Capture Group Interpolation: Code Companion

Reference code for the Capture Group Interpolation lecture. Sections correspond to the lecture document.

Section 1: The Interpolation Function Signature

```
/// Interpolate capture references in `replacement` and write the interpolation
/// result to `dst`. References in `replacement` take the form of $N or $name,
/// where `N` is a capture group index and `name` is a capture group name.
#[inline]
pub fn interpolate<A, N>(
    mut replacement: &[u8],       // Mutable slice - will be narrowed as we parse
    mut append: A,                // Closure: writes captured text given an index
    mut name_to_index: N,         // Closure: resolves named groups to indices
    dst: &mut Vec<u8>,            // Output buffer, caller-controlled
) where
    A: FnMut(usize, &mut Vec<u8>),        // May mutate state between calls
    N: FnMut(&str) -> Option<usize>,      // Returns None for unknown names
{
    // ... implementation
}
```

The `FnMut` bounds allow closures that capture and modify external state. The `Option<usize>` return type for `name_to_index` handles the case where a named group doesn't exist—the caller decides whether that's an error or should be silently ignored.

Section 2: Efficient Scanning with memchr

```
while !replacement.is_empty() {
    match memchr(b'$', replacement) {
        None => break,  // No more $ characters, exit loop
        Some(i) => {
            // Copy literal text before the $ to output
            dst.extend(&replacement[..i]);
            // Narrow slice to start at the $
            replacement = &replacement[i..];
        }
    }

    // ... handle the $ reference ...
}
// After loop: append any remaining literal text
dst.extend(replacement);
```

The `memchr` crate provides SIMD-accelerated byte searching on supported platforms. By narrowing `replacement` rather than tracking an index, each iteration always starts at a `$` character, simplifying the parsing logic.

Section 3: Escape Sequence Handling

```
// Check if this is an escaped dollar sign ($$)
if replacement.get(1).map_or(false, |&b| b == b'$') {
    dst.push(b'$');           // Output single literal $
    replacement = &replacement[2..];  // Skip both $ characters
    continue;                 // Restart loop to find next $
}
```

The `get(1)` returns `Option<&u8>`, safely handling the case where `$` is at the end of the string. The `map_or(false, ...)` pattern provides a default when the index is out of bounds, avoiding a separate length check.

Section 4: The CaptureRef and Ref Types

```rust
/// Represents a parsed capture reference with its position.
/// The lifetime 'a ties this to the input slice for zero-copy parsing.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct CaptureRef<'a> {
    cap: Ref<'a>,     // The actual reference (number or name)
    end: usize,       // Byte position immediately after the reference
}

/// A reference to a capture group - either by index or by name.
/// e.g., `$2`, `$foo`, `${foo}`.
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
enum Ref<'a> {
    Named(&'a str),   // Borrows from input - no allocation
    Number(usize),    // Self-contained, no borrowing needed
}
```

The `end` field enables the caller to advance past the reference correctly—`${foo}` consumes 6 bytes while `$foo` consumes 4. The lifetime on `Ref::Named` ensures the borrowed string slice remains valid.

Section 5: The From Trait Implementations

```rust
impl<'a> From<&'a str> for Ref<'a> {
    #[inline]
    fn from(x: &'a str) -> Ref<'a> {
        Ref::Named(x)  // Lifetime flows from input to output
    }
}

impl From<usize> for Ref<'static> {
    #[inline]
    fn from(x: usize) -> Ref<'static> {
        Ref::Number(x)  // 'static because numbers don't borrow
    }
}

// Test macro exploiting the From implementations
macro_rules! c {
    ($name_or_number:expr, $pos:expr) => {
        CaptureRef { cap: $name_or_number.into(), end: $pos }
    };
}

// Usage in tests:
// c!("foo", 4)  -> CaptureRef with Named("foo")
// c!(5, 2)      -> CaptureRef with Number(5)
```

The difference in lifetimes (`'a` vs `'static`) reflects a real semantic distinction: named references borrow from their source, while numeric references are independent values.

Section 6: Parsing Capture References

```
#[inline]
fn find_cap_ref(replacement: &[u8]) -> Option<CaptureRef<'_>> {
    let mut i = 0;
    // Must have at least $ and one more character
    if replacement.len() <= 1 || replacement[0] != b'$' {
        return None;
    }

    let mut brace = false;
    i += 1;  // Skip past $

    // Check for braced form: ${...}
    if replacement[i] == b'{' {
        brace = true;
        i += 1;
    }

    // Scan valid identifier characters
    let mut cap_end = i;
    while replacement.get(cap_end).map_or(false, is_valid_cap_letter) {
        cap_end += 1;
    }

    // Must have at least one valid character
    if cap_end == i {
        return None;
    }

    // Safe because is_valid_cap_letter only accepts ASCII
    let cap = std::str::from_utf8(&replacement[i..cap_end])
        .expect("valid UTF-8 capture name");

    // Braced form requires closing brace
    if brace {
        if !replacement.get(cap_end).map_or(false, |&b| b == b'}') {
            return None;
        }
        cap_end += 1;
    }

    // Determine if it's a number or a name
    Some(CaptureRef {
        cap: match cap.parse::<u32>() {
            Ok(i) => Ref::Number(i as usize),
            Err(_) => Ref::Named(cap),
        },
        end: cap_end,
    })
}
```

The function returns `None` for malformed references rather than erroring, allowing the caller to treat unrecognized patterns as literal text.

Section 7: Number vs Name Disambiguation

```
/// Returns true if and only if the given byte is allowed in a capture name.
#[inline]
fn is_valid_cap_letter(b: &u8) -> bool {
    match *b {
        b'0'..=b'9' | b'a'..=b'z' | b'A'..=b'Z' | b'_' => true,
        _ => false,
    }
}

// In find_cap_ref, after extracting the identifier:
Some(CaptureRef {
    cap: match cap.parse::<u32>() {
        Ok(i) => Ref::Number(i as usize),  // "123" -> Number(123)
        Err(_) => Ref::Named(cap),         // "foo" or "42a" -> Named
    },
    end: cap_end,
})
```

The `parse::<u32>()` attempt determines the reference type. Note that `$42a` becomes `Named("42a")` because "42a" fails to parse as a number. Using `u32` before casting to `usize` prevents accepting negative numbers while keeping the index size platform-appropriate.

Quick Reference

Supported Reference Syntax

Syntax	Example	Meaning
`$N`	`$1`	Numeric reference to group N
`$name`	`$foo`	Named reference (greedy scan)
`${N}`	`${1}`	Braced numeric (for disambiguation)

`${name}`	`${foo}`	Braced named (for disambiguation)
`$$`	`$$`	Literal `$` character

Key Types

```
struct CaptureRef<'a> { cap: Ref<'a>, end: usize }
enum Ref<'a> { Named(&'a str), Number(usize) }
```

Function Signatures

```
pub fn interpolate<A, N>(replacement: &[u8], append: A, name_to_index: N, dst:
&mut Vec<u8>)
where
    A: FnMut(usize, &mut Vec<u8>),
    N: FnMut(&str) -> Option<usize>;

fn find_cap_ref(replacement: &[u8]) -> Option<CaptureRef<'_>>;
fn is_valid_cap_letter(b: &u8) -> bool;
```

Valid Identifier Characters

- Digits: `0-9`
- Lowercase: `a-z`
- Uppercase: `A-Z`
- Underscore: `_`