Source: ccs_json docs/ccs_json_user_guide.md @ 5d0c2a2 — curated 2026-08-06; edits land in the source first.

ccs_json User Guide — a JSON compiler from a 26-line grammar

This guide walks the complete ccs_json loop on b3u.dev: take the `json.sgr` grammar, compile it on the site into a working JSON compiler (`ccsjson`), download the result, and work with it locally on your own machine. Every step below is performable with exactly what the site delivers — nothing here requires tooling you don't have.

1. What ccs_json is

`json.sgr` is a grammar for JSON written in SGDL (Source Grammar Definition Language), the input language of the CCS (Compiler Compiler System) toolchain. Compiling the grammar on b3u.dev produces a **target compiler frontend**: a native binary, `ccsjson`, that reads JSON files and writes each one as a CCS **SCB instance** — a syntax-controlled binary that downstream code loads and walks without ever re-tokenising the original JSON.

The division of labour:

- **b3u.dev** compiles grammars. You never install the CCS compiler-compiler; the site runs it for you.
- **Your machine** runs the resulting frontend and consumes the SCB instances it produces, using the delivered SCB-API packages.

2. The grammar

```
(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 )
)
```

Reading it as an SGDL author:

- The grammar's name is `jsonp` — the first identifier. It names the generated artifacts (and the frontend binary's `ccs<name>` form comes from the grammar FILE stem, here `json` → `ccsjson`).
- `stringToken` and `integerToken` are predefined lexical tokens. A single numeric token covers both integers and floats — the tokenizer distinguishes them internally, so one grammar rule handles `-36000` and `-65.6136` alike.
- `'true'` / `'false'` / `'null'` are quoted literals, making them keywords of the target language.
- Alternation lists (`a | b | c`) take rule names. That is why each literal and predefined token gets a one-line wrapper rule (`trueLit ::= 'true'`, `str ::= stringToken`): the wrapper gives the alternative a name that can sit in the `value` list.
- One optional `[ ]` or iteration `{ }` per rule body: nested patterns are factored into named rules. JSON's "zero-or-more comma-separated pairs" becomes the three-rule shape `object` / `members` / `restPair` instead of one nested expression.

These two habits — wrapper rules for alternation, factoring around `[ ]` / `{ }` — carry to any grammar you write in SGDL.

3. Compiling on b3u.dev

1. Sign in and open the workbench.
2. Create a unit and upload `json.sgr` as its grammar.
3. Validate, then compile. The unit's page reports the compile result; a grammar error comes back with the compiler's own diagnostic tail, so you can fix and re-upload.
4. Open the downloads page. Two deliverables matter here: - **Compiled frontends** — the frontend zip for your unit, `<unit>-frontend-linux-x86_64.zip`, built on demand. - **Language bundles** — the per-grammar SCB-API material (`<stem>-bundles.zip`) with per-language directories.

4. The frontend zip, locally

Unzip the frontend delivery. Its contents:

```
ccsjson              the compiled frontend for your grammar
json.sgr.fgr         the grammar binary (portable decimal-text form)
json.sgr.bfgr        the grammar binary (raw form, the BUILD host's
                     byte order)
README.md            usage, flags, platform notes
MANIFEST.json        delivery provenance
NOTICE.txt           license notice
PROVENANCE.txt       build provenance
```

Run it on any JSON file:

```
$ ./ccsjson demo.json
```

This compiles `demo.json` and writes the SCB instance files beside the input:

```
demo.json.fgr        the SCB instance, portable decimal-text form
demo.json.bfgr       the SCB instance, raw native-byte-order form
                     (20-30x faster to load; not endian-portable)
demo.json.lis        compile listing
demo.json.urt        runtime decompilation — a human-readable
                     rendering of what the parser saw
demo.json.log        log
```

Two more flags you will use:

```
$ ./ccsjson -b file.fgr     populate file.bfgr from a portable .fgr
                            in YOUR machine's native byte order —
                            ship .fgr across hosts, run -b locally
$ ./ccsjson -h              the full flag list
```

Platform: the binary is a dynamically linked linux-x86_64 ELF; the zip's README states the glibc floor it was built against. If it does not run on your target, contact support@b3u.dev.

Sanity check on real-world JSON

The standard JSON benchmark corpus (twitter, citm_catalog, canada — from the simdjson and nativejson-benchmark projects) compiles clean:

```
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` decompilation of canada.json round-trips to JSON-shaped text identical (modulo whitespace) to the input, with full numeric fidelity within double precision.

5. Consuming SCB instances from your code

The `.bfgr` your frontend writes is the artifact the rest of your application consumes: load it once, walk it many times — "parse once, process many times". The loaders live in the SCB-API packages delivered on the downloads page (python / js / rust / cpp), and the language bundles carry the per-grammar material each package needs to resolve your grammar's symbols:

```
cpp/      generated parser+generator sources for your grammar,
          plus the grammar binaries
python/   jsonKeyWordDefinition.py  + grammar binaries
js/       jsonKeyWordDefinition.mjs + grammar binaries
rust/     jsonKeyWordDefinition.rs  + grammar binaries
```

The KeyWordDefinition module is your grammar's symbol table — it is what lets an SCB-API runtime decompile and navigate the instances YOUR frontend produced, by name (`jsonp`, `pair`,

`elements`, …) rather than by number.

A companion document (the ccs_json benchmarking use case) shows real C++ code working against the SCB C++ API on this exact grammar — including walk strategies and the performance numbers they earn — and is the recommended next read once the loop above runs on your machine.

6. License and support

Development and evaluation use is covered by your b3u.dev subscription (Terms of Service §5.3). PRODUCTION deployment of components produced with the frontend requires a separate written agreement — contact support@b3u.dev BEFORE deploying (Terms §5.4). The NOTICE.txt in every delivery states the same.

Questions, platform issues, feedback: support@b3u.dev.