Repetition Cardinality Constraints

- Cardinality Assertions

Cardinality Assertions

CongoCC supports **cardinality assertions** that allow grammar authors to express detailed constraints on how often particular expansions may appear within a loop. These assertions can be applied to individual expansions, parenthesized groups, or even nested loops — providing precise control over repetition, subset selection, and optionality.

Syntax

Cardinality assertions are most clearly used as **prefixes** to expansions within a loop. They take the following forms:

Syntax	Meaning
`&`	Zero or one occurrence (optional)
`&&`	Exactly one occurrence (required)
`&n&`	Exactly `n` occurrences
`&n: &`	At least `n` occurrences
`&n: m&`	Between `n` and `m` inclusive occurrences

All values `n` and `m` must be non-negative integers with `n ≤ m`.

Application: Loops and Set-Based Choices

Cardinality assertions apply only to expansions **within loops** — either `(...)+` or `(...)*`. In these contexts, they modify the **loop semantics**, not the expansion's inner meaning.

Set-Based Choice

A typical usage pattern involves **annotating each choice within a loop**, effectively defining a constrained power set:

```
( &&A | &B | &1:3&C )+
```

This loop:

- Requires `A` exactly once
- Allows `B` optionally
- Allows `C` between 1 and 3 times
- Accepts any order of the above, as long as constraints are satisfied

---

Scope and Placement Rules

1. **Assertions May Appear Anywhere**

Cardinality assertions are **not restricted to the top-level loop elements**. You may place them on:

- Plain expansions
- Parenthesized sequences
- Terminals and nonterminals
- Even nested loops

In fact, since they are assertions applied both in lookahead and in parsing, they may be used anywhere a normal `ENSURE` or `ASSERT` may be used.

This allows expressing rich structure with scoped repetition requirements.

```
( &1:2&( Foo Bar ) | &Baz )+
```

In this example, the group `Foo Bar` must appear 1–2 times, as a sequence, and `Baz` may appear at most once in the scope of the *OneOrMore* expansion.

2. **Assertions Apply to the Nearest Enclosing Loop**

A cardinality assertion always modifies the **nearest loop** that encloses the annotated expansion.

This allows two powerful patterns:

a. **Constraining the Outer Loop Itself**

You can use a parenthesized group with an assertion to control how many times the **entire loop body** executes:

```
( &2: 4&( A | B | C ) )+
```

This forces each iteration of the outer loop to consume **2 to 4 elements** drawn from the inner choice.

b. **Constraining an Inner or Nested Loop**

You can place an assertion on a nested loop or sub-sequence to constrain its behavior:

```
( &( Foo )+ | Bar )+
```

This says:

- A sequence of `Foo`s (at least one) may appear at most once as a group
- `Bar` is unrestricted

You can also nest deeper:

```
( &2&( ( Foo | Bar )+ ) )*
```

Here, each `(Foo | Bar)+` inner loop must yield **exactly 2 elements**, and the outer loop can repeat zero or more times.

Behavior Notes

- Assertions outside of any loop are warned and **ignored**.
- Violating a cardinality constraint causes the **loop to fail during lookahead**. If a violated constraint entails lookahead that succeeded it will subsequently **fail as an assertion during parsing**.
- Multiple annotations in the same scope are **independent** — each is counted and checked separately.
- Repeated choices are only valid if explicitly allowed by cardinality and the controlling loop is only valid if no associated cardinality is exceeded.

Examples

Simple Optional Elements

```
( &X | &Y | &Z )*
```

Each of `X`, `Y`, and `Z` may appear zero or one time.

Required Combinations

```
( &&Modifier | &1:2&Annotation )+
```

The loop must include `Modifier`, and may include 1 or 2 `Annotation`s.

Loop-wide Constraint via Parenthesis

```
( &3:5&( &Option1 | Option2 )+ )*
```

The outer loop must consume either **no options**, or between **3 and 5** **`Option1`****s and any number of** **`Option2`****s**. In other words, `( &Option1 | Option2 )+` must appear 3–5 times or not at all, but when it does appear, `Option1` may appear at most once while `Option2` may appear any number of times.

What If Cardinality Assertions Weren’t Available (Baseline Rewrites)

Below are two common strategies you’d need without cardinality assertions. They both work, but they’re **verbose**, **hard to maintain**, and they move correctness checks to parse-time rather than lookahead-time.

1) Permutation Enumeration (Combinatorial Explosion)

Goal (with cardinality):

```
( &&A |  &B |  &1:3&C )+
```

- `A` exactly once, `B` optional (0–1), `C` between 1 and 3 — in any order.

Without cardinality, you might enumerate sequences to cover all valid orders and counts. Even for three elements this gets big fast:

```
Rule :
    A Tail
  | B A Tail
  | A B Tail
  | C A TailC1
  | A C TailC1
  | B C A TailC1
  | C B A TailC1
  | C A B TailC1
  | A C B TailC1
  ;

Tail :
    /* zero B, C already in order */
  | B         /* add optional B once */
  ;

TailC1 :      /* we've seen 1 C so far */
    C TailC2  /* 2 Cs total */
  | B         /* optional B, stop at 1 C */
  |           /* stop at 1 C */
  ;

TailC2 :      /* we've seen 2 Cs so far */
    C TailEnd /* up to 3 Cs total */
  | B
  |
```

```
  ;

TailEnd :     /* exactly 3 Cs */
    B
  |
  ;
```

This only covers **some** orderings; completing it cleanly requires many more rules or a different strategy. The maintenance and readability costs escalate as options grow.

2) Lookahead Predicates with `ENSURE` (Flags/Counters)

A more compact and **unbiased** approach is to place **lookahead predicates** *before* each choice. This mirrors cardinality behavior: constraints are enforced **prior to consumption**.

```
Rule() : {
  boolean seenA = false;
  boolean seenB = false;
  int cCount = 0;
}
(
    ENSURE { !seenA } A { seenA = true; }   // A: exactly once
  | ENSURE { !seenB } B { seenB = true; }   // B: at most once
  | ENSURE { cCount < 3 } C { cCount++; }   // C: up to 3 times
)+
ASSERT {
  seenA && cCount >= 1 : "Require A once and at least one C"
}
```

This avoids permutation blow-ups and keeps checks in lookahead. The final `ASSERT` ensures minimum totals (e.g., “A once” and “≥1 C”).

Summary

Cardinality assertions in CongoCC provide:

- Fine-grained control over repetition within loops

- A mechanism for expressing unordered, subset-based combinations
- Declarative grammar constructs that eliminate ambiguity

This system gives you a **powerful toolset** to express grammars that are traditionally hard to encode in LL(k) parsers — without sacrificing predictability, performance, or clarity.