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

Reference code for the Implementing Matcher lecture. Sections correspond to the lecture document.

Section 1: The Builder Pattern Completed

```rust
/// A builder for constructing a `Matcher` using regular expressions.
///
/// This builder re-exports many of the same options found on the regex crate's
/// builder, in addition to a few other options such as smart case, word
/// matching and the ability to set a line terminator which may enable certain
/// types of optimizations.
#[derive(Clone, Debug)]  // Clone enables reuse, Debug enables logging
pub struct RegexMatcherBuilder {
    config: Config,  // Delegates storage to Config struct
}

impl Default for RegexMatcherBuilder {
    fn default() -> RegexMatcherBuilder {
        RegexMatcherBuilder::new()
    }
}

impl RegexMatcherBuilder {
    /// Create a new builder for configuring a regex matcher.
    pub fn new() -> RegexMatcherBuilder {
        RegexMatcherBuilder { config: Config::default() }
    }
}
```

The builder holds a single `Config` field, creating a clean separation between the fluent API (builder methods) and the actual configuration storage. The `Default` trait delegates to `new()`, ensuring consistent initialization.

Section 2: From Pattern to Matcher

```rust
/// Build a new matcher using the current configuration for the provided
/// pattern.
pub fn build(&self, pattern: &str) -> Result<RegexMatcher, Error> {
    self.build_many(&[pattern])  // Single pattern delegates to multi-pattern
}

/// Build a new matcher using the current configuration for the provided
/// patterns. The resulting matcher behaves as if all of the patterns
/// given are joined together into a single alternation.
pub fn build_many<P: AsRef<str>>(
    &self,
    patterns: &[P],
) -> Result<RegexMatcher, Error> {
    // Step 1: Build configured HIR from patterns
    let mut chir = self.config.build_many(patterns)?;

    // Step 2: Apply whole-line or word transformations
    // 'whole_line' is a strict subset of 'word', so when it is enabled,
    // we don't need to both with any specific to word matching.
    if chir.config().whole_line {
        chir = chir.into_whole_line();
    } else if chir.config().word {
        chir = chir.into_word();
    }

    // Step 3: Compile HIR to actual regex automaton
    let regex = chir.to_regex()?;
    log::trace!("final regex: {:?}", chir.hir().to_string());

    // ... optimization setup continues in next section
}
```

The generic bound `P: AsRef<str>` allows passing `&[&str]`, `&[String]`, or any slice of string-like types. The `into_whole_line()` and `into_word()` methods consume and transform the HIR.

Section 3: The Fast-Line Optimization

```rust
pub fn build_many<P: AsRef<str>>(
    &self,
    patterns: &[P],
) -> Result<RegexMatcher, Error> {
    // ... pattern compilation from previous section ...

    let non_matching_bytes = chir.non_matching_bytes();

    // If we can pick out some literals from the regex, then we might be
    // able to build a faster regex that quickly identifies candidate
    // matching lines. For example, for a regex like `\w+foo\w+`, we can
    // look for `foo`, and when a match is found, look for the line
    // containing `foo` and then run the original regex on only that line.
    let fast_line_regex = InnerLiterals::new(&chir, &regex).one_regex()?;

    // We override the line terminator in case the configured HIR doesn't
    // support it.
    let mut config = self.config.clone();
    config.line_terminator = chir.line_terminator();

    // Construct the final matcher with all four components
    Ok(RegexMatcher {
        config,              // Preserved for line_terminator queries
        regex,               // The main matching engine
        fast_line_regex,     // Optional optimization (may be None)
        non_matching_bytes   // Bytes that can never appear in matches
    })
}
```

The `Option<Regex>` type for `fast_line_regex` makes the optimization's optionality explicit—callers must handle both cases.

Section 4: Builder Methods as Documentation

```
/// Whether to enable "smart case" or not.
///
/// When smart case is enabled, the builder will automatically enable
/// case insensitive matching based on how the pattern is written. Namely,
/// case insensitive mode is enabled when both of the following things
/// are true:
///
/// 1. The pattern contains at least one literal character.
/// 2. Of the literals in the pattern, none of them are considered to be
///    uppercase according to Unicode.
pub fn case_smart(&mut self, yes: bool) -> &mut RegexMatcherBuilder {
    self.config.case_smart = yes;
    self  // Returns &mut Self for method chaining
}

/// Set an ASCII line terminator for the matcher.
///
/// The purpose of setting a line terminator is to enable a certain class
/// of optimizations that can make line oriented searching faster. Namely,
/// when a line terminator is enabled, then the builder will guarantee that
/// the resulting matcher will never be capable of producing a match that
/// contains the line terminator.
///
/// If the aforementioned guarantee cannot be made because of how the pattern
/// was written, then the builder will return an error. For example, the
/// pattern `a\nb` will result in an error since the `\n` cannot be easily
/// removed without changing the fundamental intent of the pattern.
pub fn line_terminator(
    &mut self,
    line_term: Option<u8>,
) -> &mut RegexMatcherBuilder {
    self.config.line_terminator = line_term.map(LineTerminator::byte);
    self
}

/// Require that all matches occur on word boundaries.
///
/// Enabling this option is subtly different than putting `\b` assertions
/// on both sides of your pattern. A `\b` assertion requires that one side
/// match a word character while the other match a non-word character.
/// This option merely requires that one side match a non-word character.
pub fn word(&mut self, yes: bool) -> &mut RegexMatcherBuilder {
    self.config.word = yes;
    self
}
```

Each builder method modifies the config and returns `&mut Self`, enabling fluent chaining like `builder.case_smart(true).word(true).build(pattern)`.

Section 5: The RegexMatcher Structure

```rust
/// An implementation of the `Matcher` trait using Rust's standard regex
/// library.
#[derive(Clone, Debug)]
pub struct RegexMatcher {
    /// The configuration specified by the caller.
    config: Config,

    /// The regular expression compiled from the pattern provided by the
    /// caller.
    regex: Regex,  // regex_automata::meta::Regex

    /// A regex that never reports false negatives but may report false
    /// positives that is believed to be capable of being matched more quickly
    /// than `regex`. Typically, this is a single literal or an alternation
    /// of literals.
    fast_line_regex: Option<Regex>,

    /// A set of bytes that will never appear in a match.
    non_matching_bytes: ByteSet,
}

impl RegexMatcher {
    /// Create a new matcher from the given pattern using the default
    /// configuration.
    pub fn new(pattern: &str) -> Result<RegexMatcher, Error> {
        RegexMatcherBuilder::new().build(pattern)
    }

    /// Create a new matcher from the given pattern using the default
    /// configuration, but matches lines terminated by `\n`.
    ///
    /// This is meant to be a convenience constructor for enabling
    /// line-oriented optimizations.
    pub fn new_line_matcher(pattern: &str) -> Result<RegexMatcher, Error> {
        RegexMatcherBuilder::new().line_terminator(Some(b'\n')).build(pattern)
    }
}
```

The four fields each serve distinct purposes: `config` for queries, `regex` for matching, `fast_line_regex` for optimization, and `non_matching_bytes` for skipping.

Section 6: Implementing the Matcher Trait

```rust
use grep_matcher::{
    ByteSet, Captures, LineMatchKind, LineTerminator, Match, Matcher, NoError,
};

impl Matcher for RegexMatcher {
    type Captures = RegexCaptures;
    type Error = NoError;  // Regex operations are infallible once compiled

    #[inline]
    fn find_at(
        &self,
        haystack: &[u8],
        at: usize,
    ) -> Result<Option<Match>, NoError> {
        // Create input spanning from 'at' to end of haystack
        let input = Input::new(haystack).span(at..haystack.len());
        // Convert regex_automata::Match to grep_matcher::Match
        Ok(self.regex.find(input).map(|m| Match::new(m.start(), m.end())))
    }

    #[inline]
    fn new_captures(&self) -> Result<RegexCaptures, NoError> {
        Ok(RegexCaptures::new(self.regex.create_captures()))
    }

    #[inline]
    fn capture_count(&self) -> usize {
        self.regex.captures_len()
    }

    #[inline]
    fn capture_index(&self, name: &str) -> Option<usize> {
        // Look up named capture group in pattern 0
        self.regex.group_info().to_index(PatternID::ZERO, name)
    }

    #[inline]
    fn non_matching_bytes(&self) -> Option<&ByteSet> {
        Some(&self.non_matching_bytes)
    }

    #[inline]
    fn line_terminator(&self) -> Option<LineTerminator> {
        self.config.line_terminator
    }
}
```

The associated type `Error = NoError` indicates that regex operations cannot fail at runtime—all validation happens at compile time.

Section 7: The Fast-Line Search in Action

```
impl Matcher for RegexMatcher {
    // ... other methods ...

    #[inline]
    fn find_candidate_line(
        &self,
        haystack: &[u8],
    ) -> Result<Option<LineMatchKind>, NoError> {
        Ok(match self.fast_line_regex {
            // Fast path: use optimized literal search
            Some(ref regex) => {
                let input = Input::new(haystack);
                regex
                    .search_half(&input)  // Only need match position, not full
match
                    .map(|hm| LineMatchKind::Candidate(hm.offset()))
            }
            // Slow path: run full regex
            None => {
                self.shortest_match(haystack)?.map(LineMatchKind::Confirmed)
            }
        })
    }

    #[inline]
    fn shortest_match_at(
        &self,
        haystack: &[u8],
        at: usize,
    ) -> Result<Option<usize>, NoError> {
        let input = Input::new(haystack).span(at..haystack.len());
        // search_half returns just the end position, not the full match
        Ok(self.regex.search_half(&input).map(|hm| hm.offset()))
    }
}
```

`LineMatchKind::Candidate` indicates a potential match that needs verification; `LineMatchKind::Confirmed` indicates a definite match. The `search_half` method is faster than `find` because it only determines *if* a match exists and where it ends.

Section 8: Captures Implementation

```rust
use regex_automata::util::captures::Captures as AutomataCaptures;

/// Represents the match offsets of each capturing group in a match.
///
/// The first, or `0`th capture group, always corresponds to the entire match.
/// Note that not all capturing groups are guaranteed to be present in a match.
#[derive(Clone, Debug)]
pub struct RegexCaptures {
    /// Where the captures are stored.
    caps: AutomataCaptures,
}

impl Captures for RegexCaptures {
    #[inline]
    fn len(&self) -> usize {
        self.caps.group_info().all_group_len()
    }

    #[inline]
    fn get(&self, i: usize) -> Option<Match> {
        // Convert from regex_automata's Span to grep_matcher's Match
        self.caps.get_group(i).map(|sp| Match::new(sp.start, sp.end))
    }
}

impl RegexCaptures {
    #[inline]
    pub(crate) fn new(caps: AutomataCaptures) -> RegexCaptures {
        RegexCaptures { caps }
    }

    #[inline]
    pub(crate) fn captures_mut(&mut self) -> &mut AutomataCaptures {
        &mut self.caps
    }
}
```

The `pub(crate)` visibility restricts `new` and `captures_mut` to the crate—external users interact through the `Captures` trait only.

Section 9: Iteration Implementation

```rust
impl Matcher for RegexMatcher {
    #[inline]
    fn try_find_iter<F, E>(
        &self,
        haystack: &[u8],
        mut matched: F,
    ) -> Result<Result<(), E>, NoError>
    where
        F: FnMut(Match) -> Result<bool, E>,
    {
        for m in self.regex.find_iter(haystack) {
            match matched(Match::new(m.start(), m.end())) {
                Ok(true) => continue,       // Keep searching
                Ok(false) => return Ok(Ok(())),  // Caller wants to stop
                Err(err) => return Ok(Err(err)), // Caller encountered error
            }
        }
        Ok(Ok(()))
    }

    #[inline]
    fn captures_at(
        &self,
        haystack: &[u8],
        at: usize,
        caps: &mut RegexCaptures,
    ) -> Result<bool, NoError> {
        let input = Input::new(haystack).span(at..haystack.len());
        let caps = caps.captures_mut();  // Get mutable access to inner storage
        self.regex.search_captures(&input, caps);
        Ok(caps.is_match())
    }
}
```

The `Result<Result<(), E>, NoError>` return type separates regex errors (outer `Result`, always `Ok`) from callback errors (inner `Result`). The callback returns `bool` to indicate whether iteration should continue.

Quick Reference

RegexMatcher Fields

Field	Type	Purpose

`config`	`Config`	Stores line terminator for queries
`regex`	`Regex`	Main matching engine
`fast_line_regex`	`Option<Regex>`	Literal-based optimization
`non_matching_bytes`	`ByteSet`	Bytes that never match

Key Matcher Trait Methods

Method	**Returns**	**Purpose**
`find_at`	`Option<Match>`	Find first match at offset
`shortest_match_at`	`Option<usize>`	Find match end position only
`find_candidate_line`	`Option<LineMatchKind>`	Fast pre-filtering
`captures_at`	`bool`	Populate capture groups
`try_find_iter`	`Result<(), E>`	Iterate with early exit

LineMatchKind Variants

```
enum LineMatchKind {
    Candidate(usize),   // Potential match, needs verification
    Confirmed(usize),   // Definite match
}
```

Builder Method Chaining Pattern

```
RegexMatcherBuilder::new()
    .case_smart(true)
    .word(true)
    .line_terminator(Some(b'\n'))
    .build(pattern)?
```