ChatGPT

Parsing Failures in Syn Procedural Macros: Root Causes & Best Practices

1. Path Parsing and Trailing Semicolons in `syn`

How `ParseStream::parse::<ExprPath>()` advances the stream: When you call `input.parse::<ExprPath>()` or `input.parse::<Path>()`, `syn` will parse a path expression up to its natural end (identifiers and `::` separators). Crucially, it will **stop before any punctuation** that isn't part of the path grammar, such as a semicolon. The semicolon (`;`) is *not* consumed by the `ExprPath` parser – it remains in the stream to be handled afterward [1]. In other words, `ExprPath` parses the path tokens (e.g. `SomeState` or `self::foo`) and leaves the `;` for you to parse next (usually via `input.parse::<Token![;]>()?`). If you forget to consume the semicolon, the parser will end with leftover tokens, causing an error at the macro call site (since syn expects the input to be fully consumed) [1]. A forum commenter summarizes this: *"syn is cursor-aware in parsing: when tokens are fully parsed, the cursor should reach the end"* [1]. This often manifests as an *"unexpected token"* error if trailing tokens (like an extra `;` or other input) remain unparsed.

Edge cases – semicolon immediately after a path: Normally, a semicolon right after a path is fine (e.g. `initial: SomeState;`). The `Path` parse will succeed on `SomeState` and stop, and then you parse the `;` separately. There isn't a special hazard here as long as you handle the semicolon in your grammar. However, if the semicolon appears *with no preceding path tokens* (e.g. an empty entry like `entry: ;`), then `ExprPath` will error because it finds `;` where an identifier or path is expected. In such a case you'd get an error like "expected identifier or path, found `;`" since the parser sees a semicolon at a position where a path should start.

Edge cases – paths starting with `.` or `self.`: A leading dot is **not a valid start for an** `ExprPath` in Rust's syntax. If the next token is `.` (dot), `input.parse::<ExprPath>()` will immediately fail, usually with an error akin to "expected identifier" or "expected expression, found `.`". This is because a standalone `.foo` has no base expression – Rust only allows leading `.` in specific contexts (like continuing a method chain after a macro or closure). For example, syn historically would error on something like `.len()` after a macro call, treating the `.` as an unexpected token [2]. In a DSL, an entry like `entry: .some_hook;` will likely trigger an error at the `.` token ("expected identifier, found `.`"). Similarly, `self.cleanup` **is** a valid Rust expression, but `syn` will parse it not as a simple `ExprPath` but as a field access expression (`ExprField` in the AST). The `ExprPath` parser might stop at `self` and not consume the `.cleanup`. If you try to parse `self.cleanup` with `parse::<Path>()`, it will error out when encountering the `.` because the `Path` parser doesn't know how to handle the dot punctuation (it expects `::` for path separation, not `.`). In short, `ExprPath` / `Path` **only handle pure paths** (like `Ident`, `module::Ident`, or keywords like `self` / `crate` as segments). A dot denotes a property or method access, which falls outside `ExprPath`. So constructs like `self.foo` or `.foo` require special handling (either parsing as a full `Expr` or manual parsing logic).

Takeaway: To avoid parse errors, design your parser to consume the semicolon explicitly and handle non-standard path notations: - After parsing a path or expression, always consume the expected `;`. For example: `let target: Path = input.parse()?; input.parse::<Token![;]>()?;` [3]. This ensures no stray semicolon remains [3]. - If your DSL allows a leading dot (shorthand for `self`), you may need to handle it manually. For instance, peek for `Token![.]` and insert an implicit `self` segment. Alternatively, parse it as a general `Expr`: you could `input.parse::<Expr>()` to let syn handle `self.foo` as an `ExprField` expression. Just be aware that a lone leading `.` still isn't valid Rust, so you'd need to decide what it means and perhaps prepend a `self` in the generated code or parsing logic.

2. Nested `braced!` Content and Token Visibility

Using `syn::braced!(content in input)` to parse a block of tokens inside `{ ... }` is a common technique for DSLs. This macro **parses the curly braces and yields a new** `ParseBuffer` (here bound as `content`) for the interior [4]. All tokens between the `{` and matching `}` become part of the `content` buffer. From that point: - You parse the inner DSL items from `content` just like you would from any other `ParseStream`. Tokens inside are "invisible" to the outer stream until you exit the braced content. The outer `input` sees the `{ ... }` as one token (the braces), and after `braced!` it resumes at the token immediately after the closing brace. - Token visibility is naturally limited: you **cannot peek or parse tokens outside the braces from within the** `content` stream. Conversely, once you finish parsing `content` (or if you look ahead to the end of content), the next thing in `content` is essentially the `}` (end of that buffer).

In practice, parsing from a nested stream does not intrinsically change how path parsing works – an `ExprPath` inside `content` behaves the same as one in the outer stream. The main "gotcha" is to ensure you consume everything in the braced block that you expect, and then exit at the right time. If you leave unparsed tokens in `content` or consume the `}` incorrectly, you'll get errors (e.g., "unexpected token `}`" or similar). For example, if your DSL state block is defined as `{ entry: ...; exit: ...; }`, you would do something like:

```
let content;
brace_token = braced!(content in input);
 // now parse inside braces:
let maybe_entry = if content.peek(...)"{ ... } // parse entry field
...
content.parse::<Token![}]>()?; // not needed explicitly, braced! handles closing
brace.
```

Typically, you don't manually parse the `}` token – the `braced!` macro already consumed it and gave you a `token::Brace` handle. You just ensure all the inner tokens are consumed **before** content is empty or before hitting the closing brace.

Lookahead and unexpected consumption in nested context: One subtle point is managing where the inner parsing should stop. Often you might have multiple sections in one block. For instance, consider a state DSL where inside `{ ... }` you have optional `entry`, `exit` hooks, and then a `transitions { ... }` sub-block. You might parse until you detect the start of the transitions sub-block. Here,

lookahead (`ParseStream::peek`) is your friend. You can peek for a specific keyword or brace to decide when to stop parsing one thing and start another. For example, in one approach to parsing a trait with fields and methods, a developer used:

- One braced content for fields, then when a `fn` token is next, they knew the fields section ended [5]. They wrote a loop like:

```
while !content.peek(Token![fn]) {
    fields.push(content.call(Field::parse_named)?);
    let _: Token![;] = content.parse()?;
}
// Once we see `fn`, we break out and handle the methods next.
```

This ensures that if the next token is `fn` (not a field), the loop ends *before* trying to parse a semicolon after the last field [6]. Without this lookahead, a naive parser might attempt to parse another field or a semicolon and throw an error like *"expected `;`"* or *"expected identifier"* upon seeing `fn`. By using a nested parse buffer and careful peeking, they avoid consuming tokens that belong to the next section.

- Another braced content can then be parsed for the transitions themselves (perhaps via another `braced!(content2 in content)` if transitions are in their own braces).

In summary, **parsing inside braces does not fundamentally alter token behavior**, but you need to manage boundaries: - Use separate `ParseBuffer` instances for separate blocks (e.g., one for state body, another for a sub-block of transitions) [7]. - Use lookahead on the inner buffer to detect when to stop parsing a list before an upcoming section [6]. - Ensure that after finishing with `content`, you've consumed or appropriately handled all expected tokens inside it. The outer parser will continue after the `}`. If, for example, a `}` is immediately followed by a semicolon in your syntax, you'd need to parse that semicolon *after* the `braced!` call (from the outer `input`).

There are no hidden "visibility" issues – think of `content` as just a slice of the original token stream. The main gotcha is matching your grammar: forgetting to parse a token inside the braces or not handling the transition from inner to outer parsing will lead to errors (like *"unexpected token"* at the brace or at whatever is unparsed).

3. Common Parsing Pitfalls and Known Issues (`.some_hook`, `self.cleanup`, etc.)

When parsing DSL constructs, certain patterns can trigger familiar errors. Some examples and known issues:

- **Leading dot (`.some_hook`) causing *"expected identifier"* errors:** As discussed, a leading `.` is not a valid Rust expression on its own. If your macro input has `entry: .some_hook;`, syn will throw an error at the `.` token. This often appears as an error about an expected identifier or expression. In Rust's parser (and syn), a dot is only valid *after* an expression, never at the start. There isn't a syn bug here per se; it's a language grammar constraint. The fix is to interpret `.some_hook` in your

DSL (perhaps as an implicit `self.some_hook`). You would need to adjust your parser to handle this (e.g., consume the `Token![.]` and treat the following ident as a method name). Otherwise, syn will keep complaining about the unexpected `.`. (A similar situation was encountered in Rust 1.56 with macro invocations: a dot right after a macro was once rejected until the grammar was updated [2], showing how such cases can confuse the parser.)

- **Parsing `self.cleanup` as a path expression:** If you use `input.parse::<ExprPath>()` on `self.cleanup`, you might get an error like *"unexpected token `.`"* or *"expected `::` or `<`"*. That's because `ExprPath` stops at `self` (which it treats as a complete path by itself). The `.cleanup` part is left unparsed or triggers an error. In syn's AST, `self.cleanup` would actually be an `ExprField` (field access) node, not an `ExprPath`. If your DSL expects something like `self.identifier;`, you should consider parsing it as a general `Expr` or specifically handle `self.` as a prefix. One approach is to allow `Expr` parsing for hooks, which would naturally parse `self.cleanup` into an AST (you can later ensure it's the form you want). If you strictly want a path (no `()` call, no complex expression), you might parse an `Ident` or `Path` and allow an optional `.ident` after it. But syn doesn't provide a one-liner for "path or field access" – you'd implement that logic. For instance:

```
// Pseudocode for custom parse of something like [self.]ident;
let target_path: Expr = input.parse()?;  // parse any expression
// then check that target_path is either an ExprPath or ExprField with base
self.
```

This way `self.cleanup` or just `SomeFunc` would parse, but something unintended (like `foo + bar`) would be caught in validation.

- **"expected `;`" errors in nested DSLs:** These typically occur if you forget to consume a semicolon or if the input is missing one. For example, if your DSL syntax requires a semicolon after each hook or transition, but the user or parser omitted it, syn will error at the next token (which could be `}` or another keyword) complaining it wanted a semicolon. For instance, if one wrote `exit: self.cleanup` (no semicolon) before the closing brace, syn would likely error at the `}` with *"expected `;`"*. The solution is ensuring the grammar and parser align: use `Token![;]` in your `Parse` impl after each item. Syn's `parse_terminated` can help by automatically expecting separators. If using `Punctuated` (see below), it will produce a nice error like *"expected `;`"* if a separator is missing [5]. If not using `Punctuated`, you can manually generate an error via `input.error("expected ;")` when appropriate.

- **Keywords or reserved tokens where an ident is expected:** Another known pitfall is if a keyword appears where syn expects an ident in a path. For example, an issue was noted where `type = "hi"` in an attribute caused *"expected identifier, found keyword `type`"* [8]. In your DSL, if state names or hook names coincide with Rust keywords, syn's `Path` parser will not accept them. (For instance, a state named `Self` or `crate` might need special handling.) Be mindful that `syn::Ident` will reject Rust reserved words by default. You might need to use `Ident::parse_any` or handle keywords explicitly if your DSL wants to allow them as names.

- **Trailing tokens and parse completion:** A very common scenario (illustrated by the forum discussion on `TraitVarType`) is forgetting that a parse should not consume unrelated trailing tokens. The user in that thread tried to parse `Vec<T,...>; x` entirely as one type, and got an error because of the stray `x` after the semicolon [9]. In a state machine DSL, this could happen if, say, you try to parse too much at once or leave extra input unhandled. The general rule is: each `Parse` implementation should consume exactly the tokens for that syntactic element. Any extra tokens will cause an error. The fix is to either consume the extra tokens (if they are part of the syntax) or not supply them to that parse call (parse them in a larger context). In practice, ensure that after parsing a block or construct, you're at the appropriate end (brace, semicolon, etc.). If not, adjust your parser to handle whatever is left or report a clear error.

4. Patterns from Other Procedural Macros (smlang, statig, bevy, etc.)

To robustly parse paths or expressions followed by semicolons, many macro authors follow a few idioms:

- **Use `Punctuated` for lists of items with separators:** The `syn::punctuated::Punctuated` type can parse sequences of things separated by a token. For example, if you have multiple transitions or declarations separated by `;`, you can do:
 `let items: Punctuated<TransitionDefinitionAst, Token![;]> = content.parse_terminated(TransitionDefinitionAst::parse)?;`.
 This will loop internally, parsing `TransitionDefinitionAst` repeatedly, expecting a semicolon after each. It even allows an optional trailing semicolon. Using `parse_terminated` simplifies handling of separators and provides good error messages (it will complain if a semicolon is missing between items). In one of the forum examples, they parse a list of struct fields with `Punctuated<Field, Token![;]>` inside braces [7]. This approach is clean when your DSL section is a homogeneous list of things.

- **Manual parsing with lookahead for mixed content:** If the content inside braces isn't a simple homogeneous list (e.g., maybe a state block can contain an `entry` hook, an `exit` hook, a `default_child`, and then a transitions block), you can't directly use `Punctuated` because the items are different. In such cases, macros often use a **hand-rolled parser**: check for expected keywords or tokens and parse accordingly. For example:

```
let mut entry_hook = None;
let mut exit_hook = None;
let mut default_child = None;
let mut transitions = None;
while !content.is_empty() {
    if content.peek(keyword::entry) {
        content.parse::<keyword::entry>()?;
        content.parse::<Token![:]>()?;
        let expr: Expr = content.parse()?;  // parse hook expression
        content.parse::<Token![;]>()?;
        entry_hook = Some(expr);
```

```
    } else if content.peek(keyword::exit) { ... }
    else if content.peek(keyword::default_child) { ... }
    else if content.peek(keyword::transitions) {
        // parse the transitions sub-brace
        content.parse::<keyword::transitions>()?;
        content.parse::<Token![:]>()?;
        let inner;
        content.braced(&mut inner)?;
        // parse transitions inside `inner`, possibly with
Punctuated<Transition, Token![;]>
        transitions = Some(inner.parse_terminated(Transition::parse)?);
    } else {
        return Err(content.error("unexpected token in state definition"));
    }
}
```

This is a sketch, but the idea is to use `peek` to decide which field you're looking at. Each section consumes its tokens and semicolon. This way, you handle each of `entry`, `exit`, `default_child`, etc., in turn. Many real-world proc macros use this pattern for DSLs. For instance, the `smlang` crate's macro parses state transitions by looking for patterns like `StateA + Event / Action = StateB` separated by commas [10]. In doing so, it likely uses a combination of `Punctuated` (for multiple transitions separated by commas) and custom parsing for the pieces around the `+` and `=`. The key is breaking the problem down:

- Use `peek` or keyword detection to choose the parse branch.
- After parsing a piece (like an expression or path), **immediately parse the terminating semicolon** if one is expected.
- Continue until the closing brace or known terminator token.

- **Workarounds for tricky grammar parts:** Some macros use *custom combinators or split parsers* if the grammar isn't easily handled with one pass. For example, the `synstructure` crate (for deriving) doesn't parse new syntax, but it demonstrates robust handling of Rust syntax by deferring to syn's own parsing for complex parts (like expressions in attributes). In your case, you might choose to parse any user-provided hook as a full `Expr` to leverage syn's robustness, then later ensure it's the kind of expression you want (e.g., a simple path or method call). This is a form of graceful handling: accept a broad syntax then validate. Another trick: use `ParseBuffer::call()` to invoke an existing parser on a substream. In the earlier forum code, they did `content.call(Field::parse_named)` to parse a struct field using syn's built-in field parser [11]. You could similarly call `ExprPath::parse` or custom sub-parsers on parts of your DSL.

- **Examples from existing crates:**

- *Bevy ECS macros:* Many Bevy macros use `Punctuated` for lists (e.g., in `bevy_reflect` derive, to parse list of field attributes, they use punctuated lists of meta-items). They also often manually parse keyword-value pairs.

- `statig` *crate:* This crate uses an attribute macro on `impl` blocks rather than a custom DSL syntax, so it might not have an entry like `entry: hook;`. Instead, it uses attributes like `#[state(entry_action = "enter_on")]`. That means they let Rust's attribute parser give them a `Meta` list which they then interpret. However, the principle is similar – parse key-value pairs and possibly paths inside (with syn's help).
- *Other state machine crates:* If any allow code blocks or expressions for actions, they typically parse those as `Expr` or `Block` outright. For instance, a transition action might be a `{ ... }` block or a function call; parsing it as `Expr` covers both possibilities.

In summary, **consistent idioms include**: - Using `Punctuated<T, Sep>` for repeated constructs with separators [7]. - Using `peek` and manual parsing loops to handle optional or ordered sections, stopping at the right time to avoid consuming the next section's token [6]. - Parsing with broader types (like `Expr`) and then narrowing, to gracefully handle what the user wrote without needing to write a complex parser for expressions yourself. - Defining custom keywords via `syn::custom_keyword!` for things like `entry`, `exit`, then using `content.parse::<keyword::entry>()?` to consume them, which gives clearer error messages on failure than checking strings.

These patterns help make your parser both robust to user error (providing good errors like "expected `;` or `transitions`") and flexible in what it accepts.

5. Debugging Procedural Macros with `syn`

Parsing with syn can feel like a black box, but there are several techniques to debug and understand parse failures:

- **Print the remaining tokens during parsing:** The `ParseStream` (`input`) can be printed (it implements `Display` to show the upcoming tokens). You can insert `println!` or `dbg!` calls at various points in your `Parse` implementation to see what tokens are left. For example, in the forum snippet below, they print the state of `input` after parsing each piece [12]:

  ```
  println!("orig input: {}", input);
  let vis: Visibility = input.parse()?;
  println!("after vis: {}", input);
  let name: Ident = input.parse()?;
  println!("after name: {}", input);
  // ...
  ```

  This will show something like "orig input: entry : .some_hook ; exit : self . cleanup ; }" and then step by step. It's incredibly useful to pinpoint where the parser stops or fails. In fact, the snippet shows how after parsing a type, the next token was `;` and then after consuming `;`, the next token was an unexpected ident `x` [9]. Seeing that in debug output made it clear why syn error'd (there was an extra token).

- **Print tokens one by one:** If the structure is complicated, you can literally consume tokens in a loop and print them. In one example, a developer wasn't sure how syn was tokenizing a type, so they did:

```
while !input.is_empty() {
    if input.peek(Token![;]) {
        println!("EXIT token: `{}`", input.parse::<TokenTree>()?);
        break;
    }
    let token: TokenTree = input.parse()?;
    println!("token: {}", token);
}
```

[13] This prints each token (using `proc_macro2::TokenTree` which can represent any token or group) until a semicolon. Such debug loops help reveal the exact sequence of tokens syn sees. For instance, you might discover that `.some_hook` is two tokens: `.` and `Ident(some_hook)`. Or that `self.cleanup` is actually three tokens: `Ident(self)`, `Punct(.)`, `Ident(cleanup)`.

- **Use `lookahead1()` for precise error reporting:** While not a debug technique per se, using `ParseStream::lookahead1()` can improve your error messages. For example, if none of your expected branches match, you can do:

```
let la = content.lookahead1();
return Err(la.error());
```

This will produce an error like "expected `entry` or `exit` or `transitions`" based on the tokens you peeked for. It helps both you and the user understand what was expected when parse fails.

- **Utilize `ParseBuffer::fork()` to test alternatives:** If you are attempting to parse one of several possibilities (and want to avoid consuming tokens on the wrong guess), use `input.fork()`. A forked parse buffer lets you try parsing without affecting the original cursor [14]. For example, you might fork the input, attempt `forked.parse::<ExprPath>()`, and if it errors, you know that approach failed without consuming the original stream. This is advanced debugging (and parsing) technique useful for backtracking or checking *"would this parse succeed?"*. It requires some manual work to handle errors from the fork, but it can be a lifesaver for tricky grammars. You can also use `ParseBuffer::advance_to()` to commit the original input to where the fork got to if an alternative succeeds [14].

- **Leverage span info in errors:** When syn returns an error (a `syn::Error`), it includes a span pointing to the offending token. In a complex macro input, this span can localize the issue. You can catch `syn::Result` in your parser and do something like `eprintln!("Parse error at {:?}", error.span())` to see roughly where it happened. (Or simply allow the compiler to report it – it will show the span in the user's code.)

- **Write small test cases in isolation:** Much like writing unit tests, it helps to isolate parts of your parser. You can use `syn::parse_str::<Type>("some tokens")` in a `#[test]` to see if a particular snippet parses as you expect. The forum example we've been citing had a `#[test]` that directly parsed a snippet and printed debug info [15]. You can do the same for your

`TransitionDefinitionAst` or others – feed in a minimal example (e.g., just an `entry` line) and see how it behaves.

- **Consult similar implementations:** Sometimes the quickest way to debug is to see how others did it. Look at open-source macros (like those in `bevy` or `smlang`) for patterns. If they parse something similar, their code can shed light on the intended approach.

By applying these tips, you can often pinpoint why a parse is failing: - Maybe a token isn't being consumed when you thought it was. - Maybe an `if/else` branch in your parser isn't matching as expected. - Or perhaps your `peek` logic is off by one token.

For instance, using the above techniques you might discover that after parsing an `ExprPath`, the semicolon is still in `input` (meaning you forgot to parse it). Or that `.some_hook` never even enters your `if content.peek(Ident)` branch because the first token is a punct (`.`), not an ident.

Finally, when you fix your parser, consider adding **better error messages** for your users. You can manually check for a leading dot and produce a custom error like *"entry hook cannot start with* `.` *(did you mean* `self.`*?)"* to guide users. Syn allows you to create errors with spans (e.g., `return Err(Error::new(span, "helpful message"));`). This can make your macro much more user-friendly and easier to debug in the future.

Conclusion: Applying these insights to `lit-bit-macro`

To address the parsing bugs in your `TransitionDefinitionAst`, `LifecycleHookAst`, and `DefaultChildDeclarationAst`:

- **Handle the leading-dot hook syntax:** Modify `LifecycleHookAst::parse` to accept an optional leading `.`. For example, if `content.peek(Token![.])`, consume it and record that you should treat the next ident as a method on `self`. You might internally store the hook as an `Expr` or a special enum variant (e.g., `Hook::Method(Ident)`) indicating an implicit self call. This will prevent the "expected identifier" error for `.some_hook;`. If you want `.foo` and `self.foo` to mean the same, you can normalize both to a form of `Expr` that represents `self.foo`.

- **Parse `self.cleanup` properly:** Instead of using `parse::<ExprPath>()` for hooks, consider using `parse::<Expr>()` and then ensuring it's an allowed form. This way `self.cleanup` will be parsed as an expression (likely an `ExprField`), and you can accept it. If you need just an identifier (like for a function name), you could also parse an `Ident` or a `Path` after handling `self.` explicitly. The key is to not treat `self.cleanup` as a bare `Path` — handle the dot.

- **Ensure all semicolons are consumed:** Audit each section of your grammar – after parsing an entry/exit/default declaration, are you calling `input.parse::<Token![;]>()`? If not, add it. If using `Punctuated`, that might already handle it, but be cautious with optional sections. For example, if `DefaultChildDeclarationAst` parses something like `default_child: StateName;`, make sure it always eats the semicolon. If it's optional (maybe the whole clause is optional), you might parse it with something like:

```
let default_child = if content.peek(keyword::default_child) {
    content.parse::<keyword::default_child>()?;
    content.parse::<Token![:]>()?;
    let path: Path = content.parse()?;
    content.parse::<Token![;]>()?;
    Some(path)
} else {
    None
};
```

This way, if the `default_child` appears, it will consume its semicolon, and if not, it won't accidentally consume something else.

- **Use consistent structure for state blocks:** If your state blocks contain multiple kinds of statements, consider the manual parsing loop with `peek` as described. This will make the parser more robust to ordering and missing tokens. It also simplifies error handling since you can pinpoint which clause was expected next.

- **Test incrementally:** Write tests for snippets like:

- Only an entry hook: e.g. `StateX { entry: self.foo; }`
- Only an exit hook with leading dot: `StateX { exit: .cleanup; }`
- State with both hooks and default_child: ensure all semicolons are needed and parsed.
- A transition block present vs. absent. Each test can use `syn::parse_str` on a small TokenStream and assert that parsing succeeds (or fails with a specific error). This will validate that your fixes work.

Using the above strategies, you'll avoid the common parsing pitfalls and make your macro's parser much more reliable. The result will be clearer compile errors for your users and less frustration when extending the DSL. Good luck with `lit-bit-macro`! [13] [16]

---

[1] [3] [5] [6] [7] [9] [11] [12] [13] [15] [16] Issue with crate Syn and `Parse` trait in functional macro - help - The Rust Programming Language Forum
https://users.rust-lang.org/t/issue-with-crate-syn-and-parse-trait-in-functional-macro/109367

[2] Parse dot and question mark after curly braced macro invocation · Issue #1068 · dtolnay/syn · GitHub
https://github.com/dtolnay/syn/issues/1068

[4] braced in syn - Rust
https://docs.diesel.rs/2.1.x/syn/macro.braced.html

[8] Meta does not parse paths with leading double-colon · Issue #909 ...
https://github.com/dtolnay/syn/issues/909

[10] smlang - Rust
https://docs.rs/smlang

[14] Expr parsing from `syn` crate - help - The Rust Programming Language Forum
https://users.rust-lang.org/t/expr-parsing-from-syn-crate/117507