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.

Automated Semantic State-Layout Synthesis for Generated Lexers:Structural Evaluation with Rollback-Aware Semantics

Reda Belaiche

reda.belaiche@u-pec.fr

Paris-Est Créteil University

Research Article

Keywords: lexical analysis, automated software engineering, code generation, DFA representation, state assignment, semantic dispatch

Posted Date: March 31st, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9238282/v1

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

Additional Declarations: No competing interests reported.

Automated Semantic State-Layout Synthesis for Generated Lexers: Structural Evaluation with Rollback-Aware Semantics

Reda Belaiche

LISSI, Université Paris-Est Créteil, Créteil, France

Corresponding author: reda.belaiche@u-pec.fr

Abstract

Generated lexical analyzers are shaped not only by the DFA transition graph they implement, but also by ordinary code-generation choices such as how states are numbered and how that numbering is interpreted at run time. In practice, a state identifier often carries more meaning than simple acceptance: it may encode token family, priority, auxiliary status, and, under maximal-munch scanning, whether the state matters for rollback or last-accept recovery. Many generators recover that meaning from incidental numeric identifiers through lookup tables or ad hoc conditionals. This paper asks whether that semantic layer can itself be synthesized. More precisely, we study how to construct a semantics-preserving numeric layout for DFA states together with a comparison-based classifier over state identifiers, while leaving the recognized language, transition graph, and longest-match behavior unchanged.

After formalizing the problem, we prove basic structural properties and present constructive procedures for contiguous class layouts, profile-guided interval ordering, and hierarchical comparison-tree layouts. We then evaluate these layouts structurally on ten lexer-oriented benchmarks ranging from 13 to 1368 states, under semantic labelings that range from binary acceptance to fine token-family distinctions with rollback-aware variants. Across 12,600 completed generated configurations, a consistent picture emerges. Flat multi-class interval layouts are always realizable, but their worst-case comparison depth grows with the number of semantic classes. Profile-aware ordering leaves interval complexity unchanged while reducing expected depth substantially when the profile matches deployment behavior, especially for finer labelings. Hierarchical comparison trees sharply reduce worst-case depth, particularly on the larger synthetic token-trie benchmarks, though they also generate larger classifier code. Rollback-aware semantic splitting improves rollback-oriented predicates at a modest semantic cost in the coarse and operational settings.

Keywords: lexical analysis, automated software engineering, code generation, DFA representation, state assignment, semantic dispatch

1 Introduction

Deterministic finite automata (DFAs) often appear in papers as abstract machines, but in practice they end up as code: lexers, recognizers, protocol handlers, and other generated components [4, 12]. In lexical-analysis tools and finite-state compilers, much of the engineering work lies precisely in turning an automaton-level description into executable code [4, 6, 7, 8]. Once the automaton has been fixed, a generator still has a surprising amount of implementation freedom. One of the quieter choices is the numbering of states and, related to it, the way the current state is

interpreted semantically at run time. In lexer code, that interpretation is touched constantly, so small representational decisions can have visible consequences in the generated dispatch logic.

In real scanners, the semantic meaning of a state is usually richer than a simple accepting versus non-accepting split. Generators routinely distinguish ordinary non-accepting states, dead states, token families, priority-sensitive accepts, and, in maximal-munch pipelines, accepting states whose meaning matters for rollback or last-accept recovery [1, 2, 4, 6, 7]. None of that changes the language recognized by the DFA, but it does shape the control structure of the emitted code.

In many implementations, numeric state identifiers come from construction order or some other incidental enumeration, and semantic information is recovered afterwards through tables, membership tests, or scattered conditionals. That works, but it leaves an obvious question on the table: if the generator is free to choose the numbering, can it choose one that makes the semantic structure easier to compile? Put differently, can state identifiers be arranged so that semantic classes become contiguous or hierarchically organized, and can the resulting classifier be emitted as cleaner comparison code? Comparable implementation questions arise elsewhere in automata engineering, including scanner construction and compact DFA representations [9, 10, 11]; here the focus is the semantic dispatch layer attached to generated DFA states. That is the problem studied here. It is a representation question about generated software, not a change to the automaton itself.

The central question can be stated plainly:

> Given a fixed DFA and a semantic labeling of its states, can a generator automatically synthesize a numeric state layout together with a comparison-based semantic classifier so that semantic dispatch becomes structurally cleaner while preserving the automaton and lexer semantics exactly?

We refer to this task as *semantic state-layout synthesis*. The goal is to derive layouts that a code generator can use directly, possibly informed by visitation profiles or by a semantic hierarchy already present in the scanner design. The resulting classifiers may take the form of threshold chains, interval tests, or comparison trees over numeric state identifiers. The idea is reminiscent of implementation-level state assignment in hardware synthesis [3, 13, 14], but the object of study here is generated software and, more specifically, the semantic dispatch code attached to DFA states.

In concrete terms, this paper makes three contributions.

1. **Problem formulation and basic structural results.** We cast semantic state-layout synthesis as an explicit construction problem for semantically labeled DFA states, and we prove the basic correctness and realizability facts that justify treating state numbering as a design space rather than as an incidental by-product.

2. **Constructive synthesis procedures.** We present explicit procedures for contiguous class layouts, profile-aware interval ordering, and hierarchical comparison-tree layouts, so the representation space can be used directly in a code-generation pipeline.

3. **Structural evaluation on lexer-oriented generators.** We evaluate six layout families on ten lexer-oriented benchmarks under binary, coarse, operational, and fine semantic labelings, including rollback-aware variants. The results outline a stable design space: frequency-aware interval ordering helps expected depth, hierarchical comparison trees help worst-case depth, rollback-aware splitting improves rollback-oriented predicates at a small semantic cost, and the DAG-sharing variant provides no benefit on the supplied suite.

The scope of the paper is intentionally narrow. We do not change the recognized language, the transition graph, the state count, the longest-match discipline, or rollback behavior. What we study

instead is the shape of the generated semantic dispatch. For that reason, the empirical analysis centers on structural properties of the emitted classifier—comparison depth, interval structure, predicate geometry, and source size—while target-specific runtime behavior is left for follow-up work.

2 Preliminaries and Problem Statement

2.1 DFA model and semantic labeling

We assume a deterministic finite automaton

$$A = (Q, \Sigma, \delta, q_0, F),$$

where Q is a finite set of states, Σ is a finite input alphabet, $\delta : Q \times \Sigma \to Q$ is a total transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states.

For $w = a_1 \cdots a_n \in \Sigma^*$, the extended transition function is defined as usual by

$$\delta^*(q, \varepsilon) = q, \qquad \delta^*(q, wa) = \delta(\delta^*(q, w), a).$$

The recognized language is

$$L(A) = \{\, w \in \Sigma^* \mid \delta^*(q_0, w) \in F \,\}.$$

For plain language recognition, acceptance is a binary property. In scanner implementations, however, that binary view is usually too coarse to be useful. We therefore attach a semantic labeling map

$$\lambda : Q \to \mathcal{C},$$

where $\mathcal{C}$ is a finite set of semantic classes. Typical examples include:

- non-accepting states,
- accepting states for identifiers,
- accepting states for integers,
- accepting states for operators,
- dead or error states,
- rollback-relevant accepting states,
- priority-sensitive accepting states,
- other generator-specific internal categories.

The map λ can be chosen at different levels of granularity. It may merge several states into a single class, or it may separate states that are all accepting but differ in token kind, priority, auxiliary category, or rollback behavior.

Definition 2.1 (Semantic signature)**.** *A* semantic signature *is any tuple-valued map*

$$\sigma : Q \to S_1 \times \cdots \times S_r$$

collecting implementation-relevant state attributes. In a lexer, a useful example is

$$\sigma(q) = \big(\mathsf{acc}(q), \mathsf{tok}(q), \mathsf{prio}(q), \mathsf{rb}(q), \mathsf{aux}(q)\big),$$

where acc *is an acceptance flag,* tok *is a token kind or* $\bot$*,* prio *is a priority descriptor,* rb *indicates rollback relevance, and* aux *stores optional internal category information.*

In that sense, a semantic class labeling λ may be read either as a direct annotation or as a coarsening of a richer signature σ.

2.2 Renumberings and classifiers

Let $n = |Q|$. A *numeric state layout* is a bijection

$$\pi : Q \to \{0, \dots, n-1\}.$$

Given such a layout, transitions can be transported to numeric identifiers by defining

$$\delta_\pi(i, a) = \pi\Big(\delta(\pi^{-1}(i), a)\Big).$$

Likewise, acceptance becomes

$$F_\pi = \{\pi(q) \mid q \in F\}.$$

Definition 2.2 (Comparison-based semantic classifier)**.** *Let $\mathcal{D} = \{0, \dots, n-1\}$. A* comparison-based semantic classifier *is a function*

$$g : \mathcal{D} \to \mathcal{C}$$

implemented using a prescribed family of comparisons on numeric identifiers. Examples include:

- *one-interval-per-class classifiers,*
- *k-interval-per-class classifiers,*
- *linear threshold chains,*
- *balanced comparison trees,*
- *comparison DAGs obtained by sharing common subtests.*

The correctness requirement is

$$g(\pi(q)) = \lambda(q) \qquad \text{for all } q \in Q.$$

Definition 2.3 (Semantic state-layout synthesis problem)**.** *Given (A, λ) and a classifier family $\mathfrak{G}$, synthesize:*

(i) a bijection $\pi : Q \to \{0, \dots, n-1\}$, and

(ii) a classifier $g \in \mathfrak{G}$,

such that $g(\pi(q)) = \lambda(q)$ for all $q \in Q$, while optimizing one or more objectives such as:

- *total number of intervals,*
- *worst-case comparison depth,*
- *expected comparison depth under a visitation distribution,*
- *code-size or branch-structure proxies for generated classifiers.*

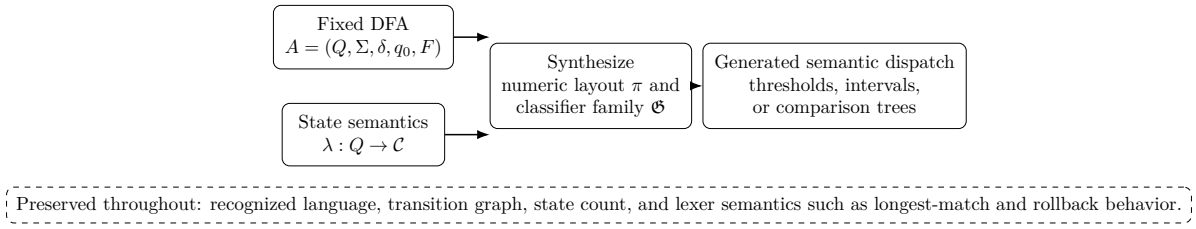


Figure 1: Overview of semantic state-layout synthesis. The DFA and semantic labeling are fixed inputs; only the numeric state assignment and the compiled semantic classifier are synthesized.

2.3 What is and is not being changed

What we synthesize is the representation, not the automaton's behavior.

Remark 2.4. *It does* not *change the recognized language, the transition graph, the number of states, or the longest-match discipline of a scanner. It changes only:*

(i) the numeric assignment of state identifiers, and

(ii) the generated code used to classify the current state semantically.

Transition computation itself may still be table driven, direct coded, compressed, or otherwise implemented.

Figure 1 gives the overall picture: the automaton and its state semantics are fixed, while the numeric layout and compiled semantic classifier are the objects being synthesized.

3 Basic Structural Results

3.1 Semantics-preserving renumbering

Theorem 3.1 (Semantics-preserving renumbering)**.** *Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA, and let $\pi : Q \to \{0, \dots, n-1\}$ be any bijection. Define*

$$A_\pi = (\{0, \dots, n-1\}, \Sigma, \delta_\pi, \pi(q_0), F_\pi),$$

with

$$\delta_\pi(i, a) = \pi\Big(\delta(\pi^{-1}(i), a)\Big), \qquad F_\pi = \{\pi(q) \mid q \in F\}.$$

Then $L(A_\pi) = L(A)$.

Proof. We show by induction on $w \in \Sigma^*$ that

$$\delta_\pi^*(\pi(q), w) = \pi(\delta^*(q, w)) \qquad \text{for all } q \in Q.$$

For $w = \varepsilon$, both sides equal $\pi(q)$. Suppose the property holds for w, and let $a \in \Sigma$. Then

$$\begin{aligned}
\delta_\pi^*(\pi(q), wa) &= \delta_\pi\big(\delta_\pi^*(\pi(q), w), a\big) \\
&= \delta_\pi\big(\pi(\delta^*(q, w)), a\big) \\
&= \pi\Big(\delta(\pi^{-1}(\pi(\delta^*(q, w))), a)\Big) \\
&= \pi(\delta(\delta^*(q, w), a)) \\
&= \pi(\delta^*(q, wa)).
\end{aligned}$$

Now let $w \in \Sigma^*$. Then

$$w \in L(A_\pi) \iff \delta_\pi^*(\pi(q_0), w) \in F_\pi \iff \pi(\delta^*(q_0, w)) \in \{\pi(q) \mid q \in F\}.$$

Since π is injective, this is equivalent to $\delta^*(q_0, w) \in F$, hence to $w \in L(A)$. Therefore $L(A_\pi) = L(A)$. □

The statement is elementary, but it matters: it is the formal reason we are free to treat state numbering as a space for optimization rather than as a fixed implementation accident.

3.2 Contiguous layouts and interval classification

Fix a class set $\mathcal{C} = \{c_1, \ldots, c_m\}$ and define class fibers

$$Q_c = \{q \in Q \mid \lambda(q) = c\}.$$

Proposition 3.2 (Contiguous class-block layout)**.** *For any finite labeled state set (Q, λ), there exists a bijection $\pi : Q \to \{0, \ldots, n-1\}$ such that, for every class $c \in \mathcal{C}$, the image $\pi(Q_c)$ is a contiguous interval of integers.*

Proof. Choose any ordering $c_1, \ldots, c_m$ of the classes. For each class c_i, choose any ordering of the states in Q_{c_i}. Number first all states of Q_{c_1} consecutively, then all states of Q_{c_2} consecutively, and so on until Q_{c_m}. By construction, the image of each Q_{c_i} is a contiguous interval. □

Remark 3.3. *Proposition 3.2 is purely about class grouping. It does not optimize transition locality, cache layout, or global code size. It states only that if semantic grouping is the objective, then a one-interval-per-class layout always exists.*

Proposition 3.4 (Interval classification)**.** *Assume π is such that for every class $c \in \mathcal{C}$, the set $\pi(Q_c)$ is an interval $[L_c, U_c]$. Then the semantic classifier*

$$g(i) = c \quad \iff \quad L_c \leq i \leq U_c$$

is correct. Hence runtime semantic classification reduces to interval comparison.

Proof. Since the class fibers partition Q and π is bijective, the intervals $[L_c, U_c]$ are pairwise disjoint and cover $\{0, \ldots, n-1\}$. Let $q \in Q$ and set $c = \lambda(q)$. Then $\pi(q) \in \pi(Q_c) = [L_c, U_c]$, so $g(\pi(q)) = c = \lambda(q)$. Thus the classifier is correct. □

Proposition 3.5 (Comparison-structure realization)**.** *Assume each class occupies one contiguous interval and let the classes be ordered so that*

$$U_{c_1} < U_{c_2} < \cdots < U_{c_m}.$$

Then semantic dispatch can be implemented as:

(a) a linear threshold chain of worst-case depth $m-1$ if the final class is the fall-through case, or m if every class is guarded explicitly, and

(b) a balanced comparison tree over interval boundaries of worst-case depth $\lceil \log_2 m \rceil$ up to additive constants.

A DAG implementation can be obtained by sharing repeated suffixes or leaves.

Proof. Since the intervals are disjoint and ordered, an identifier i belongs to class c_j exactly when

$$U_{c_{j-1}} < i \leq U_{c_j},$$

with the convention that $U_{c_0} = -1$. A linear chain therefore checks whether $i \leq U_{c_1}$, else whether $i \leq U_{c_2}$, and so on. If the last class is returned by fall-through, only the first $m - 1$ boundaries require explicit tests.

Alternatively, class determination is equivalent to predecessor search in the sorted boundary list $(U_{c_1}, \ldots, U_{c_m})$. A balanced binary search tree on these boundaries has worst-case depth $\lceil \log_2 m \rceil$ up to standard rounding and constant-offset effects. If multiple branches lead to identical actions, subtrees or leaves may be shared, yielding a DAG. □

3.3 Frequency-aware ordering

Let $f(c) \geq 0$ be a class visitation frequency with $\sum_{c \in \mathcal{C}} f(c) = 1$. If a linear chain tests classes in order $c_1, \ldots, c_m$, its expected number of class comparisons is

$$E(c_1, \ldots, c_m) = \sum_{j=1}^{m} j\, f(c_j).$$

Lemma 3.6 (Frequency-aware ordering)**.** *Among all linear threshold chains using one interval per class, the expected comparison cost $E(c_1, \ldots, c_m)$ is minimized by ordering classes in non-increasing visitation frequency.*

Proof. Consider two adjacent positions $j < j + 1$ holding classes x and y. Their contribution to the expected cost is

$$j\, f(x) + (j + 1)\, f(y).$$

If they are swapped, the contribution becomes

$$j\, f(y) + (j + 1)\, f(x).$$

The original order is no worse than the swapped order exactly when

$$j\, f(x) + (j + 1)\, f(y) \leq j\, f(y) + (j + 1)\, f(x),$$

which simplifies to $f(x) \geq f(y)$. Therefore any adjacent inversion in which a less frequent class precedes a more frequent one can be exchanged to reduce or preserve the expected cost. Repeatedly removing such inversions yields a non-increasing frequency order, which is optimal. □

Remark 3.7. *Lemma 3.6 concerns only the order of already contiguous class blocks in a* linear *chain. It does not claim global optimality for general comparison DAGs or for layouts with multiple intervals per class.*

4 Classifier Families and Objective Functions

The existence results above show that one-interval-per-class layouts are always possible. Whether they are desirable depends on the chosen runtime classifier family and objective.

4.1 Classifier families

Definition 4.1 (One-interval-per-class classifier)**.** *A classifier is* one-interval-per-class *if for every $c \in \mathcal{C}$ there exist integers $L_c \le U_c$ such that*

$$g(i) = c \iff i \in [L_c, U_c],$$

and these intervals partition $\{0, \dots, n-1\}$.

Definition 4.2 (k-interval-per-class classifier)**.** *A classifier is k-*interval-per-class *if each class c is represented by a union of at most k disjoint intervals:*

$$g(i) = c \iff i \in \bigcup_{t=1}^{k_c} [L_{c,t}, U_{c,t}], \qquad k_c \le k.$$

Allowing several intervals per class can be useful when several structural objectives must be balanced at once, for example semantic grouping and transition-locality heuristics. In this paper, however, we focus mainly on the one-interval and hierarchical forms implemented in the supplied repository.

4.2 Objective functions

Given a layout π and a classifier implementation g, the following quantities are natural:

Total interval count. If class c uses k_c intervals, define

$$I(\pi, g) = \sum_{c \in \mathcal{C}} k_c.$$

For one-interval-per-class layouts, $I(\pi, g) = |\mathcal{C}|$.

Worst-case comparison depth. If $\mathrm{depth}_g(i)$ is the number of comparisons used to classify identifier i, define

$$D_{\max}(\pi, g) = \max_{i \in \{0,\dots,n-1\}} \mathrm{depth}_g(i).$$

Expected comparison depth. Let $\mu : Q \to [0, 1]$ be a state-visitation distribution with $\sum_{q \in Q} \mu(q) = 1$. Then

$$D_{\exp}(\pi, g; \mu) = \sum_{q \in Q} \mu(q)\, \mathrm{depth}_g(\pi(q)).$$

Generated-classifier source size. We use generated classifier source bytes as a code-size proxy. Unlike end-to-end runtime, this proxy directly reflects the shape of the emitted semantic dispatch and is independent of the target execution speed.

These objectives do not generally align. A balanced tree may reduce worst-case depth while increasing source size. A profile-ordered linear chain may improve expected depth without changing the worst case. A hierarchical layout may produce cleaner semantic structure without globally minimizing interval count.

Algorithm 1 Contiguous-by-class numbering

Require: finite state set Q, semantic class set $\mathcal{C}$, labeling $\lambda : Q \to \mathcal{C}$
Ensure: bijection $\pi : Q \to \{0, \ldots, |Q| - 1\}$ and interval map $(L_c, U_c)_{c \in \mathcal{C}}$

choose an order $c_1, \ldots, c_m$ of the classes
$next \leftarrow 0$
for $j = 1$ to m **do**
    let $B \leftarrow \{q \in Q \mid \lambda(q) = c_j\}$
    choose any order of the states in B
    $L_{c_j} \leftarrow next$
    for all $q \in B$ in the chosen order **do**
        $\pi(q) \leftarrow next$
        $next \leftarrow next + 1$
    end for
    $U_{c_j} \leftarrow next - 1$
end for
return $(\pi, (L_c, U_c)_{c \in \mathcal{C}})$

Algorithm 2 Frequency-aware class ordering

Require: Q, $\mathcal{C}$, λ, class visitation frequencies f
Ensure: bijection π optimized for linear threshold-chain expectation

sort classes as $c_1, \ldots, c_m$ such that

$$f(c_1) \geq f(c_2) \geq \cdots \geq f(c_m)$$

run Algorithm 1 with this class order
emit a linear threshold chain in the same order
return the resulting layout and classifier

5 Constructive Synthesis Algorithms

This section presents three constructive algorithms that match the implemented technical core. The point is to make the synthesis problem explicit and usable in practice, not to claim that these procedures are universally optimal.

5.1 Algorithm 1: contiguous-by-class numbering

What it optimizes. Algorithm 1 gives the cleanest possible guarantee for flat semantic grouping: one contiguous interval per class.

What it does not optimize. That guarantee is deliberately narrow. The algorithm says nothing about expected comparison depth, transition locality, cache behavior, or branch prediction, and it leaves the class order open.

5.2 Algorithm 2: frequency-aware class ordering

Now assume that a class visitation estimate $f : \mathcal{C} \to [0, 1]$ with $\sum_{c \in \mathcal{C}} f(c) = 1$ is available, whether from profiling, static estimation, training traces, or representative corpora.

Algorithm 3 Hierarchical semantic layout synthesis

Require: Q, a partition tree T over Q
Ensure: bijection π with nested numeric blocks

```
function AssignBlock(X, [L, U])
    if X is a leaf of T then
        assign the states of X arbitrarily to the consecutive identifiers in [L, U]
        return
    end if
    let X_1, ..., X_r be the ordered children of X in T
    choose child order according to a policy (fixed, frequency-aware, or code-generation driven)
    let s_t ← |X_t| for each t
    split [L, U] into consecutive subintervals of lengths s_1, ..., s_r
    for t = 1 to r do
        recursively call AssignBlock(X_t, [L_t, U_t])
    end for
end function
call AssignBlock(Q, [0, |Q| − 1])
return π
```

What it optimizes. By Lemma 3.6, this ordering is optimal for expected class-comparison cost when the emitted dispatch takes the form of a linear threshold chain over class blocks.

What it does not optimize. That result should not be overstated. It does not generally minimize worst-case depth, it says nothing about arbitrary comparison DAGs, and it depends on the quality of the visitation estimate.

5.3 Algorithm 3: hierarchical semantic layout synthesis

For scanner code generation, a single flat partition is often not expressive enough. In many cases it is more natural to separate accepting from non-accepting states first, then split the accepting region by token family, and only after that distinguish finer subtypes such as rollback-sensitive or priority-sensitive accepting states.

Definition 5.1 (Hierarchical semantic partition)**.** *A* hierarchical semantic partition *of Q is a rooted tree whose root is Q, and where each internal node is partitioned into disjoint child subsets whose union is the node subset.*

Typical top-down partitions in a lexer are:

all states $\Rightarrow$ accepting vs. non-accepting $\Rightarrow$ identifier / integer / operator / whitespace / ... $\Rightarrow$ priority or rol

What it optimizes. Algorithm 3 builds nested numeric blocks that map naturally to nested `if/else` dispatch or recursively generated comparison trees.

What it does not optimize. Unless the partition tree and child order are optimized as part of the construction, the algorithm does not guarantee minimum worst-case or expected depth. Its main strength is structural: the generated code can follow the semantic hierarchy instead of flattening it away.

6 Lexer-Oriented Semantics and Runtime Integration

6.1 Real lexer behavior

The key scanner-specific point is easy to miss: semantic state classification is not the same thing as language recognition.

In a practical lexer:

- accepting states usually carry a token family, not merely a Boolean acceptance flag;
- multiple accepting states may belong to the same token family but differ in priority or rollback behavior;
- maximal munch means the scanner often continues after reaching an accepting state;
- the implementation typically remembers the *last accepting state* and its token kind while scanning forward;
- on failure, the implementation may need lookahead handling or rollback to the last accept.

A cleaner state layout therefore does *not* replace longest-match logic. It changes only the way the current state is interpreted semantically inside that logic.

6.2 Skeleton scanner loop with arbitrary numbering

Algorithm 4 sketches a standard maximal-munch scanner loop parameterized by an abstract classifier Classify. The control flow is intentionally agnostic to the numbering strategy.

Algorithm 4 Scanner loop with last-accept tracking

Require: input buffer $x[0..N-1]$, transition function δ, initial state q_0

$p \leftarrow 0$
while $p < N$ **do**
    $q \leftarrow q_0$
    $i \leftarrow p$
    $lastAcceptPos \leftarrow -1$
    $lastClass \leftarrow$ NoAccept
    while $i < N$ **do**
        $q \leftarrow \delta(q, x[i])$
        $cls \leftarrow$ Classify(q)
        if cls is accepting **then**
            $lastAcceptPos \leftarrow i$
            $lastClass \leftarrow cls$
        end if
        if $cls =$ Dead **then**
            break
        end if
        $i \leftarrow i + 1$
    end while
    if $lastAcceptPos \geq p$ **then**
        emit token determined by $lastClass$
        $p \leftarrow lastAcceptPos + 1$
    else
        report lexical error at position p
        $p \leftarrow p + 1$
    end if
end while

This is exactly where state-layout synthesis matters: once the numbering has semantic structure, the call Classify can often be compiled into much more regular comparison code.

6.3 Interval-based semantic dispatch

When each semantic class occupies a single interval, classification can be written as interval tests or, more compactly, as an ordered threshold chain.

Algorithm 5 Semantic dispatch by intervals

Require: numeric state identifier s
Require: class intervals $[L_c, U_c]$ for all $c \in \mathcal{C}$

for all $c \in \mathcal{C}$ **do**
    if $L_c \leq s$ and $s \leq U_c$ **then**
        return c
    end if
end for
return Invalid

When intervals are ordered and contiguous, this can be simplified further into threshold com-

parisons on upper bounds.

6.4 Comparison-tree or DAG dispatch

When the semantic hierarchy is nested, the generated comparison structure can mirror that hierarchy directly.

Algorithm 6 Semantic dispatch by comparison tree / DAG

```
1: function ClassifyTree(s, node)
2:     if node is a leaf then
3:         return node.class
4:     end if
5:     if s ≤ node.threshold then
6:         return ClassifyTree(s, node.left)
7:     else
8:         return ClassifyTree(s, node.right)
9:     end if
10: end function
```

A DAG variant simply reuses identical subtrees or common leaves. In generated C, that can appear as nested conditionals or as a dispatch structure that resembles binary search.

6.5 A compact C-style illustration

The following fragment gives a small C-style illustration of direct semantic dispatch over structured numeric identifiers. It is meant to show the kind of code shape a generator can emit.

```
typedef enum {
  SC_NONACC,
  SC_ID,
  SC_INT,
  SC_OP,
  SC_WS,
  SC_DEAD
} sem_class;

static inline sem_class classify_state(unsigned s) {
  if (s <= 0) return SC_NONACC;   /* [0,0] */
  if (s <= 2) return (s == 1) ? SC_ID : SC_INT; /* [1,1], [2,2] */
  if (s <= 5) return SC_OP;       /* [3,5] */
  if (s <= 6) return SC_WS;       /* [6,6] */
  return SC_DEAD;                 /* [7,7] */
}
```

The transition logic itself is unchanged. Only the semantic interpretation of the current state has been reorganized.

7 Worked Example: A Small C-like Lexical DFA

7.1 Token families and state roles

Consider a small DFA fragment from a C-like scanner. It handles:

- identifiers,
- integer literals,
- operators `=`, `==`, and `+`,
- whitespace,
- a dead state.

A simplified state set is

$$Q = \{q_0, q_{id}, q_{int}, q_{=}, q_{==}, q_{+}, q_{ws}, q_{\perp}\}.$$

The intended meanings are:

- q_0: start state, non-accepting;
- q_{id}: currently reading an identifier, accepting IDENTIFIER;
- q_{int}: currently reading an integer, accepting INTEGER;
- $q_{=}$: after reading `=`, accepting ASSIGN, but extendable to `==`;
- $q_{==}$: after reading `==`, accepting EQEQ;
- q_{+}: after reading `+`, accepting PLUS;
- q_{ws}: reading whitespace, accepting WHITESPACE;
- $q_{\perp}$: dead or error state.

Even in this tiny example, the accepting states do not all mean the same thing. Some correspond to different token families, some differ in rollback relevance, and the scanner still has to obey maximal munch. For instance, after reaching $q_{=}$, the lexer may continue if the next character is `=` and eventually emit `==`; otherwise it may have to fall back to the remembered last accept and emit `=`.

7.2 Semantic signatures

Table 1 gives one possible semantic signature. The semantic classes used below are intentionally coarser than the full signatures.

For the coarse class map λ, let

$$\mathcal{C} = \{\text{NONACC}, \text{IDACC}, \text{INTACC}, \text{OPACC}, \text{WSACC}, \text{DEAD}\}.$$

Table 1: Example states, signatures, and coarse semantic classes.

State	Signature sketch	Coarse class	Notes
q_0	$(0, \bot, \bot, 0, \textsc{Start})$	NonAcc	start/core
q_{id}	$(1, \textsc{Id}, p_{id}, 1, \textsc{Word})$	IdAcc	accepting, rollback-relevant
q_{int}	$(1, \textsc{Int}, p_{int}, 1, \textsc{Num})$	IntAcc	accepting, rollback-relevant
$q_=$	$(1, \textsc{Assign}, p_=, 1, \textsc{Op})$	OpAcc	accepting, extendable to `==`
$q_{==}$	$(1, \textsc{EqEq}, p_{==}, 0, \textsc{Op})$	OpAcc	accepting
q_+	$(1, \textsc{Plus}, p_+, 0, \textsc{Op})$	OpAcc	accepting
q_{ws}	$(1, \textsc{Ws}, p_{ws}, 0, \textsc{Ws})$	WsAcc	often skipped
$q_\bot$	$(0, \bot, \bot, 0, \textsc{Dead})$	Dead	dead/error

7.3 Arbitrary numbering

Suppose an incidental construction order yields the following state identifiers:

$$\pi_{\text{arb}}(q_=) = 0, \quad \pi_{\text{arb}}(q_{id}) = 1, \quad \pi_{\text{arb}}(q_+) = 2, \quad \pi_{\text{arb}}(q_\bot) = 3,$$

$$\pi_{\text{arb}}(q_{==}) = 4, \quad \pi_{\text{arb}}(q_0) = 5, \quad \pi_{\text{arb}}(q_{ws}) = 6, \quad \pi_{\text{arb}}(q_{int}) = 7.$$

Under this layout:

- operator states occupy $\{0, 2, 4\}$,
- accepting states occupy $\{0, 1, 2, 4, 6, 7\}$,
- non-accepting core and dead are split as $\{5\}$ and $\{3\}$.

Under such a numbering, the generated semantic tests are likely to involve discontiguous membership checks rather than simple interval tests.

7.4 Semantics-aware synthesized numbering

Now apply contiguous-by-class numbering with the class order

$$\textsc{NonAcc},\ \textsc{IdAcc},\ \textsc{IntAcc},\ \textsc{OpAcc},\ \textsc{WsAcc},\ \textsc{Dead}.$$

One valid synthesized layout is:

$$\pi(q_0) = 0, \quad \pi(q_{id}) = 1, \quad \pi(q_{int}) = 2, \quad \pi(q_=) = 3,$$

$$\pi(q_{==}) = 4, \quad \pi(q_+) = 5, \quad \pi(q_{ws}) = 6, \quad \pi(q_\bot) = 7.$$

The resulting intervals are:

$$\textsc{NonAcc} = [0, 0], \quad \textsc{IdAcc} = [1, 1], \quad \textsc{IntAcc} = [2, 2],$$

$$\textsc{OpAcc} = [3, 5], \quad \textsc{WsAcc} = [6, 6], \quad \textsc{Dead} = [7, 7].$$

The class test for operators changes from a set-membership condition

$$s \in \{0, 2, 4\}$$

to a single interval test

$$3 \le s \le 5.$$

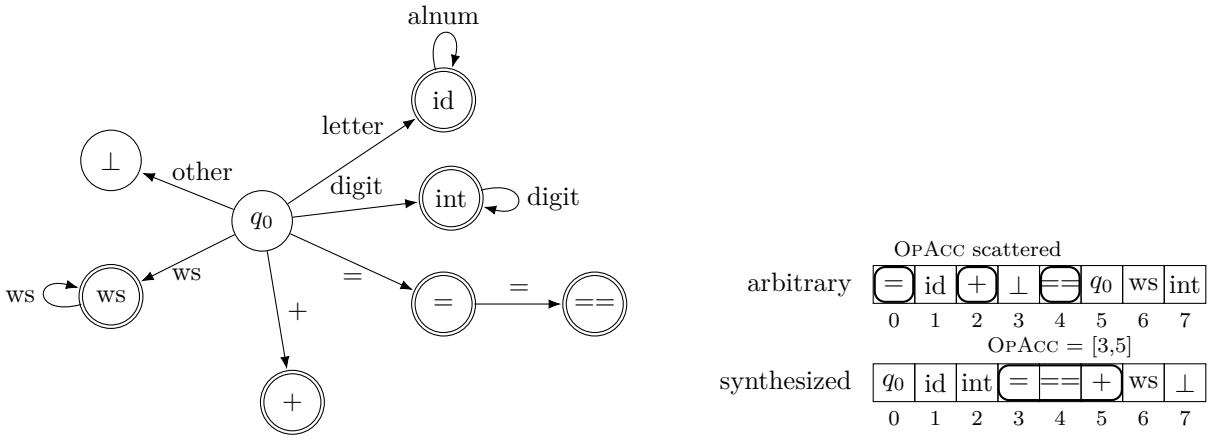


Figure 2: Toy lexer fragment from the worked example. Left: a small DFA with accepting families for identifiers, integers, operators, and whitespace. Right: an incidental numbering scatters the operator-accepting states, whereas the synthesized numbering makes them a single interval.

7.5 Hierarchical variant

A hierarchical synthesis may first carve out all accepting states as one block,

$$[1, 6] \text{ accepting}, \qquad [0, 0] \text{ non-accepting}, \qquad [7, 7] \text{ dead},$$

then subdivide the accepting block as

$$[1, 1] \text{ identifier}, \quad [2, 2] \text{ integer}, \quad [3, 5] \text{ operator}, \quad [6, 6] \text{ whitespace}.$$

That structure supports the kind of nested questions a lexer naturally asks: is the state accepting; if so, which token family does it belong to; and, if necessary, does a finer rollback-sensitive or priority-sensitive subtype matter?

Figure 2 puts the toy scanner fragment and the numbering contrast side by side.

7.6 What the example does and does not show

The example illustrates a genuine multi-class dispatch problem:

- it is not only final versus non-final,
- accepting states have different token families,
- rollback-relevant states exist,
- maximal munch still requires last-accept tracking.

At the same time, the example does *not* alter recognition power or transition behavior. The DFA transitions are the same throughout; only the numeric layout and the dispatch logic are synthesized differently.

Table 2: Benchmark suite used in the structural study. "RB-acc" counts accepting states marked rollback relevant in the supplied metadata. "Split fam." counts token families in which both rollback-relevant and non-rollback-relevant accepting states occur.

Benchmark	Family	States	Accepting	RB-acc	Split fam.
BM1	Small C-like lexer	17	15	3	1
BM2	Richer C-like lexer	56	38	17	1
BM3	JSON-like lexer	30	14	6	0
BM4	HTML-like lexer	13	9	1	0
BM5	SQL-like lexer	66	30	15	1
BM6	Python-like lexer	56	28	12	1
BM7	INI-like lexer	14	10	3	0
BM_SYN_S	Synthetic trie (high)	263	64	31	3
BM_SYN_M	Synthetic trie (medium)	621	171	81	0
BM_SYN_L	Synthetic trie (low)	1368	370	193	0

8 Experimental Setup

8.1 Repository scope and usable artefacts

The supplied repository implements six layout families: B1 (arbitrary numbering with generic lookup dispatch), B2 (binary accepting/non-accepting threshold), B3 (contiguous multi-class interval encoding), B4 (frequency-aware interval encoding), B5-TREE (hierarchical comparison-tree encoding), and B5-DAG (a shared-substructure DAG variant of B5). The generated artefacts include structural summaries, rollback ablations, profile-shift summaries, code-size summaries, benchmark metadata, plots, and runtime-oriented outputs.

For the purposes of this paper, we rely on the part of that evidence that is structurally stable: the completed structural summaries and the associated rollback and profile-shift aggregates. We do not use the throughput tables and hardware-counter outputs as core evidence because, in the provided environment, they were not uniform enough to support the structural claims made here.

Across the supplied structural summaries, all 12,600 generated configurations were marked as completed and none was flagged as a compilation failure. That point is worth stating explicitly, because the paper is about synthesizable code-generation structures rather than about abstract layouts in isolation.

8.2 Benchmarks

The benchmark suite contains ten lexer-oriented DFAs: seven hand-designed, language-like tokenizers and three larger synthetic token-trie families. Table 2 summarizes the benchmark metadata extracted from the generated summaries.

Taken together, these benchmarks cover both small hand-crafted scanners and much larger synthetic accepting-prefix structures. The synthetic cases are especially useful because they put more pressure on semantic partitioning and rollback-sensitive accepting-state structure than the smaller language-like examples do.

8.3 Semantic granularities

The repository evaluates several semantic labelings over the same DFA states:

- **binary**: accepting versus non-accepting/dead;
- **coarse**: accepting token family versus non-accepting auxiliary categories, optionally split by rollback relevance;
- **operational**: a coarser accepting partition designed around operational acceptance classes, again optionally split by rollback relevance;
- **fine**: accepting states separated by token family, auxiliary category, priority, and related attributes.

The main tables below focus on the coarse, operational, and fine labelings, because B2 is meaningful only for Boolean acceptance and therefore does not represent the full multi-class setting. We report the rollback ablation only where the rollback split actually changes the semantic partition. In the supplied fine labeling, the explicit rollback toggle does not refine the class partition any further, so the ablation effect there is null.

8.4 Metrics and aggregation

We emphasize structural metrics that describe the generated semantic classifier directly:

- number of semantic classes,
- worst-case comparison depth,
- expected comparison depth under supplied state-visitation profiles,
- predicate-oriented expected depth for Accept(s) and RollbackAcc(s),
- generated classifier source bytes.

The structural tables report averages over the two matched visitation profiles (`same_A` and `same_B`) on a representative compiler slice. This avoids repeating effectively identical rows across redundant compiler settings. Semantic class counts, interval structure, comparison depth, and generated source bytes are all properties of the emitted classifier itself, so they do not depend on end-to-end runtime measurements.

9 Results

9.1 Main structural trade-offs on hand-designed benchmarks

Table 3 reports the main multi-class structural results for the seven hand-designed lexer-like benchmarks. Only B3, B4, and B5-TREE appear here because B2 is a binary baseline and B1 is a lookup baseline rather than a genuine multi-class comparison classifier.

Three patterns stand out.

First, the flat contiguous multi-class layouts (B3) behave exactly as the theory suggests. They achieve one interval per class, but their worst-case depth grows almost linearly with the number of semantic classes. That growth is fairly mild under the operational labeling and much more visible under the fine labeling.

Table 3: Average structural results on the hand-designed benchmarks (BM1–BM7), aggregated over the two matched visitation profiles.

Granularity	Method	Classes	Worst depth	Expected depth	Source bytes
Coarse	B3	11.57	10.57	5.22	1664
Coarse	B4	11.57	10.57	2.48	1662
Coarse	B5-TREE	11.57	5.29	3.77	2468
Operational	B3	6.36	5.36	2.01	1381
Operational	B4	6.36	5.36	1.65	1381
Operational	B5-TREE	6.36	4.36	2.35	1721
Fine	B3	25.00	24.00	13.67	3021
Fine	B4	25.00	24.00	2.73	3023
Fine	B5-TREE	25.00	8.00	4.01	5641

Second, profile-aware ordering (B4) leaves both class count and worst-case depth unchanged, but it substantially improves expected depth. The gain is small when the labeling is already compact, as in the operational setting, and much larger in the coarse and especially fine settings. On the hand-designed benchmarks, for example, the fine-grained expected depth drops from 13.67 under B3 to 2.73 under B4.

Third, hierarchical comparison trees (B5-TREE) are most compelling when worst-case depth matters. Under the hand-designed fine labeling, B5-TREE reduces the worst-case depth from 24.00 to 8.00. The price is larger emitted source: on average 5641 bytes instead of roughly 3020 bytes for the flat interval encodings. Figure 3 makes that depth trade-off visible for both benchmark families.

So the story here is not that one method wins outright. Under matched profiles, B4 is the strongest flat method for expected depth, while B5-TREE is the strongest method for worst-case depth. Which one is preferable depends on which structural objective matters more.

9.2 Scaling behavior on synthetic token-trie benchmarks

Table 4 reports the same metrics for the three larger synthetic benchmarks.

The synthetic suite makes the same trade-off even harder to miss.

Under coarse semantics, B4 nearly halves expected depth relative to B3 (8.88 versus 4.24), while B5-TREE cuts worst-case depth from 12.83 to 5.33 and reaches a very similar expected depth (4.20). Under fine semantics, the contrast becomes sharper still: B3 reaches expected depth 18.23 and worst-case depth 23.33, whereas B4 lowers expected depth to 4.73 and B5-TREE lowers worst-case depth to 7.00.

These larger benchmarks help explain why the paper's main claims are structural rather than merely existential. It is easy enough to prove that contiguous class layouts exist. The harder and more interesting question is what happens once the semantic partition becomes richer. Flat interval chains remain workable, but their worst-case depth grows quickly. Profile-aware ordering repairs the expectation, not the worst case. Hierarchical trees keep worst-case depth under much better control, though they do so with visibly larger generated classifier code.

Figure 3 gives a compact visual summary of the fine-grained depth trade-off already reported

Table 4: Average structural results on the synthetic benchmarks (BM_SYN_S/M/L), aggregated over the two matched visitation profiles.

Granularity	Method	Classes	Worst depth	Expected depth	Source bytes
Coarse	B3	13.83	12.83	8.88	6508
Coarse	B4	13.83	12.83	4.24	6513
Coarse	B5-TREE	13.83	5.33	4.20	7528
Operational	B3	8.50	7.50	4.22	6211
Operational	B4	8.50	7.50	3.65	6212
Operational	B5-TREE	8.50	5.00	3.76	6744
Fine	B3	24.33	23.33	18.23	7610
Fine	B4	24.33	23.33	4.73	7627
Fine	B5-TREE	24.33	7.00	4.48	9887

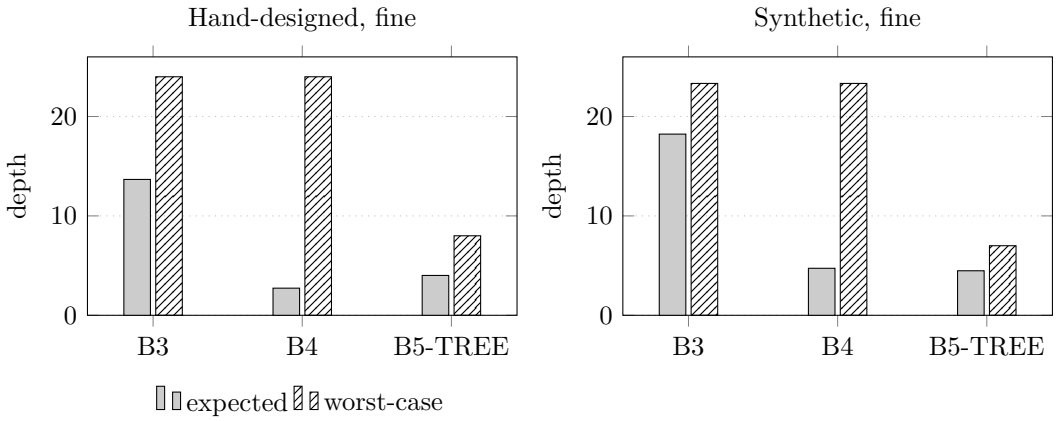


Figure 3: Fine-grained depth trade-off derived directly from Tables 3 and 4. The chart is a visual complement to those tables, not a replacement for the full numerical results.

numerically in Tables 3 and 4.

9.3 Predicate-oriented evidence and rollback-aware semantics

The repository also records comparison-oriented predicates derived from the generated numbering, in particular Accept(s) and RollbackAcc(s). These measurements answer a narrower question than the multi-class tables above: if one wants comparison-based predicates over state identifiers, how fragmented do the relevant semantic regions become under each layout?

Table 5 reports expected predicate depth for representative families. B1 and B2 appear here because they are meaningful predicate baselines even though they are not full multi-class competitors. B1 represents arbitrary numbering, which tends to fragment comparison-based predicates, while B2 is the dedicated binary-threshold baseline for acceptance.

Two conclusions are worth highlighting.

First, arbitrary numbering is a poor substrate for comparison-based acceptance predicates. On the synthetic benchmarks, the expected depth of a comparison-style Accept(s) probe is 157.25

Table 5: Predicate-oriented expected depth for Accept(s) and RollbackAcc(s), averaged over matched profiles. For Accept(s), B2 is the dedicated binary threshold baseline; for RollbackAcc(s), B3/B4/B5-TREE illustrate how rollback-sensitive accepting regions can be made structurally cheaper.

Family	Granularity	B1 Accept	B2 Accept	B3 RBAcc	B4 RBAcc	B5-TREE RBAcc
Hand	Coarse	6.60	1.00	2.61	1.79	1.88
Hand	Operational	6.60	1.00	2.44	2.15	1.76
Hand	Fine	6.60	1.00	2.78	1.82	2.03
Synthetic	Coarse	157.25	1.00	9.85	1.81	1.87
Synthetic	Operational	157.25	1.00	43.77	37.89	5.15
Synthetic	Fine	157.25	1.00	6.01	1.67	1.47

Table 6: Rollback ablation: average change induced by enabling rollback-sensitive semantic splitting. Positive Δ means an increase; negative Δ in rollback-predicate depth means a cheaper rollback-sensitive predicate.

Family	Granularity / method	Δ classes	Δ expected depth	Δ RBAcc depth
Hand	Coarse / B3	+0.57	+0.30	−0.63
Hand	Coarse / B4	+0.57	+0.02	−0.19
Hand	Coarse / B5-TREE	+0.57	+0.02	−0.36
Hand	Operational / B3	+1.00	+0.70	−1.47
Hand	Operational / B4	+1.00	+0.43	−1.98
Hand	Operational / B5-TREE	+1.00	+0.65	−0.21
Synthetic	Coarse / B3	+1.00	+0.89	−6.58
Synthetic	Coarse / B4	+1.00	+0.06	−0.84
Synthetic	Coarse / B5-TREE	+1.00	+0.07	−0.65
Synthetic	Operational / B3	+1.00	+0.90	−83.73
Synthetic	Operational / B4	+1.00	+0.45	−73.28
Synthetic	Operational / B5-TREE	+1.00	+0.20	−7.89

under B1, whereas the dedicated binary-threshold layout B2 fixes it at 1.00 by construction.

Second, rollback-sensitive predicates can be improved substantially, but the best method depends on the semantic granularity. Under the synthetic operational labeling, B5-TREE is dramatically better for RollbackAcc(s) than either flat interval baseline (5.15 versus 43.77 for B3 and 37.89 for B4). Under the synthetic fine labeling, both B4 and B5-TREE work well, with B5-TREE only slightly lower (1.47 versus 1.67).

To isolate the effect of adding rollback-aware semantic distinctions, we also use the supplied rollback-ablation summaries. Table 6 reports average deltas when the rollback split is enabled in the coarse and operational labelings.

The rollback ablation makes it clear that the added distinction is not merely cosmetic. In the operational labeling on the synthetic benchmarks, enabling rollback splitting adds only one semantic class on average, yet it dramatically reduces rollback-predicate expected depth for the flat interval methods. Even where the gain is smaller, the structural cost remains modest: roughly one

extra class and only a limited increase in overall semantic-dispatch depth. That is strong evidence that rollback-aware semantic distinctions are worth modeling explicitly when rollback-sensitive predicates matter.

9.4 Profile mismatch and the limits of DAG sharing

Because B4 depends on profile order, the repository includes profile-shift experiments. The effect is visible, though not catastrophic. On the hand-designed benchmarks, B4 expected depth rises from 2.48 to 2.85 in coarse semantics and from 2.73 to 3.25 in fine semantics when moving from matched to shifted profiles. On the synthetic benchmarks, the corresponding changes are 4.24 to 4.41 (coarse) and 4.73 to 5.41 (fine). These increases are real, but even under shifted profiles B4 remains well below the matched-profile B3 baseline in the same benchmark families. In other words, profile-aware ordering is sensitive to profile quality, but the supplied shifts do not wipe out its advantage.

The B5-DAG results are informative in a different way because they are negative. In every completed B5-DAG configuration in the supplied summaries, the number of shared internal nodes is zero. On this benchmark suite, the hierarchical comparison structures induced by the semantic partition simply do not contain repeated internal subtrees worth sharing. As a result, B5-DAG offers no structural advantage over B5-TREE here. We therefore keep the tree variant as the main hierarchical representative and treat the DAG outcome as a benchmark-conditioned limitation rather than as evidence for a stronger method.

9.5 Discussion

Taken together, the experiments sketch a fairly clear design space for generated semantic dispatch.

1. **Semantic layout is a real structural degree of freedom.** Once the DFA is fixed, changing only the numbering can materially change the geometry of semantic predicates and multi-class dispatch.

2. **Flat interval encodings are viable but not uniformly good.** B3 gives the cleanest one-interval-per-class realization, yet its depth scales directly with semantic richness.

3. **Profile-aware ordering and hierarchical trees solve different problems.** B4 is the best flat method for expected depth under the supplied profiles. B5-TREE is the best method for worst-case depth, especially on larger and richer semantic partitions.

4. **Rollback-aware labeling changes the structural picture.** Explicit rollback splitting can make rollback-sensitive predicates substantially cheaper with only small increases in the semantic alphabet size.

5. **Not every implementation idea pays off.** The DAG-sharing variant did not help on the present suite, so it should not be presented as an established advantage of the framework.

All of these are structural conclusions about semantic state-layout synthesis and about the implementation forms induced by each layout family.

10 Related Work and Positioning

This work sits at the intersection of automata-based code generation, representation choices in generated implementations, and state encoding [12, 8].

Standard compiler and automata texts describe DFA-based lexical analysis, token-carrying accepting states, and longest-match scanning [1, 2, 4]. Mature lexer generators and finite-state compilers operationalize that setting in different ways, but still within the same underlying DFA-based model [6, 7, 8]. Everything in this paper stays within that familiar setting and preserves those scanner semantics.

The closest conceptual relative is the hardware and logic-synthesis literature on *state assignment*, where symbolic FSM states are encoded for implementation reasons [3, 13, 14]. The connection is straightforward at the level of representation choice: in both settings, symbolic states must be mapped to implementation-level encodings. The difference lies in the artifact being generated. Here the focus is software, and more specifically the semantic classification logic attached to DFA states inside generated recognizers.

Generated scanners already use a wide range of implementation strategies, including table-driven execution, direct-coded transitions, compressed transition tables, and specialized accept-state handling [4, 5, 6, 7, 9, 10, 11]. The present proposal is complementary to those techniques rather than a replacement for them. It targets a different step in the pipeline: how a code generator should organize numeric state identifiers and compile semantic dispatch once the automaton and its state semantics are fixed.

Against that background, the paper's contribution is to formulate semantic state-layout synthesis as an explicit software-construction problem for semantically labeled DFA states and to evaluate the resulting design space through structural properties of the generated artifact.

11 Limitations and Threats to Validity

The scope of the study is intentionally limited, and the main limitations are the following.

1. **This is mainly a representation study.** The method changes numbering and dispatch structure, not recognition power, transition semantics, or state count.

2. **The paper's evidence is structural, not end-to-end performance evidence.** The supplied artefacts also contain runtime-oriented measurements, but we exclude them from the main validation because throughput is target dependent and the hardware-counter data were incomplete in the provided environment. Target-specific runtime studies remain future work.

3. **B4 depends on profile quality.** The profile-shift experiments show degradation under mismatch, even if the supplied mismatches do not erase the gain entirely.

4. **B5-TREE trades depth for code size.** The tree layout is structurally stronger for worst-case depth, but it emits larger classifier source. Whether that is desirable depends on the target and code generator.

5. **B5-DAG did not help on the studied suite.** The absence of shared internal subtrees on these benchmarks means that DAG-style sharing should not be presented as a general empirical advantage.

6. **The benchmark suite is lexer oriented.** The conclusions apply directly to generated lexical analyzers and recognizers with similar semantic state structure. Other application domains may induce different semantic hierarchies and different sharing opportunities.

7. **Longest-match and rollback logic remain necessary.** Structured semantic layouts do not eliminate last-accept tracking, fallback logic, or token-priority handling.

A natural next step is a genuinely target-specific runtime study, especially for embedded systems, constrained hardware, or deployment-specific code generators where code shape, branching structure, and compact predicates may interact more directly with performance.

12 Conclusion

This paper treats numeric state layout as an explicit synthesis problem in automata-based code generation. For DFAs with semantically labeled states, a generator can preserve the automaton exactly and still choose a representation that compiles semantic decisions into intervals, threshold chains, or hierarchical comparison structures.

On the theoretical side, the paper establishes the basic representation facts that make this possible: arbitrary renumbering preserves the recognized language, one-interval-per-class layouts always exist, profile-aware ordering minimizes expected depth for flat chains, and hierarchical partitions induce nested numeric layouts that fit tree-based dispatch naturally.

On the experimental side, the lexer-oriented benchmark suite shows what those constructions look like in practice. Flat contiguous layouts are easy to realize, but their worst-case depth grows poorly as semantic granularity increases. Frequency-aware ordering materially lowers expected depth without changing interval complexity. Hierarchical comparison trees reduce worst-case depth substantially, especially on larger and finer semantic partitions, though they emit larger classifier code. Rollback-aware semantic splitting is empirically worthwhile when rollback-sensitive predicates matter. The DAG-sharing variant, by contrast, does not provide a benefit on this benchmark family.

Taken together, these results support a straightforward software-engineering claim: even after language recognition has been fixed, generated automata still contain meaningful representation choices, and semantic state layout is one of them. The central evidence is structural—the geometry of semantic partitions, comparison depth, predicate organization, and emitted classifier shape—since those are properties of the generated artifact itself.

Conflict of Interest

The author declares no conflict of interests.

Data Availability Statement

Data for the benchmarks as well as the code used to run the experiments of this article can be found in the following link : https://github.com/ReedA2022/semantic-state-layout-pipeline

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[2] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[3] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[4] M. E. Lesk and E. Schmidt. *Lex—A Lexical Analyzer Generator*. Bell Laboratories, 1975.

[5] V. Paxson. *Flex: A Fast Scanner Generator*. Version 2.5, 1995.

[6] P. Bumbulis and D. D. Cowan. RE2C: A more versatile scanner generator. *ACM Letters on Programming Languages and Systems*, 2(1–4):70–84, 1993.

[7] U. Trofimovich. RE2C: A lexer generator based on lookahead-TDFA. *Software Impacts*, 6:100027, 2020.

[8] M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[9] K. Brouwer, W. Gellerich, and E. Pl"odereder. Myths and facts about the efficient implementation of finite automata and lexical analysis. In *Proceedings of the 7th International Conference on Compiler Construction (CC)*, pages 1–15, 1998.

[10] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. S. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the ACM SIGCOMM 2006 Conference*, pages 339–350, 2006.

[11] M. Becchi and P. Crowley. A-DFA: A time- and space-efficient DFA compression algorithm for fast regular expression evaluation. *ACM Transactions on Architecture and Code Optimization*, 10(1):4:1–4:26, 2013.

[12] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[13] G. De Micheli, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 4:269–285, 1985.

[14] T. Villa and A. L. Sangiovanni-Vincentelli. NOVA: state assignment of finite state machines for optimal two-level logic implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(9):905–924, 1990.