Source: cppcc docs/MeTTa_on_CCS_Draft_0.1.md @ 934786a — curated 2026-08-08; edits land in the source first.

MeTTa on CCS

Language support, source positions, and conformance

Draft 0.1 — 2026-08-08

How the Compiler Compiler System targets MeTTa — a language outside the ALGOL family that CCS was first built for. Three parts: what the MeTTa lexical extension provides, what the `-D` flag builds for a consumer, and the external corpus used to test the generated frontend.

Companion to the CCS Technical Specification, whose vocabulary this document follows. Sections referenced as [Spec §N] are that document's.

---

1. MeTTa as a lexical extension

1.1 Where the extension point is

The Specification defines four predefined token classes available in every SGDL grammar — `identifier`, `integerToken`, `stringToken` and `termToken` [Spec §3.1.4] — and states where anything beyond them belongs:

> Lexical extensions to the four predefined token classes … are not specified within the SGDL file. They are provided by `Language` subclasses written in C++ and registered with the tokenizer at parser construction. [Spec §3.3]

MeTTa is a worked instance of that procedure. It is the interesting instance because MeTTa is not an ALGOL-family language: it is s-expression based, its symbols admit characters an `identifier` may not contain, and it carries two sigil-prefixed name spaces that have no counterpart in the predefined set.

Everything in this section describes **what the extension provides**. How a `Language` subclass is written is the Specification's Chapter 10 and is not repeated here.

1.2 The three added token classes

`LanguageMeTTa` supplies three token classes beyond the predefined four:

token class	recognises
mettaSymbol	Lisp-permissive symbols — names containing characters `identifier` excludes (hyphens and other punctuation used in ordinary MeTTa naming)
mettaVariable	`$`-prefixed variables
mettaSpaceRef	`&`-prefixed space references

Comment convention is also part of the extension: MeTTa uses `;` to end-of-line, where the ALGOL-family default does not.

Once registered, these behave exactly like the predefined classes from the grammar author's point of view — they are named on a rule's right-hand side and nothing else changes. That is the property worth noticing: the extension is lexical only. No parsing construct, no rule syntax, and no runtime concept is specific to MeTTa.

1.3 The grammar

The consequence of the previous paragraph is that the MeTTa grammar is small. In full, it is a `program` of s-expressions, an `sexpr` that is either an atom or a parenthesised list, and an `atom` that alternates over the token classes and the standalone terminals MeTTa uses:

```
(sexpr ::= atom | list )
(list  ::= '(' { sexpr } ')' )
(atom  ::= identifier | mettaSymbol | mettaVariable | mettaSpaceRef
         | stringToken | integerToken | … )
```

The iteration group `{ sexpr }` is SGDL's Kleene closure [Spec §3.4]; the alternation is an alternative group. Both are ordinary constructs used ordinarily. The complete grammar is 49 lines, and it is published with the `metta` use case.

1.4 What this demonstrates

A language whose surface syntax is entirely unlike the one CCS was first built for turned out to need **no parser change, no runtime change, and no binary-format change** — only three token classes and a grammar. That is the claim the MeTTa work supports, and it is a claim about where CCS's generality actually lives: in the separation of lexical recognition from syntax processing [Spec §1.2].

---

2. What `-D` builds

2.1 The flag

`cppcc -D 1 <file.sgr>` records **source positions** into the compiled grammar binary. `-D 0`, the default, omits them.

Two properties matter more than the flag itself:

- **Every generated frontend inherits it.** The flag is handled by the shared shell that all CCS-generated compilers link, so a frontend generated for any grammar accepts `-D` without its author doing anything.
- **It is off by default and costs nothing when off.** No position section is written, no per-token capture happens, and the binary is byte-identical to one compiled without the feature existing.

2.2 The data structure

When positions are present, a consumer sees this:

```
typedef std::pair<unsigned int, unsigned int>  LineColumn;
typedef unsigned int                           Length;
typedef std::map<LineColumn, Length>           Positions;

const Positions& positions(TagLong kwnnum, TagLong edpnum) const;
bool             hasPositionSection() const;
```

A `Positions` map answers, for one element of the parsed structure, *every source location where it was matched* and how long the match was at each.

2.3 The semantics worth knowing

Line and column are 0-based, matching the Language Server Protocol convention. A consumer feeding an editor or a diagnostic renderer can use them directly.

Positions attach to data elements, not to names. The design records locations against the parsed occurrences rather than against symbol definitions, because a name appears once while its occurrences are many.

One element can hold many positions, which is why the structure is a map rather than a single triple. A grammar element reached through an iteration group, an alternative group, or several distinct references is matched at each of those places, and all of them are recorded. This is the property that makes the feature useful for anything beyond trivial grammars.

Decoding is lazy and idempotent. Positions cost nothing until first asked for; asking repeatedly costs nothing more.

Legacy binaries are safe, and distinguishable. A binary compiled before the feature existed, or compiled with `-D 0`, loads normally: `hasPositionSection()` returns false and `positions()` returns an empty map. A consumer can therefore tell "this binary carries no positions at all" from "this binary carries positions but this element has none" — a distinction that matters when the absence of a position is itself information.

2.4 What it is for

Anything that needs to point back at source text: editor tooling and language servers, diagnostics that underline the offending span, refactoring tools, coverage and provenance tracking, and debuggers for languages whose compilers CCS generated.

The internal representation of the position section is not part of this contract and is deliberately not documented here. Consumers use the two accessors above; the encoding is free to change.

---

3. The external conformance corpus

3.1 Why the corpus is external

A generated frontend can be tested against examples its authors wrote, which proves it does what they expected. The stronger test is a corpus someone else wrote for a different implementation.

MeTTa has one: the Hyperon project's own `.metta` sources and its reference parser. Testing against those asks a harder question — not "does our parser do what we intended" but "does it agree with the implementation the language community actually uses".

3.2 What is tested

Two harnesses run over that material.

Decompile round-trip. Each `.metta` source is compiled to a grammar binary, then decompiled, and the result compared against the expected textual form. This is the round-trip oracle applied to real corpus files rather than hand-written cases.

Parser equivalence. The same source is parsed twice — once through Hyperon's reference `SExprParser`, once through the CCS-generated frontend — and the resulting atom streams compared.

A small set of hand-written golden files covers the token classes of §1.2 individually, so a failure can be localised to a lexical class before the corpus tests are consulted.

3.3 ⚠ What the harnesses do and do not assert

This is stated plainly because it would be easy to overclaim.

Both harnesses run in **discovery mode** by an explicit decision. They assert that the run completed and that files were actually evaluated, and they **report** per-file divergences — but individual divergences are **not** failures. A catalogue of the known divergent files is maintained alongside the tests, with a characterisation of each.

So what the current state supports is:

> a conformance harness exists, it runs against an external corpus and a reference implementation, and its divergences are enumerated and characterised rather than unknown.

What it does **not** yet support is a claim that the generated frontend agrees with the reference parser. Promoting the harnesses to strict mode — where any divergence fails the run — is deliberate future work, gated on the divergence list being either closed or explicitly accepted.

A reader evaluating CCS should weigh that accordingly. An enumerated divergence list is a considerably better position than an untested frontend, and a considerably weaker one than proven

equivalence.

References

- *CCS Technical Specification*, Draft 1.0 — the vocabulary and section references used throughout.
- *Compiler Compiler System with Syntax-Controlled Runtime and Binary Application Programming Interfaces*, US Patent 8,464,232.
- The `metta` and `metta_bench` use cases, published alongside this document, which carry the grammar and the runnable code.