ripgrep crates/globset/src/glob.rs: Code Companion

Reference code for the Glob Implementation lecture. Sections correspond to the lecture document.

Section 1: The Match Strategy Optimization

```rust
/// Describes a matching strategy for a particular pattern.
///
/// This provides a way to more quickly determine whether a pattern matches
/// a particular file path in a way that scales with a large number of
/// patterns.
#[derive(Clone, Debug, Eq, PartialEq)]
pub(crate) enum MatchStrategy {
    /// Pattern matches if and only if the entire file path matches this literal
    Literal(String),
    /// Pattern matches if and only if the file path's basename matches this
literal
    BasenameLiteral(String),
    /// Pattern matches if and only if the file path's extension matches this
literal
    Extension(String),
    /// Pattern matches if and only if this prefix literal is a prefix of the
path
    Prefix(String),
    /// Pattern matches if and only if this suffix literal is a suffix of the
path
    Suffix {
        suffix: String,
        /// Whether this must start at the beginning of a path component
        component: bool,
    },
    /// Extension match is necessary but NOT sufficient - still need regex
    RequiredExtension(String),
    /// No optimization available - use full regex matching
    Regex,
}

impl MatchStrategy {
    /// Returns a matching strategy for the given pattern.
    /// Tries optimizations in order of specificity/speed.
    pub(crate) fn new(pat: &Glob) -> MatchStrategy {
        // Try each optimization in order - first match wins
        if let Some(lit) = pat.basename_literal() {
            MatchStrategy::BasenameLiteral(lit)
        } else if let Some(lit) = pat.literal() {
            MatchStrategy::Literal(lit)
        } else if let Some(ext) = pat.ext() {
            MatchStrategy::Extension(ext)
        } else if let Some(prefix) = pat.prefix() {
            MatchStrategy::Prefix(prefix)
        } else if let Some((suffix, component)) = pat.suffix() {
            MatchStrategy::Suffix { suffix, component }
        } else if let Some(ext) = pat.required_ext() {
            MatchStrategy::RequiredExtension(ext)
        } else {
            MatchStrategy::Regex
        }
    }
}
```

The ordering in `new` reflects optimization priority: basename literal is checked first because patterns like `**/Makefile` are extremely common in `.gitignore` files and can be matched with a single string comparison.

Section 2: The Glob Type and Builder Pattern

```rust
/// Glob represents a successfully parsed shell glob pattern.
#[derive(Clone, Eq)]
pub struct Glob {
    glob: String,       // Original pattern text
    re: String,         // Generated regex string
    opts: GlobOptions,  // Parsing/matching options
    tokens: Tokens,     // Parsed token representation
}

// Only compare pattern and options - not internal representation
impl PartialEq for Glob {
    fn eq(&self, other: &Glob) -> bool {
        self.glob == other.glob && self.opts == other.opts
    }
}

impl std::hash::Hash for Glob {
    fn hash<H: std::hash::Hasher>(&self, state: &mut H) {
        self.glob.hash(state);
        self.opts.hash(state);
    }
}

// Compact debug by default, expanded with {:#?}
impl std::fmt::Debug for Glob {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        if f.alternate() {
            // {:#?} shows all internals for debugging
            f.debug_struct("Glob")
                .field("glob", &self.glob)
                .field("re", &self.re)
                .field("opts", &self.opts)
                .field("tokens", &self.tokens)
                .finish()
        } else {
            // {:?} shows just the pattern
            f.debug_tuple("Glob").field(&self.glob).finish()
        }
    }
}
```

```rust
/// A builder for a pattern - lifetime tied to input pattern string.
#[derive(Clone, Debug)]
pub struct GlobBuilder<'a> {
    glob: &'a str,        // Borrowed pattern text
    opts: GlobOptions,    // Accumulated options
}

#[derive(Clone, Copy, Debug, Eq, Hash, PartialEq)]
struct GlobOptions {
    case_insensitive: bool,
    literal_separator: bool,   // When true, * and ? won't match /
    backslash_escape: bool,    // Platform-dependent default
    empty_alternates: bool,    // Allow {,a} to match "" and "a"
    allow_unclosed_class: bool,
}

impl GlobOptions {
    fn default() -> GlobOptions {
        GlobOptions {
            case_insensitive: false,
            literal_separator: false,
            // On Unix, \ is escape; on Windows, \ is path separator
            backslash_escape: !is_separator('\\'),
            empty_alternates: false,
            allow_unclosed_class: false,
        }
    }
}
```

The `backslash_escape` default uses `is_separator('\\')` to detect the platform: on Windows where `\` is a path separator, escaping is disabled by default.

Section 3: The Token Representation

```rust
/// Wrapper type around Vec<Token> for type clarity
#[derive(Clone, Debug, Default, Eq, PartialEq)]
struct Tokens(Vec<Token>);

// Deref allows using Tokens as a Vec<Token>
impl std::ops::Deref for Tokens {
    type Target = Vec<Token>;
    fn deref(&self) -> &Vec<Token> {
        &self.0
    }
}

impl std::ops::DerefMut for Tokens {
    fn deref_mut(&mut self) -> &mut Vec<Token> {
        &mut self.0
    }
}

#[derive(Clone, Debug, Eq, PartialEq)]
enum Token {
    /// A literal character to match exactly
    Literal(char),
    /// ? - matches any single character
    Any,
    /// * - matches zero or more characters
    ZeroOrMore,
    /// **/ at pattern start - matches any prefix path
    RecursivePrefix,
    /// /** at pattern end - matches any suffix path
    RecursiveSuffix,
    /// /**/ in middle - must match at least one /
    RecursiveZeroOrMore,
    /// Character class like [a-z] or [!0-9]
    Class { negated: bool, ranges: Vec<(char, char)> },
    /// Alternation like {foo,bar,baz}
    Alternates(Vec<Tokens>),
}
```

The three recursive variants (`RecursivePrefix`, `RecursiveSuffix`, `RecursiveZeroOrMore`) capture the semantic differences of `**` in different positions, which affects both regex generation and optimization extraction.

Section 4: Strategy Extraction Methods

```rust
impl Glob {
    /// Returns pattern as literal only if it contains no wildcards.
    fn literal(&self) -> Option<String> {
        // Case insensitive requires regex machinery
        if self.opts.case_insensitive {
            return None;
        }
        let mut lit = String::new();
        for t in &*self.tokens {
            // Bail if anything other than a literal
            let Token::Literal(c) = *t else { return None };
            lit.push(c);
        }
        if lit.is_empty() { None } else { Some(lit) }
    }

    /// Returns extension for patterns like *.ext or **/*.ext
    fn ext(&self) -> Option<String> {
        if self.opts.case_insensitive {
            return None;
        }
        // Check for optional **/ prefix
        let start = match *self.tokens.get(0)? {
            Token::RecursivePrefix => 1,
            _ => 0,
        };
        // Must have * that can match path separators
        match *self.tokens.get(start)? {
            Token::ZeroOrMore => {
                // Without **, * must be able to match /
                if start == 0 && self.opts.literal_separator {
                    return None;
                }
            }
            _ => return None,
        }
        // Must have literal dot
        match *self.tokens.get(start + 1)? {
            Token::Literal('.') => {}
            _ => return None,
        }
        // Rest must be literals without . or /
        let mut lit = ".".to_string();
        for t in self.tokens[start + 2..].iter() {
            match *t {
                Token::Literal('.') | Token::Literal('/') => return None,
                Token::Literal(c) => lit.push(c),
                _ => return None,
            }
        }
        if lit.is_empty() { None } else { Some(lit) }
    }

    /// Returns tokens for basename-only matching (e.g., **/filename)
```

```
    fn basename_tokens(&self) -> Option<&[Token]> {        if
self.opts.case_insensitive {            return None;        }        let start =
match *self.tokens.get(0)? {            Token::RecursivePrefix => 1,            _
=> {                // Without ** prefix, can't assume basename-only is correct
return None;            }        };        if self.tokens[start..].is_empty() {
return None;        }        // Check remaining tokens don't escape basename
for t in self.tokens[start..].iter() {            match *t {
Token::Literal('/') => return None,                Token::Literal(_) => {} // OK
Token::Any | Token::ZeroOrMore => {                    if
!self.opts.literal_separator {                        // * or ? could match /,
escaping basename                        return None;                    }
}                Token::RecursivePrefix                | Token::RecursiveSuffix
| Token::RecursiveZeroOrMore => return None,                Token::Class { .. } |
Token::Alternates(..) => {                    // Too complex - give up on
optimization                    return None;                }            }
}        // Return slice to avoid allocation        Some(&self.tokens[start..])
}}
```

Note that `basename_tokens` returns `&[Token]` rather than allocating a new `Vec`—this is efficient since we're just providing a view into existing data.

Section 5: The Parser Structure

```rust
struct Parser<'a> {
    /// The glob pattern being parsed
    glob: &'a str,
    /// Stack tracking where alternation groups begin
    alternates_stack: Vec<usize>,
    /// Active branches - tokens added to last one
    branches: Vec<Tokens>,
    /// Character iterator with peek capability
    chars: std::iter::Peekable<std::str::Chars<'a>>,
    /// Previous character (for context like detecting /**)
    prev: Option<char>,
    /// Current character
    cur: Option<char>,
    /// Tracks if we found unclosed [ (optimization to avoid quadratic time)
    found_unclosed_class: bool,
    /// Options that may influence parsing
    opts: &'a GlobOptions,
}

impl<'a> Parser<'a> {
    fn parse(&mut self) -> Result<(), Error> {
        while let Some(c) = self.bump() {
            match c {
                '?' => self.push_token(Token::Any)?,
                '*' => self.parse_star()?,
                '[' if !self.found_unclosed_class => self.parse_class()?,
                '{' => self.push_alternate()?,
                '}' => self.pop_alternate()?,
                ',' => self.parse_comma()?,
                '\\' => self.parse_backslash()?,
                c => self.push_token(Token::Literal(c))?,
            }
        }
        Ok(())
    }

    fn push_alternate(&mut self) -> Result<(), Error> {
        // Mark where this alternation starts
        self.alternates_stack.push(self.branches.len());
        // Start first branch of this alternation
        self.branches.push(Tokens::default());
        Ok(())
    }

    fn pop_alternate(&mut self) -> Result<(), Error> {
        let Some(start) = self.alternates_stack.pop() else {
            return Err(self.error(ErrorKind::UnopenedAlternates));
        };
        // Collect all branches since start into Alternates token
        let alts = Token::Alternates(self.branches.drain(start..).collect());
        self.push_token(alts)?;
        Ok(())
    }
```

```
    fn parse_comma(&mut self) -> Result<(), Error> {        // Comma is special
only inside alternation        if self.alternates_stack.is_empty() {
self.push_token(Token::Literal(','))        } else {            // Start new
branch at same nesting level            Ok(self.branches.push(Tokens::default()))
}    }}
```

The `alternates_stack` and `branches` work together: when `{` is encountered, the current branch count is pushed to the stack. When `}` is seen, all branches since that index are collected into an `Alternates` token.

Quick Reference

Token Types

Token	Pattern	Meaning
`Literal(c)`	`a`, `b`, `/`	Match exact character
`Any`	`?`	Match any single character
`ZeroOrMore`	`*`	Match zero or more characters
`RecursivePrefix`	`**/`	Match any directory prefix
`RecursiveSuffix`	`/**`	Match any directory suffix
`RecursiveZeroOrMore`	`/**/`	Match one or more directories
`Class`	`[a-z]`	Match character in class
`Alternates`	`{a,b}`	Match any alternative

Match Strategy Selection

```
Pattern           → Strategy
──────────────────────────────────────
**/Makefile       → BasenameLiteral
foo/bar           → Literal
*.rs              → Extension
build/**          → Prefix
**/test.js        → Suffix (component=true)
**/*.test.js      → RequiredExtension
**/*_test/**/*.c  → Regex
```

Key Builder Options

Option	Default	Effect
`case_insensitive`	`false`	Enable case-insensitive matching
`literal_separator`	`false`	Require `/` to match path separator
`backslash_escape`	platform	Enable `\` as escape character
`empty_alternates`	`false`	Allow `{,a}` to match empty string
`allow_unclosed_class`	`false`	Treat `[` without `]` as literal