Source: authored in b3u_use_cases as this use case's build guide.

Building the MeTTa frontend

This guide walks the complete `metta` loop on b3u.dev: take the `metta.sgr` grammar, compile it on the service into a working MeTTa frontend, download the result, and run it on your own machine against your own `.metta` files.

Every step is performable with the artifacts published beside this guide. Nothing here requires a local CCS installation — the compiler runs on the service; what you build locally is the consumer side.

For what the grammar actually does and why MeTTa needs three token classes beyond SGDL's predefined four, see the companion document *MeTTa on CCS*.

1. What you start with

`metta.sgr` — 49 lines of grammar plus a header comment. It describes MeTTa's s-expression core: a program is a sequence of s-expressions, an s-expression is an atom or a parenthesised list, and an atom is one of the token classes or standalone terminals MeTTa uses.

⚠ 2. One thing to know before you compile

This grammar must be compiled with the MeTTa tokenizer set. On the b3u.dev workbench that is the tokenizer selection; on a command line it is the `-lt` flag.

The reason matters, and it will save you an afternoon: the tokenizer set governs the lexical rules used to read *the grammar file itself*, not only the language it describes. Under the MeTTa tokenizer, comments begin with `;` — not the `--` used by ALGOL-family grammars.

A grammar file carrying `--` comments compiled under `-lt` does not report a syntax error in its listing. The comment text is lexed as MeTTa symbols, the rules absorb them, and the compile fails later with an internal range error that says nothing about comments. If you adapt this grammar, keep the `;`.

3. Compile it on b3u.dev

Create a unit named `metta`, upload `metta.sgr`, and compile it with the MeTTa tokenizer selected.

Name the unit `metta`. The generated files take their names from the grammar's first identifier, so a unit named `metta` produces `mettaParser`, `mettaGenerator` and `mettaKeyWordDefinition` — and the paths in the rest of this guide line up. A different unit name still works; you will just be adjusting filenames as you read.

A successful compile produces:

- the **frontend** — a native `ccsmetta` executable that reads `.metta` sources and writes CCS SCB binaries;
- the **language bundles** — the generated runtime definitions for C++, Python, JavaScript and Rust, so a consumer in any of those can interpret what the frontend emits;
- the **grammar binaries** — `metta.sgr.fgr` and `metta.sgr.bfgr`.

4. Lay it out locally

```
mkdir -p ~/ccsUseCases/metta
cd ~/ccsUseCases/metta
mv ~/Downloads/metta-bundles.zip .
mv ~/Downloads/metta-frontend-linux-x86_64.zip .
unzip -q metta-bundles.zip
unzip -q metta-frontend-linux-x86_64.zip
chmod +x ccsmetta
```

You now have the frontend — named `ccsmetta`, the service prefixes `ccs` to the grammar name — a `cpp/` directory holding `mettaKeyWordDefinition.h`, sibling directories for the other languages, and an `scbcpp/` directory containing the standalone SCB C++ library — `include/` and `lib/libscbcpp.a`.

5. Run it

```
cat > hello.metta <<'EOF'
; a MeTTa source file
(= (greet $who) (say "hello" $who))
(greet &self)
EOF
```

```
./ccsmetta -lt hello.metta
```

That produces `hello.metta.bfgr` — the compiled binary form — alongside `.fgr`, `.lis` and `.urt`. The `.urt` is a readable decompilation and is the quickest way to see that the frontend understood your input.

Note what the sample exercises: `$who` is a variable, `&self` is a space reference, `=` is a standalone terminal, `"hello"` is a string token, and `;` starts a comment. Those are the MeTTa-specific lexical classes, all in three lines.

⚠ 5.1 The `-lt` is required here too, and omitting it fails quietly

`-lt` selects the MeTTa tokenizer for reading **your `.metta` sources**, exactly as it did for the grammar in §2. It is a separate occasion, and the frontend does not infer it.

Omit it and the run **succeeds**. Exit status 0. A `.bfgr` is written. No error is reported anywhere. What actually happened is that the default tokenizer read until it met something it did not recognise and stopped — for the sample above, after the first comment line:

```
$ ./ccsmetta hello.metta          # no -lt
$ wc -c hello.metta.bfgr
224                               # a 6-line file
$ cat hello.metta.lis
    1   ; a MeTTa source file     # and nothing else
```

versus

```
$ ./ccsmetta -lt hello.metta
$ wc -c hello.metta.bfgr
1504
```

Check two things after every run, because nothing else will tell you: that the `.lis` listing contains all your source lines, and that the `.bfgr` is a plausible size for the input. A silently truncated parse looks exactly like a fast one — this is the same failure mode that once inflated our own benchmark numbers, and it was caught by an atom count that did not match.

6. Add source positions, if you want them

```
./ccsmetta -lt -D 1 hello.metta
```

The `-D` flag records the source line and column of each parsed element into the binary. It is off by default and costs nothing when off. Consumers read it through two accessors on the loaded binary; *MeTTa on CCS* §2 documents the data structure and its semantics.

Every CCS-generated frontend accepts `-D` — it is part of the shared shell, not something this grammar provides.

7. Consuming what you produced

A `.bfgr` is only useful if something reads it. The `scbcpp/` package from step 4 is the C++ side of that: link `libscbcpp.a`, include the generated `mettaKeyWordDefinition.h`, and walk the loaded structure.

The companion `metta_bench` use case ships a complete, buildable consumer that does exactly this and reports what it found — start there rather than from a blank file.

8. Automating the loop

`e2e_test.sh`, published beside this guide, performs steps 3 through 5 against b3u.dev end to end: log in, compile the grammar, download both zips, lay out the directory, and verify the delivered members. It is both a worked example of driving the service from a script and this use case's own end-to-end test.

```
export B3U_BASE=https://b3u.dev
export B3U_EMAIL=you@example.com
export B3U_PASSWORD=...
./e2e_test.sh
```

Terms