Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Disambiguating Context-Free Grammars by Selecting the Minimum-Cost Parse Tree

Mina Hematikarimkhani

mina2002hemmati@gmail.com

Islamic azad University https://orcid.org/0009-0003-6135-6457

Research Article

Keywords: ambiguity, context-free grammars, disambiguation, parsing, dynamic programming, operator precedence, dangling else

Posted Date: November 5th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11613500/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Abstract

A grammar that is ambiguous gives some strings more than one parse tree, but a compiler or interpreter has to work with a single one. The usual remedies each depend on something specific: precedence declarations work inside LR generators, grammar rewriting has to be done by hand, and tree filters act only after parsing. We take a different route and treat disambiguation as an optimisation problem. Each parse tree is given a cost, made up of the number of violated disambiguation patterns, the number of non-terminal nodes, and a canonical tie-breaking key, compared in that order. The cheapest tree is the one we keep. Since violations are penalised and never forbidden outright, every string in the language still receives a tree, and the language itself is untouched. When penalties depend only on a node and its immediate surroundings, the optimum can be found with a CYK-style dynamic program over input spans. For patterns that look further down the tree, such as the dangling else with an optional else-branch, each subtree also reports a small profile of whatever its parent may ask about it. The method works for any context-free grammar, including grammars with ε-rules and cycles, and returns exactly one tree per string of the language. Our implementation runs in $O(n^4 \cdot C)$ time, where C depends only on the grammar and the patterns. We do not suggest replacing linear-time LR parsing in production compilers. The method is meant for general-purpose tools that must cope with arbitrary grammars and still give a reproducible answer. It has clear limits. The chosen tree is only as meaningful as the patterns supplied, and deciding whether a grammar is ambiguous remains undecidable.

1. Introduction

Grammars used to describe programming languages are often ambiguous. The standard textbook example is

E → E + E | E * E | (E) | INT | Id

The string Id + INT * Id can be read as Id + (INT * Id) or as (Id + INT) * Id. Two other familiar cases are the dangling else, as in if c if c a else a, and associativity, as in a - b - c. In every case whatever consumes the tree, whether an interpreter, a translator or an editor, needs one reading, and usually a particular one.

Existing techniques deal with the problem at different points in the pipeline. LR generators such as Yacc and Bison settle shift/reduce conflicts while building the parse table, using precedence declarations [1, 2]. That works only if the grammar becomes conflict-free once those declarations are applied. Rewriting the grammar into levels (Expr, Term, Factor) must be done manually, changes the shape of the trees, and has no obvious counterpart for ambiguities that are not about operators. Disambiguation filters [3] discard unwanted trees after or during general parsing, but a filter that only deletes can delete everything and so reject a valid string, or can leave several trees standing. Parsing expression grammars sidestep ambiguity through ordered choice, which changes the language being defined instead of choosing among its structures.

We start from a different question. Rather than deciding which trees to delete, we ask **how bad** each tree is and keep the best one. This small change has useful consequences. A tree survives for every string in the language, the result is unique by construction, and when badness is local the best tree can be computed by dynamic programming whatever the class of the grammar.

The main contributions are the following. First, we define a cost for parse trees as a lexicographic triple, which guarantees that a minimum exists, that cycles are harmless, and that the answer is unique (Section 3). Second, we give a CYK-style dynamic program that finds the minimum-cost tree for any context-free grammar, including ε-rules and cycles (Section 4). Third, we introduce profiles, which extend the method from patterns about direct children to patterns about siblings, grandchildren and the right spine; this is what the dangling else with an optional else needs (Section 4.2). Finally, we report on an implementation and worked examples with the trees it produces, and we say plainly what the method does and does not guarantee (Sections 5 to 7).

2. Related Work

Precedence and associativity in LR parsing. Aho and Johnson [1] and Aho, Johnson and Ullman [2] showed that ambiguous grammars can drive deterministic LR parsers if conflicts are resolved with precedence and associativity information, including the preference for shift in the dangling-else case. The approach is efficient, but it is limited to grammars whose conflicts those rules are enough to remove.

Disambiguation filters. Klint and Visser [3] studied filters over parse forests. A filter removes trees. It is complete if every sentence of the language keeps at least one tree, and local if it can be decided from a node and its children. Our penalty patterns are local in a similar way, and our selection is complete in the same sense, because exactly one tree is returned for each sentence of the language. We have not matched the formal definitions any more closely than this informal correspondence.

Safe precedence specification. Afroozeh et al. [4] look at how operator precedence can be specified safely in a declarative grammar formalism. Our pattern language is not tied to operators, and an incomplete or unsatisfiable specification does not cause failure. Violations are simply counted.

Weighted parsing. Choosing the least costly derivation is an instance of Viterbi-style parsing for weighted grammars, and Goodman’s semiring framework [5] treats such computations in a uniform way. What we add is a lexicographic cost whose second and third components do structural work (removing cycles and fixing a deterministic choice), together with profiles that keep non-local patterns tractable for dynamic programming.

Ambiguity analysis. Whether a context-free grammar is ambiguous is undecidable, and the literature offers conservative approximations [6]. Our method makes no attempt to decide ambiguity. It works on one input string at a time and returns one tree whenever the string belongs to the language.

3. Preliminaries and Cost Model

Let G = (N, Σ, R, S) be a context-free grammar and w = w_0 … w_{n-1} an input string. For a non-terminal A and a span [i, j), T(A, i, j) is the set of parse trees rooted at A that derive w[i..j). Rules with an empty right-hand side (ε-rules) are allowed. For a tree t, |t| denotes its number of non-terminal nodes.

3.1 Forbidden patterns

In their simplest form, patterns are triples (p, ℓ, c), read as “in a node that uses rule p, the child at position ℓ must not use rule c”. Precedence and associativity compile into such triples. For a rule E → E ⊕ E with precedence level L(⊕), a child built from a weaker operator is forbidden at either recursive position. A child of the same level is forbidden on the side that the associativity does not allow: the right child for left-associative operators, the left child for right-associative ones, and either child for non-associative operators. Users can also write patterns directly, as in the dangling-else example below.

3.2 Cost

Let v(t) be the number of pattern matches in t, counting one for each node and each pattern it satisfies. The cost of t is the triple c(t) = (v(t), |t|, κ(t)), compared lexicographically. The key κ(t) is a canonical sequence of rule indices and split points that identifies the tree uniquely. The tree we select is

t* = argmin { c(t) : t ∈ T(S, 0, n) }.

Each component has its own job. The violation count carries the user’s intent. It is deliberately soft, so a pattern can penalise a tree but never exclude it, and a tree always exists whenever the string is in the language. The size component deals with cycles. In a grammar that contains A → A, any tree could be extended by an arbitrarily long chain of unit derivations, but every added node raises |t|, so the minimum never goes around a cycle. The key settles whatever ties remain, which makes the output deterministic.

Existence and uniqueness follow from a short argument. For any bound, only finitely many trees have size within it, and v and |t| are natural numbers, so the order on the first two components is a well-order. The key makes the order total on trees. A unique minimum therefore exists whenever w is in the language, and T(S, 0, n) is empty, so that the method reports failure, exactly when it is not.

4. Algorithm

4.1 Simple patterns

Suppose every pattern has the form (p, ℓ, c). Then the only thing a parent needs to know about a child is the rule at the child’s root, and we can keep a table B[A, i, j, r] holding the cheapest tree rooted at A over [i, j) whose root rule is r. We fill it span by span, shortest first. For each rule r: A → X_1 … X_k and each way of splitting the span among the symbols, a candidate’s cost is the sum of the children’s costs, plus one for

every child whose rule matches a pattern with r at that position, plus one for the node itself. For binary rules this is exactly the CYK recurrence.

4.2 Profiles for non-local patterns

Simple patterns cannot express every disambiguation rule. Take the dangling else with an optional else-branch:

S → if c S O | a

O → else S | ε

For if c if c a else a, the two trees differ in which if owns the else, which shows up in the grandchild position O (Fig. 4). The rule we want is that in an if with an else-branch, the then-branch must not end, along its right edge, in an if that has no else. Stating this requires conditions on sibling rules and on the right spine of a subtree. In the implementation the rule is written as a single line, with the two alternatives of O labelled els and none and the first alternative of S labelled x:

S -> if c S O #x | a

O -> else S #els | ε #none

forbid: x[3 = els] 2 x[~ = none]

Read from left to right, the pattern says: in a node of rule x whose child 3 uses els (this if has an else), child 2 (the then-branch) must not be a node of rule x that has none on its right spine (the symbol ~ stands for the right spine).

To support this we generalise patterns to fragments. A fragment names a parent rule, conditions on sibling positions, a child position and rule, and conditions on that child’s grandchildren or right spine. To keep the dynamic program valid, each subtree publishes a profile σ(t) that contains exactly what some pattern may ask about it: its root rule, the rules of the grandchildren that patterns refer to, and the subset of watched rules found on its right spine. The table becomes B[A, i, j, σ], and the best tree is stored separately for each profile.

The general design principle is simple. Whatever the patterns ask about a subtree has to be part of its profile, and a profile has to be computable from the profiles of the children. When both hold, the cost a parent adds depends only on profiles, and keeping the minimum per profile is enough.

4.3 ε-rules and cycles

With ε-rules or unit rules, a child can cover the same interval as its parent, so the entries for one span depend on one another. We process spans in order of increasing length, starting from length zero, and inside each span we repeat the rule updates until nothing improves. This is the same fixed-point idea as

in Bellman–Ford. Termination comes from the size component of the cost, since going around a cycle always makes a tree larger and therefore worse.

4.4 Procedure

DISAMBIGUATE(G, patterns, w):

B[A,i,j,σ] ← ∞

for len = 0 … n: // empty spans included (ε)

for i = 0 … n − len: j ← i + len

repeat // fixed point within the span

changed ← false

for each rule r : A → X_1 … X□:

for (σ, c) in COMPOSE(r, i, j): // sequence DP over X_1 … X□,

// best cost per partial profile

if c < B[A,i,j,σ]: B[A,i,j,σ] ← c; changed ← true

until not changed

return min over σ of B[S,0,n,σ] // FAIL iff w ∉ L(G)

5. Correctness and Complexity

We only outline the argument here. A complete proof is available as a separate document.

The first ingredient is optimal substructure. For a fixed child profile, the parent's cost is strictly increasing in each child's cost, because everything the parent adds depends only on profiles. Replacing a child by a cheaper subtree with the same profile therefore cannot make the parent worse, and the optimum for a parent can be assembled from the per-profile optima of its children. The second ingredient is termination. Every update in the inner loop strictly lowers some table entry in a well-ordered set, so the loop must stop, and at the fixed point the table equals the true minimum for each (A, i, j, σ).

As for running time, for rules with at most two non-terminals the base algorithm has the usual CYK shape. Our implementation handles arbitrary right-hand sides, ε-rules and cycles in one uniform way and runs in $O(n^4 \cdot C)$, where C depends only on the grammar and the pattern set. In our tests the number of fixed-point iterations per span stayed below $2 + |N| \cdot q$, where q is the number of profile states.

For operator grammars there is a clean special case. If all operators on the same level share the same associativity, a tree without violations exists and is unique. Its root is the lowest-precedence operator outside parentheses: the last one for left-associative levels, the first for right-associative ones. This matches the familiar precedence-climbing rule. We have checked the correspondence by testing but have not proved it formally.

6. Worked Examples

6.1 Precedence and associativity

Consider a grammar with the rules r_0: E → E + E (level 1, left-associative), r_1: E → E * E (level 2, left-associative), r_2: E → INT, r_3: E → Id and r_4: E → (E), and take the input Id + INT * Id. The precedence declarations turn into forbidden patterns such as (r_0, 2, r_0), (r_1, 0, r_0), (r_1, 2, r_0) and (r_1, 2, r_1). The table produced by the program, with empty entries left out, is shown in Table 1.

Table 1
Dynamic-programming table for the input Id + INT * Id

Span	Text	Root rule	Violations	Size
[0,1)	Id	E → Id	0	1
[2,3)	INT	E → INT	0	1
[4,5)	Id	E → Id	0	1
[0,3)	Id + INT	E → E + E	0	3
[2,5)	INT * Id	E → E * E	0	3
[0,5)	whole input	E → E + E	0	5
[0,5)	whole input	E → E * E	1	5

Over the full span there are two candidates. The tree rooted at + has INT * Id as its right child and violates nothing. The tree rooted at * has Id + INT as its left child, which matches the pattern (r_1, 0, r_0) and so costs one violation. Both trees have size 5, so the violation count decides the matter, and the result is Id + (INT * Id). Figure 1 shows the same decision for the input a + 3 * b, as drawn by the implementation: the selected tree is framed in green and the other candidate is drawn beside it.

Associativity works the same way. With a left-associative -, the input a - b - c has two trees and the pattern that forbids a same-level child on the right selects ((a - b) - c) (Fig. 2).

Several levels and a right-associative operator can be mixed freely. For the grammar with ^ (level 3, right-associative), * and / (level 2), and + and - (level 1) the input 2 ^ 3 ^ x * y - z + 4 yields ((((2 ^ (3 ^ x)) * y) - z) + 4) (Fig. 3), in agreement with the usual conventions.

6.2 The dangling else

With the ε-based dangling else and the pattern forbid: x[3 = els] 2 x[~ = none], the input if c if c a else a has two trees, differing only in which if owns the else. The selected tree is (if c (if c a (else a)) ε): the else attaches to the nearest if, with zero violations (Fig. 4). The alternative would let the outer if own the else, so its then-branch ends in an else-less if and matches the pattern.

The pattern also scales to deeper nesting. For if c if c if c a else a else a the program returns the tree in Fig. 5, in which each else is bound to the closest unmatched if.

6.3 Missing patterns, ε-rules and cycles

Without any pattern the choice is left to the canonical key. For S → S S | a on a a a there are two trees; the program returns (a (a a)) (Fig. 6). The result is deterministic, but it is an arbitrary choice, which is exactly why the quality of the answer depends on the patterns supplied.

Termination does not need any special treatment. For the cyclic grammar A → A | a, the input a yields the single leaf and the cycle is never entered. For A → A A | ε | a on a a, infinitely many trees exist, since ε can be inserted anywhere, yet the smallest tree is unique and is the one returned (Fig. 7). The number of trees reported by the notebook for this grammar is small only because its enumeration skips repeated derivations.

Table 2 collects the cases that we ran in the accompanying notebook. In every case the selected tree has zero violations.

Table 2
Summary of the cases run with the implementation

Grammar / input	Selected tree	Trees found	Remark
expressions: a + 3 * b	(a + (3 * b))	2	precedence
expressions: a * 3 + b	((a * 3) + b)	2	precedence
expressions: (a + 3) * b	((a + 3) * b)	1	parentheses
expressions: a - b - c	((a - b) - c)	2	left associativity
dangling else: if c if c a else a	(if c (if c a else a))	2	forbid: long 2 short
A → A | a: a	a	1	cycle
S → S S | a: a a a	(a (a a))	2	no patterns, key decides
S → a S b | ε: a a b b	(a (a ε b) b)	1	ε-rule
S → a S b | ε: (empty string)	ε	1	empty input
S → A B, A and B optional: a	(a ε)	2	forbid: s 0 A_eps
dangling else with ε: if c if c a else a	(if c (if c a (else a)) ε)	2	right-spine pattern
A → A A | ε | a: a a	(a a)	1 (no repeats)	ε and cycle

7. Discussion

Table 3 sets the method beside the two classical approaches.

Table 3
Comparison with classical approaches

Grammar class	LR/Yacc + precedence	Tree filters	This method
	must be LALR(1) after resolution	**any CFG (general parsing)**	**any CFG, including ε and cycles**
Mechanism	conflict resolution	tree deletion	local penalty plus minimum
Guarantee	one parse, or a conflict error	zero or several trees may remain	exactly one tree per sentence of the language
Language altered?	usually not (but %nonassoc can reject strings)	yes, if all trees are removed	no
Time	linear	depends on general parsing	$O(n^4 \cdot C)$ in our implementation
Who defines “correct”	programmer (precedences)	programmer (filters)	programmer (patterns)

The method is not a substitute for LR parsing in production compilers, since its running time is far from linear. Its natural setting is tooling that has to accept arbitrary, possibly user-supplied grammars and still return one reproducible tree: language workbenches, editors and static-analysis front ends.

Several limitations should be stated openly. The first is that “correct” is whatever the user’s patterns say. The algorithm returns the cheapest tree with respect to the patterns it was given. If they are incomplete or contradictory, for example a non-associative operator chained twice or two different associativities on one level, every tree may incur violations, and the choice falls back to the canonical key, which is deterministic but has no principled justification (Fig. 6). The second is that patterns only look downward. They can see direct children, siblings, grandchildren and the right spine, but not the left spine or arbitrary depth, so if a distinction lies outside these patterns the grammar has to be written differently. The third is speed, because $O(n^4 \cdot C)$ is slow for large inputs. The fourth is that the method says nothing about whether a grammar is ambiguous, which is undecidable. Last, regarding novelty, choosing a minimum-cost derivation is standard in weighted parsing. We are not aware of earlier work that combines the lexicographic triple with profiles for non-local patterns in this way, but our search of the literature was not exhaustive, and we claim no priority.

8. Conclusion

We have described a uniform way to disambiguate context-free grammars: give each parse tree a lexicographic cost, and pick the cheapest tree by dynamic programming over spans, using profiles to cover patterns that reach beyond direct children. The approach does not depend on the grammar class, copes with ε-rules and cycles, leaves the defined language unchanged, and returns exactly one tree per sentence. Several directions remain open: speeding up the implementation, widening the pattern

language to include the left spine and deeper context, relating the method more formally to filter-based disambiguation, and evaluating it on grammars of real programming languages.

Declarations

Funding: [to be completed].

Competing interests: [to be completed].

Data and code availability: [to be completed; the reference implementation is a single Python module, accompanied by a notebook that produces Figures 1 to 7].

References

1. Aho, A.V., Johnson, S.C.: LR parsing. ACM Computing Surveys 6(2), 99–124 (1974)
2. Aho, A.V., Johnson, S.C., Ullman, J.D.: Deterministic parsing of ambiguous grammars. Communications of the ACM 18(8), 441–452 (1975)
3. Klint, P., Visser, E.: Using filters for the disambiguation of context-free grammars. Technical Report P9426, University of Amsterdam (1994)
4. Afroozeh, A., van den Brand, M., Johnstone, A., Scott, E., Vinju, J.: Safe specification of operator precedence rules. In: Software Language Engineering (SLE 2013)
5. Goodman, J.: Semiring parsing. Computational Linguistics 25(4), 573–606 (1999)
6. Brabrand, C., Giegerich, R., Møller, A.: Analyzing ambiguity of context-free grammars. [venue and year to be confirmed; originally 2007]

Figures

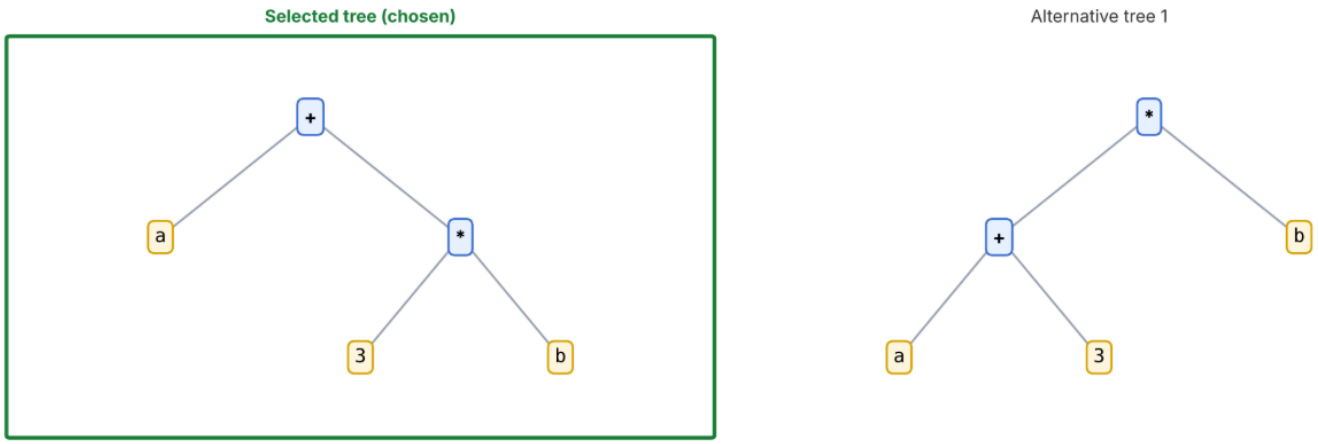


Figure 1

Both parse trees of a + 3 * b. The tree a + (3 * b) is selected with zero violations; the other tree would apply + below *.

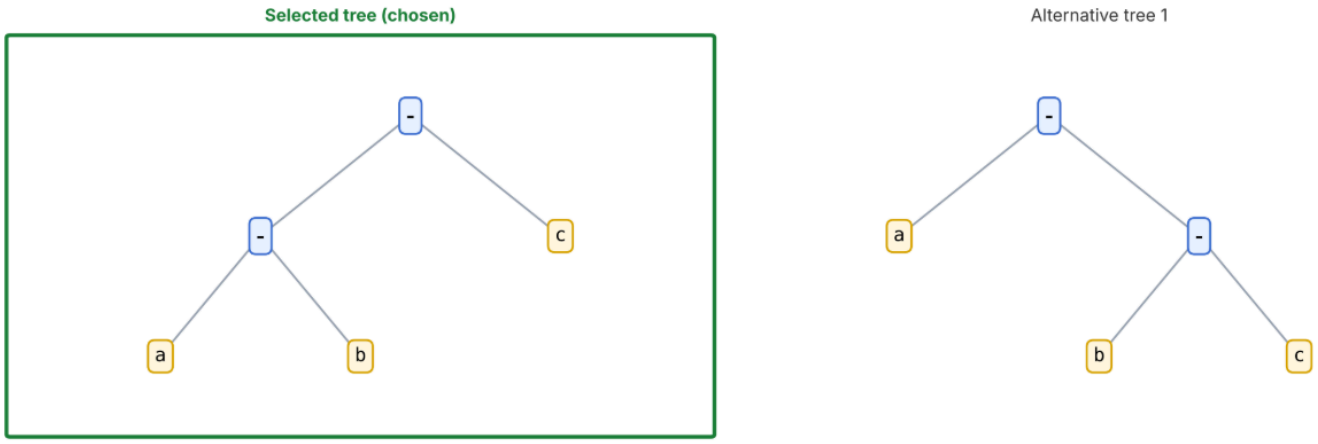


Figure 2

Left associativity. The selected tree groups a - b first; the alternative groups b - c first and violates the associativity pattern.

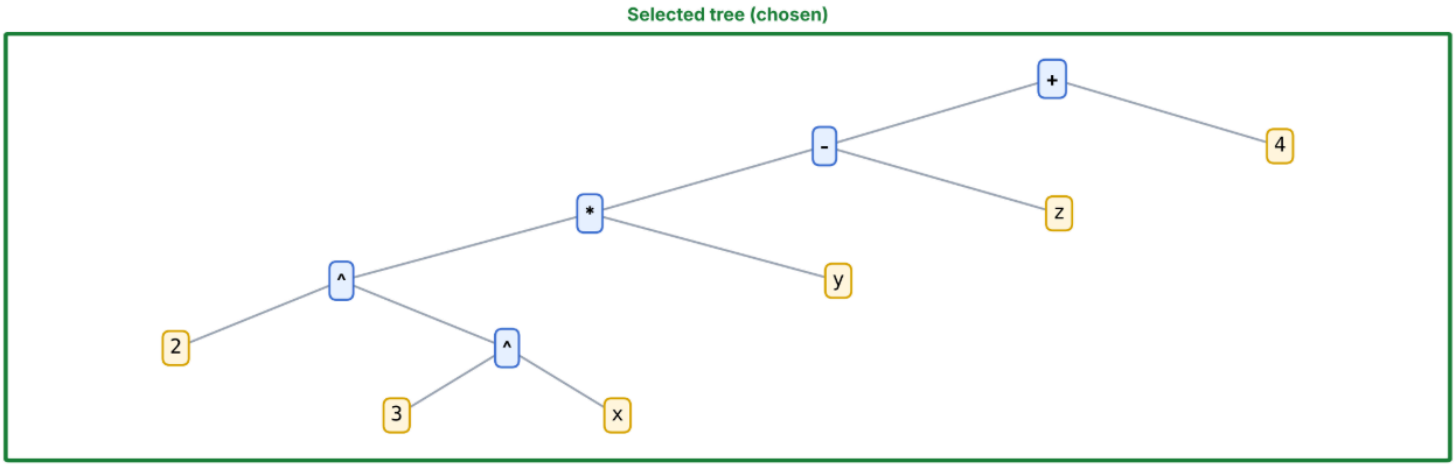
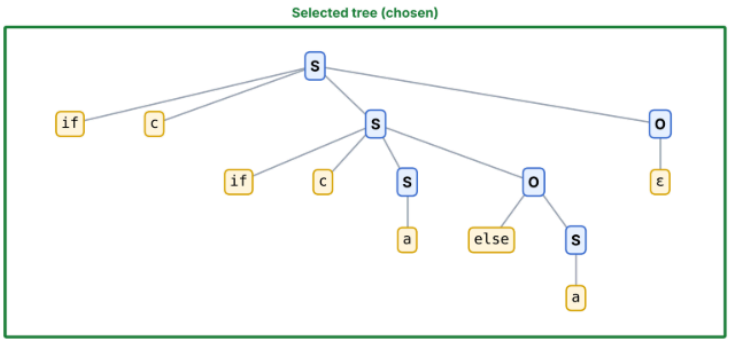


Figure 3

Mixed precedence and associativity. The compact operator tree of 2 ^ 3 ^ x * y - z + 4: ^ is right-associative and binds tightest, the other levels are left-associative.



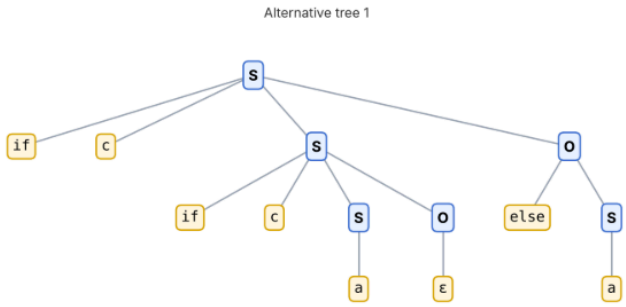


Figure 4

Dangling else with an optional else (O → else S | ε). Left (green): the else belongs to the inner if. Right: the else belongs to the outer if, which the right-spine pattern penalises.

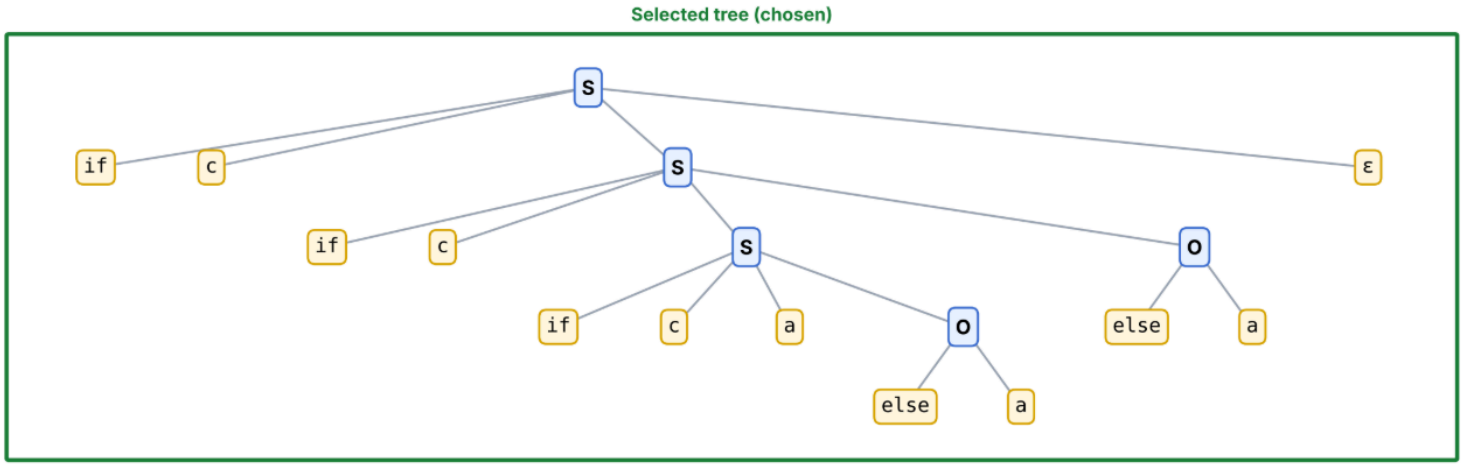


Figure 5

Three nested if statements with two else-branches. The innermost two ifs each own one else; the outermost if has an empty (ε) else.

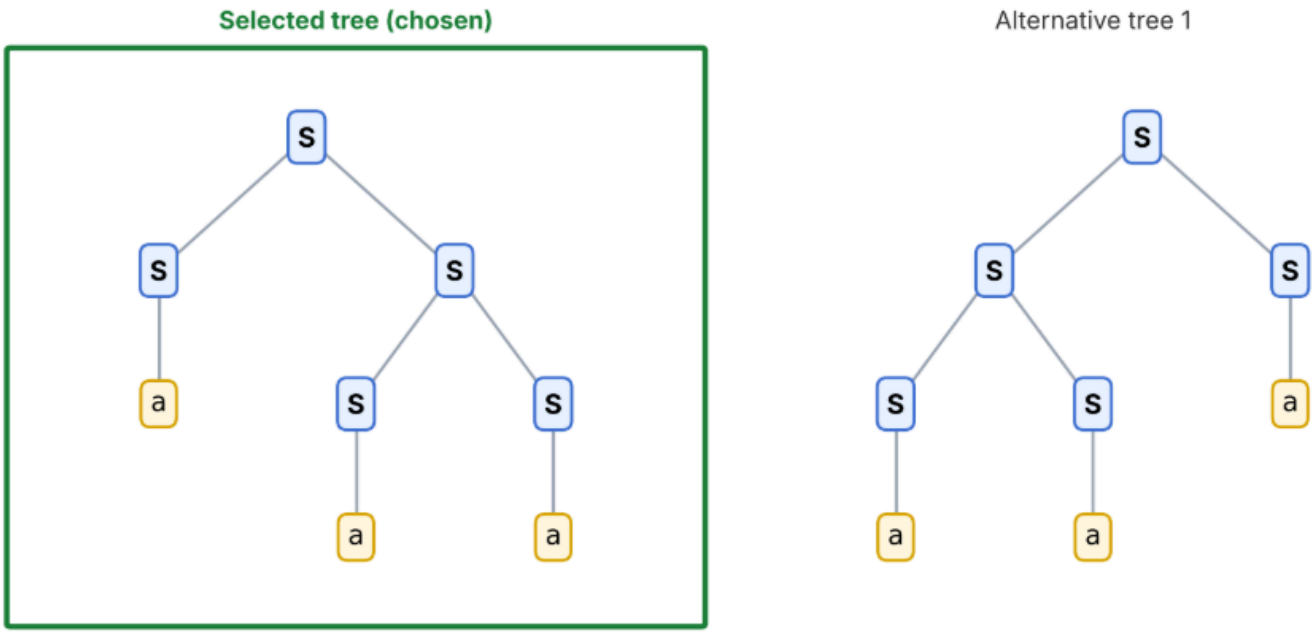


Figure 6

A grammar with no disambiguation patterns. Both trees have the same cost in the first two components, so the canonical key decides.

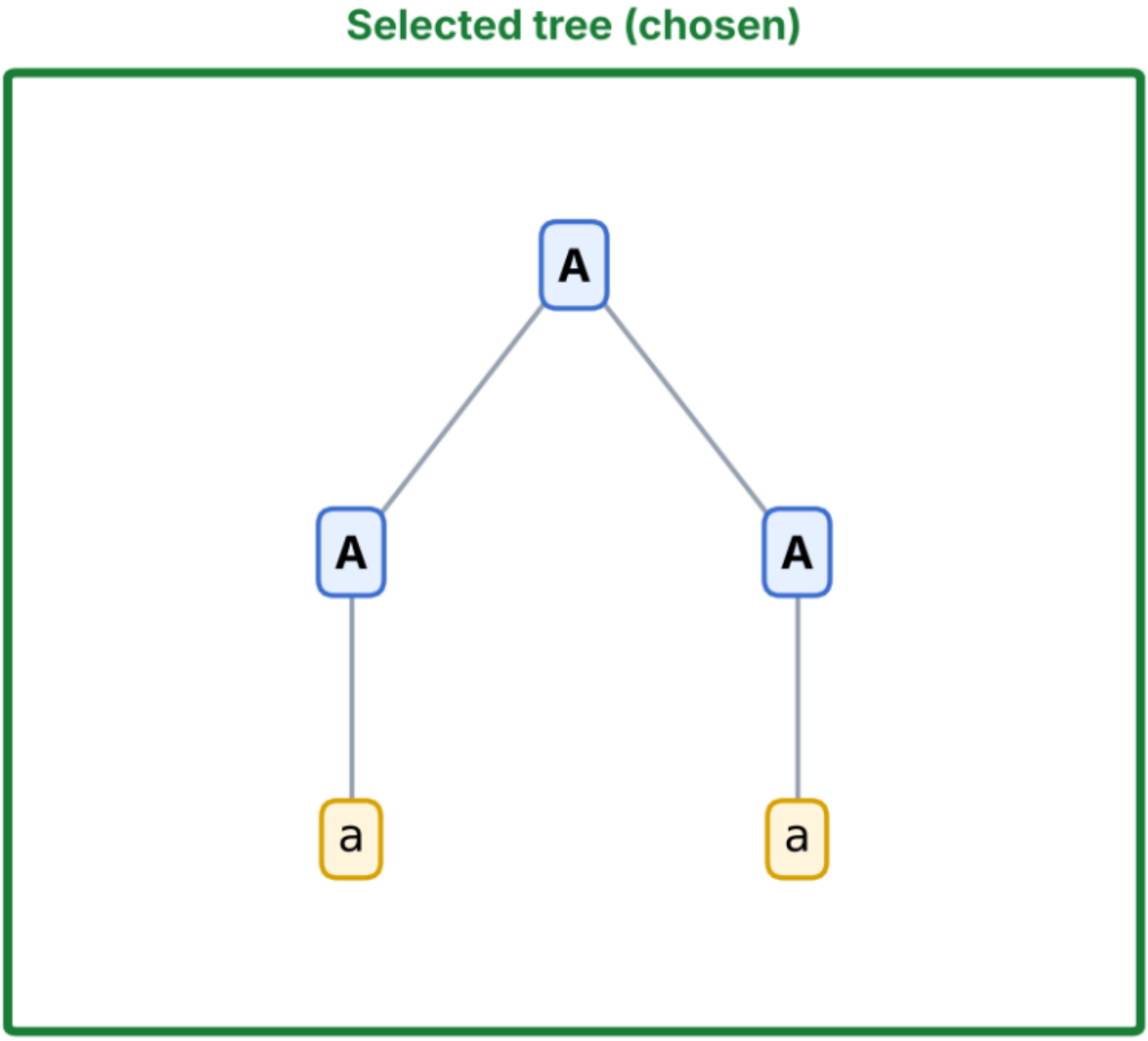


Figure 7

ε-rule combined with a cycle. The minimum-size tree (a a) is selected.