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

Building the MeTTa SCB consumer

This guide builds `metta_counts` — a C++ program that reads the SCB binaries your MeTTa frontend produces — and runs it against your own data.

It assumes you have already worked through the `metta` use case, so that `~/ccsUseCases/metta` holds the delivered frontend, the `cpp` bundle, and the `scbcpp` package. Nothing else is required: no compiler toolchain from us, no cppcc source tree, no runtime library.

1. What you are building against

Two things from the `metta` delivery:

- **`scbcpp/`** — the standalone SCB C++ library: `include/` plus `lib/libscbcpp.a`. This is the whole read side of CCS, and it links alone.
- **`cpp/mettaKeyWordDefinition.h`** — generated from your grammar. It names the keyword constants the walker switches on, which is how a consumer talks about *your* language rather than about CCS.

2. Lay out and build

```
mkdir -p ~/ccsUseCases/metta_bench
cd ~/ccsUseCases/metta_bench
mv ~/Downloads/metta_bench-code.zip .
unzip -q metta_bench-code.zip
make
```

The makefile defaults to `~/ccsUseCases/metta` for both paths. If your layout differs:

```
make METTA_HOME=/path/to/your/metta
```

3. Produce something to read

```
cd ~/ccsUseCases/metta
cat > sample.metta <<'SAMPLE'
; sample
(= (greet $who) (say "hello" $who))
(= (add $a $b) (+ $a $b))
(greet &self)
(add 1 2)
(: greet (-> Symbol Expression))
SAMPLE
./ccsmetta -lt sample.metta
```

⚠ The `-lt` matters and its absence is silent — see the `metta` build guide §5.1. If `sample.metta.bfgr` comes out around 224 bytes, the frontend parsed almost nothing and exited 0 anyway.

4. Run it

```
cd ~/ccsUseCases/metta_bench
./metta_counts ~/ccsUseCases/metta/sample.metta.bfgr 200
```

```
loaded: sample.metta.bfgr (188 longs), iterations: 200
  Helper      0.50 us/walk   atoms=23  lists=10  sym=1 var=6 ref=1 id=8
str=1 int=2 term=4
  Helper2     0.36 us/walk   atoms=23  lists=10  sym=1 var=6 ref=1 id=8
str=1 int=2 term=4
OK: both walkers agree on 23 atoms
```

5. Reading the output

Check the counts against your source by hand the first time. For the sample above: `$who` twice and `$a`/`$b` twice each is `var=6`; `&self` is `ref=1`; `"hello"` is `str=1`; `1` and `2` are `int=2`; `+` is the one Lisp-permissive symbol; `=`, `=`, `:` and `->` are the four standalone terminals; and there are ten opening parentheses.

Doing that once is worth more than trusting the program, and it is how the bug described in §6 was found.

The timing is per walk, with the load excluded — the binary is read once before the loop starts. That split is the point of the format, and it is what the benchmark document's crossover analysis rests on.

6. ⚠ The mistake this code made, so you do not repeat it

Every standalone terminal — `(`, `)`, `=`, `:`, `->` — arrives with the *same* keyword id, `KW_TERMTOKEN`. Which terminal it actually is comes from the second value, `nnn`.

The first version of this walker tested the wrong one. It compiled, it ran, the two walkers agreed with each other, and it reported 43 atoms for a sample that has 23 — because all twenty parentheses were being counted as atoms. Nothing failed; the number was just wrong.

It was caught by adding up the atoms by hand from the source. That is the only check that would have caught it, which is why §5 suggests doing it.

7. Going further

`metta_counts.cc` is meant to be edited. The walk is about forty lines; the interesting extensions are shallow:

- tally by rule rather than by token class, to profile *shape* instead of content;
- collect source positions alongside the counts (compile with `ccsmetta -lt -D 1` and read them through the two accessors *MeTTa on CCS* §2 documents);
- replace the counting with your own consumer — populate a store, build an index, emit another format.

Terms