal through per-symbol wrappers; the baseline.
- **Helper2** — same recursion, flat vectors, inlined child access, no per-call bounds checks; ~2x the speed of Helper.
- **Helper3** — grammar-aware: three of the six counts read in O(1) from per-symbol totals; the rest from a targeted scan of the four value positions. No tree walk at all.

```
Input              nlohmann  simdjson    CCS   CCS2   CCS3
twitter.json        125 us    55 us    437 us 236 us  11 us
citm_catalog.json   226 us   172 us   1215 us 691 us  31 us
canada.json         585 us   589 us   3259 us 1553 us 119 us
```

Helper3 beats the simdjson DOM walk on every sample — 5-6x — and the counts agree exactly across all five implementations on every sample, so the speedups are not an artifact of silently dropped cases.

7. Why the grammar-aware walk wins

Three things stack, and they are complementary rather than alternatives.

simdjson must visit every value; Helper3 doesn't. simdjson's tape stores values in document order, so "how many floats are in this JSON" can only be answered by iterating every value and switching on its type. The SCB is organised the other way: every match the parser made is grouped under its symbol's entry, and the per-symbol total is one field read. For canada.json, one read replaces 111,080 leaf visits.

The string-count subtraction trick. Object keys are also strings, so the raw string total over-counts. A structural identity fixes it: every pair contributes exactly one key string, so value strings = total strings − total pairs. Both terms are O(1). This is the kind of identity you can only spot knowing the grammar shape — the generic walkers can't.

The targeted true/false/null scan is O(value-positions), not O(tree). Those literals have no per-symbol totals of their own, but they can only appear where the grammar puts a JSON value — four positions across the rules. Scanning only those slots costs the same order of iterations as simdjson's leaf count on the worst input, but each iteration is one memory read, one tag decompose, and one comparison, with a tight early exit for the common case.

Trade-offs, so this isn't oversold: the parse-time cost is paid up front (one-shot, simdjson is still faster end to end); the Helper3 shortcuts are grammar-specific by construction (Helper2 is the generic fallback, ~2x slower than simdjson's walk); and different queries flip the comparison — a document-order tape is optimal for traversal, a symbol-indexed SCB is optimal for type-aggregate queries, because the parser already did the grouping work that a tape walk re-does at query time.

8. The SCB C++ API

Everything scb_counts.cc does rides on one delivered header, `SyntaxControlledBinary.h`, from the SCB C++ library (namespace `ccsscb`, linked as `libscbcpp.a` — standalone, no other CCS libraries involved).

SyntaxControlledBinary is the SCB instance itself: a single `memory_` block of tagged values plus a header. Loading is one call — `readBinaryRaw(path)` for `.bfgr`, `readBinary(path)` for `.fgr` — and both fill the same in-memory contents. Every element of the parse is reachable from the header's root tag; a tag decomposes with `tag::setlongedf(&t, kkk, nnn)` into a symbol id (`kkk`, which keyword of the grammar) and an instance number (`nnn`, which occurrence). The generated `jsonKeyWordDefinition.h` from the cpp bundle is the symbol table that gives those ids names — `KW_PAIR`, `KW_ELEMENTS`, `KW_STRINGTOKEN`, and so on — so walk code reads as grammar vocabulary, not numbers.

Helper builds, once per loaded binary, a navigable index over that flat memory: for each grammar symbol a `kwn` entry (with the per-symbol total `l` — the field Helper3's O(1) counts read), and under it one `edp` entry per instance, exposing the instance's `fixed()` children (the rule's fixed positions) and `dynamic()` children (the iteration part). Recursive walks like scb_counts.cc's first strategy are a switch on `kkk` plus recursion into fixed and dynamic children.

Helper2 holds the same index as parallel flat vectors — `kwns_[kkk]` and `edps_[kkk][nnn]` — with inlined child access and no per-call bounds checks: the shape for tight, trusted read loops. **Helper3** is the same layout under a separate name, so grammar-aware shortcut algorithms can be paired with it without entangling them with the generic walker. The three strategies in scb_counts.cc are exactly this progression, and the program's closing assertion — all three agree — is the pattern to keep when you write your own: let the generic walk oracle the clever one.

One practical note, visible at the top of scb_counts.cc: the generated KeyWordDefinition header also serves frontend builds, so its tail declares a factory type the standalone SCB library does not carry; a three-line forward declaration satisfies that never-called declaration and keeps the delivered header untouched.