ripgrep crates/regex/src/literal.rs: Code Companion

Reference code for the Literal Optimization lecture. Sections correspond to the lecture document

Section 1: The Inner Literal Strategy

The is_accelerated() check queries whether the regex engine has its own internal optimizations. The contains_word_unicode() check identifies patterns with \b in Unicode mode, which can cause the regex engine to fall back to slower paths.

Section 2: The Extractor Architecture

The TSeq wrapper ("tagged sequence") tracks whether the sequence represents a prefix position in the regex. This metadata is stripped away in extract_untagged before returning the final Seq

Section 3: Handling Concatenation

```
/// Extract from concatenation via cross product of sub-expressions
fn extract_concat< a, I: Iterator Item = & a Hir>>(self, it: I) -> TSeq {
    let mut seq = TSeq::singleton(Literal::exact(vec![]));
    let mut prev: Option<TSeq> = None, // Tracks best candidate so far

for hir in it {
        // Once all literals are inexact, cross product is a no-op
        if seq.is_inexact() {
            // Empty sequence = impossible match, bail early
            if seq.is_empty() {
                return seq;
            }
            // Found something really good? Stop looking
            if seq.is_really_good() {
                  return seq;
            }
            // Save current best, start fresh for next segment
            prev = Some(match prev {
                None => seq,
                Some(prev) => prev.choose(seq), // Pick better of two
            });
            seq = TSeq::singleton(Literal::exact(vec![]));
            seq.make_not_prefix(); // No longer at pattern start
            }
            // Cross product combines current seq with next sub-expression
            seq = self.cross(seq, self.extract(hir));
        }
        // Return best of saved candidate and final sequence
        if let Some(prev) = prev { prev.choose(seq) } else { seq }
}
```

The prev variable implements the "keep the best so far" strategy. When the current sequence becomes inexact (hitting a gap like [a-z]+), we save it and start fresh, later choosing whichever is better.

Section 4: Handling Alternation

The magic number 4 for keep_first_bytes is an abstraction leak from the Teddy SIMD algorithm, which efficiently searches for literals up to 4 bytes. Trimming to this length preserves usefulness even when the full literals would exceed limits.

Section 5: Handling Repetition

```
assert!(min > 0);
```

```
// Unbounded: a+ -> inexact "a" hir::Repetition { .. } => {
subseq.make_inexact(); subseq } }}
```

The greedy flag affects the order of alternatives in the union. For a* (greedy), we prefer matching a first; for a*? (lazy), we prefer the empty match first. This ordering propagates to how the literals are used in searching.

Quick Reference

TSeq (Tagged Sequence) Key Methods

Method	
<pre>is_exact()</pre>	
<pre>is_inexact()</pre>	At least one literal is partial
<pre>is_finite()</pre>	
is_good()	
is_really_good()	
choose(other)	
<pre>make_not_prefix()</pre>	Mark as not at pattern start

Heuristic Thresholds

```
// "Good" for acceleration:
min_literal_len >= 2 && count <= 64
// Or if very short literals:
min_literal_len <= 1 && count <= 3

// "Really good" (short-circuit worthy):
min_literal_len >= 3 && count <= 8</pre>
```

Poisonous Literals

```
/// Literals likely to match too frequently
fn is_poisonous(lit: &Literal) -> bool {
    use regex_syntax::hir::literal::rank;
    // Empty string or high-frequency single byte
    lit.is_empty() || (lit.len() == 1 && rank(lit.as_bytes()[0]) >= 250)
}
```

The rank function returns a frequency score (0-255) based on how commonly a byte appears in typical text. Rank 250+ includes space, newline, and other ubiquitous characters.

Extraction Limits

Limit	Default	
limit_class		Max chars from [abc]
limit_repeat		Max unroll iterations for a{n}
limit_literal_len		Max bytes per literal
limit_total		Max literals in sequence