Turning Parser Errors into Suggestions for REPL-Driven DSLs (Functional Pearl)

Turning Parser Errors into Suggestions for REPL-Driven DSLs (Functional Pearl)

MATTHÍAS PÁLL GISSURARSON, DPella AB, Sweden
ELISABET LOBO-VESGA, DPella AB, Sweden
ALEJANDRO RUSSO, Chalmers University of Technology, Sweden, Gothenburg University, Sweden, and DPella AB, Sweden

Many of the domain-specific languages we use every day are not written to files but typed at an interactive prompt, e.g., database shells, cloud CLIs, and in-house analytics consoles. For these *REPL-driven* command languages, autocomplete is, we argue, not a polish feature but a core usability requirement—and yet they are precisely the languages whose authors often have the fewest resources to invest in tooling. Designing a new programming language tailored to specific domain challenges can be both powerful and rewarding. However, a major hurdle for adoption among users is, in our experience, the lack of tooling support, particularly features like autocomplete that enhance usability and reduce the learning curve. In this paper, we present a fresh perspective that turns parser errors—traditionally seen as roadblocks—into opportunities for generating valid, context-aware autocomplete suggestions. By leveraging the parser's built-in feedback mechanisms, our method offers a lightweight, adaptable, and simple solution, enabling domain-specific languages to provide basic development assistance with minimal overhead. The technique fits the REPL-driven DSL genre by design: inputs are modest-size single statements, the cursor sits at the end of the line, and parser-error recovery is unnecessary. We apply the technique to DPella DSL, a production REPL-driven DSL whose grammar comprises 43 top-level command constructors, 130 reserved keywords, and 22 labelled syntactic categories that drive identifier completion.

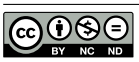
CCS Concepts: • **Theory of computation → Grammars and context-free languages**; • **Software and its engineering → Functional languages**.

Additional Key Words and Phrases: Autocomplete, Parsing, Suggestions, Domain-Specific Languages, REPL

ACM Reference Format:
Matthías Páll Gissurarson, Elisabet Lobo-Vesga, and Alejandro Russo. 2026. Turning Parser Errors into Suggestions for REPL-Driven DSLs (Functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Haskell Symposium (Haskell '26), August 24–29, 2026, Indianapolis, IN, USA*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3830439.3831275

1 Introduction

Every day, developers often reach for a long list of tools whose interface is a domain-specific language (DSL) typed at a prompt and operated as a read-eval-print loop (REPL): database shells such as `sqlite3`, devops CLIs such as `kubectl`, and a long tail of in-house command-line interfaces, each with its own bag of domain-specific commands. In our experience, few of these tools provide autocomplete—even though completion is the single feature that would make those commands discoverable to the users who depend on them. The reason is structural: their authors rarely have the engineering budget or time to build it. These command languages share a profile—short single-statement inputs, an active cursor at the end of the line, no need for parser-error recovery, and a user who is often a domain expert rather than a programmer. We will refer to these class of



Haskell '26, Indianapolis, IN, USA

ACM ISBN 979-8-4007-2869-3/2026/08
https://doi.org/10.1145/3830439.3831275

languages as *REPL-driven DSLs*. We will show that for languages of this genre an autocomplete good enough to lower the learning curve can be *derived essentially for free* from the parser the language already needs. We do not aim at a full-featured autocomplete; rather, we trade sophistication for the ability to bootstrap useful completions for minimal effort.

In this paper, we present a fresh perspective on supporting autocomplete by a simple idea: *when parsing fails due to incomplete or incorrect input, the parser typically generates error messages indicating what it expected at the point of failure, and by extracting such information, the autocompleter can make suggestions.* In this light, parsing error are no longer treated as roadblocks, but rather as guidelines for the user to navigate and discover the language. We demonstrate our technique within a REPL, and show how our technique can even be used for providing context-aware suggestions that enhance usability and lower the learning curve for new users.

The idea of suggesting tokens to repair user inputs is not new. For instance, there are techniques which require parsers to make decisions on behalf of the user—e.g., choosing a specific branch in a choice parser combinator [18], or offering an *undo* mechanism if the repair was unsatisfactory [6]. Furthermore, frameworks like Langium [11] and Xtext [5] leverage the Language Server Protocol (LSP) to provide language-agnostic autocomplete and other features for DSLs. Using Langium or Xtext for a given DSL would require replicating the parser's grammar, AST structure, and token definitions within the LSP configuration, which can be redundant and error-prone. In contrast, our technique (i) does not attempt to repair the input, but rather to surface acceptable alternatives—keeping the user in control of which suggestion to follow; and (ii) relies on out-of-the-box parser behavior with no need to replicate the DSL's grammar.

We do not claim that our technique scales to arbitrary domain-specific languages. Several design choices are deliberately tuned to the REPL-driven DSL setting: (a) we drive suggestions from the parser's *first* error, which is fine when each input is a single statement but fragile in a hundred-line file; (b) we expect the cursor at the end of the input, which matches a prompt but not the middle-of-file cursor of an editor; (c) we forgo parser-error recovery, because there is nothing past the cursor to recover for in a single-line REPL turn; and (d) we treat "context" as the running session's state, not a project-wide semantic model. Read in the right setting, these are not limitations but a precise characterisation of the genre the technique is built for.

The techniques presented in this paper are generalizable to parser libraries which support (i) modularity via parser combinators, (ii) have informative error messages with expected tokens, (iii) allows for enriching parser errors with arbitrary data (i.e., supports custom errors), and (iv) allows attaching labels to parsers. We illustrate our technique using the Haskell Megaparsec parser library—we note that performance-focused parsers (e.g. attoparsec[1]) lack the required bookkeeping and are unsuitable for our technique. The features above are not exotic: on Hackage alone, over 200 libraries depend on Megaparsec[2], which natively supports all four features. Around 900 further libraries depend on Parsec[3]; they cover (i), (ii), and (iv), and the custom-error requirement (iii) can be met by a workaround (see Section 7). Attoparsec, as noted above, lacks (ii) and is not a suitable host.

2 Motivating example

To illustrate our approach, we use a simplified version of DPella DSL, a production DSL for writing *privacy-preserving SQL-like queries* [12, 13] . This language is a key component of the DPella DSL product[4], where users are provided with an interactive command-line REPL, similar to using

[1] https://github.com/haskell/attoparsec
[2] As of August 22, 2026; count obtained from https://hackage.haskell.org/package/megaparsec/reverse.
[3] As of August 22, 2026; count obtained from https://hackage.haskell.org/package/parsec/reverse.
[4] https://dpella.io/

```
1  // Creating accounts
2  create account mithrandir with name "Gandalf";
3
4  create account pippin with name "Peregrin Took";
5
6  // Show existing accounts
7  show accounts;
8
9  // Assign permissions
10 grant role curator to mithrandir;
11
12 // Show information
13 show account mithrandir;
```

Fig. 1. Commands for a DPella DSL session

`sqlite3` [9]—DPella DSL matches the profile of other REPL-driven DSLs we have used, rather than being an outlier. Once in a running session, users have access to a set of commands. While the full details of DPella DSL are beyond the scope of this paper, we focus on a *subset of the language* that deals with account management. This subset is sufficient to illustrate our approach to *autocomplete*.

Figure 1 illustrates the sequence of commands executed during a typical DPella DSL session. In this snippet, we can see the key language constructs: the `create` command is used to create new accounts (see line 2), the `grant` command is used to assign account roles (line 10), and the `show` command is used to display information about existing accounts (see lines 7 and 13). While the exact details of each command are not essential for the purpose of this paper, it is worth noting that the language involves *various constructs that may be challenging for newcomers*.

Because DPella DSL is a DSL, users may not be familiar with its exact syntax or keywords. To enhance usability, we aim to provide *autocomplete suggestions* as they type in the REPL. For example, suppose the user writes `cre`□ (where □ represents the cursor position) and presses <tab>. Ideally, the system should expand it to `create`, since that is the only keyword matching the input:

```
$> cre□<tab>
$> create
```

Similarly, if the user types `create` □ and presses <tab>, the system should suggest `account` as valid completions, since this is the only option that can follow a `create` command:

```
$> create □<tab>
account
$> create □
```

While parsing errors help identify syntactically valid completions, they do not capture all useful suggestions. The system should also incorporate contextual information. For instance, if the user types `show account` □ and presses <tab>, the system should suggest `mithrandir` and `pippin`, as these are the only accounts defined so far:

```
$> show account □<tab>
mithrandir pippin
$> show account □
```

```
1 <cmd> ::= 'show' <show_cmd>  | 'grant' <grant_cmd>
2         | 'create account' <account_handle>
3           [ 'with' 'name' <account_name> ]
4
5 <show_cmd> ::= 'account' <account_handle> | 'accounts'
6
7 <grant_cmd> ::= 'role' <account_role>
8                 'to' <account_handle>
9
10 <account_role> ::= 'admin' | 'analyst' | 'curator'
11
12 <account_handle> ::= identifier
13 <account_name>   ::= stringLiteral
```

(a) DPella DSL's grammar (selected parts)

```
1  data Cmd =
2    | ShowAccount RTS.AccountHandle
3    | ShowAccounts
4    | GrantRole RTS.Role RTS.AccountHandle
5    | CreateAccount RTS.AccountHandle (Maybe RTS.AccountName)
```

(b) DPella DSL's command data type

Fig. 2. We parse DPella DSL's programs into DPella DSL's commands

In the following sections, we outline the implementation of such an autocompleter, showcasing its simplicity (around 55 lines of Haskell code). The full artifact—parser, autocompleter, and running examples—is available at https://github.com/dpella/autocomplete-fp-pearl.

3 Grammars and Parsers

The selected parts of the grammar of the DPella DSL language are defined in Figure 2a (for simplicity, the grammar does not account for semicolons at the end of commands). This grammar defines the syntactic rules for the core commands of our language: `show`, `grant`, and `create`. The entry point to the language is captured by the `<cmd>` rule (see lines 1 – 3), which represents a command that can be one of these three kinds. The `<show_cmd>` rule in line 5 describes how users can list all accounts or display specific ones, while the `<grant_cmd>` rule (lines 7–8) supports assigning roles defined in the `<account_role>` rule (line 10). Category `<account_handle>` is parsed by `identifier` (a bare token: letters, digits, and underscores, with a letter as the first character), while `<account_name>` is parsed by Megaparsec's standard `stringLiteral` combinator (a double-quoted literal, with the quotes acting as delimiters and not part of the value)—see lines 12–13. In the `create account` rule (see line 3), the component `[ 'with' 'name' <account_name> ]` indicates that specifying an account name is optional. This optional syntax allows users to create an account using only a handle or to include a name for additional context. The distinction between account handles and account names is crucial for the language: the former uniquely identifies users within the system, while the latter is a descriptive, human-readable attribute that provides additional context but is not used for identification. Note, however, that the grammar does not enforce the uniqueness of account handles; it only specifies that an account cannot be created without one.

```
1  cmd :: Parser Cmd
2  cmd = showCmd <|> grantCmd <|> createCmd
3
4  showCmd :: Parser Cmd
5  showCmd = do
6    keyword "show"
7    (keyword "account" *> (ShowAccount <$> rawAccountHandle))
8    <|> (keyword "accounts" $> ShowAccounts)
9
10 grantCmd = do
11   keyword "grant"
12   keyword "role"
13   role <- accountRole
14   keyword "to"
15   handle <- rawAccountHandle
16   return (GrantRole role handle)
17
18 accountRole :: Parser RTS.Role
19 accountRole = do
20         (keyword "admin" $> RTS.Admin)
21     <|> (keyword "curator" $> RTS.Curator)
22     <|> (keyword "analyst" $> RTS.Analyst)
23
24 rawAccountHandle :: Parser RTS.AccountHandle
25 rawAccountHandle =  RTS.AccountHandle <$> identifier
```

Fig. 3. Parser implementation for selected DPella DSL's commands

The syntax defined in the grammar provides the foundation for the parser, which translates user input into values of the `Cmd` data type shown in Figure 2b. This data type defines the set of commands that users can issue for the REPL, where each command in `Cmd` takes a set of arguments corresponding to the syntactic categories defined in the grammar. Argument types prefixed with `RTS` belong to the runtime system (RTS) and represent the internal data structures for account handles, account names, and roles—the details of these definitions are omitted for brevity. Additionally, optional arguments are represented using the `Maybe` type (see line 5) as is the case for the account name in the `create account` command.

Figure 3 illustrates the standard way to translate grammar rules into Megaparsec combinators. The implementation closely follows the structure of the grammar, where each non-terminal corresponds to a parser combinator. The top-level parser, `cmd` (line 2), attempts to recognize a valid command by delegating to more specific sub-parsers such as `showCmd`, `grantCmd`, and `createCmd` (this last one omitted for brevity). Each of these, in turn, attempt to parse more specific sub-commands using Megaparsec's combinators like `<|>` (choice). For example, the `showCmd` parser (line 4) mirrors the `<show_cmd>` grammar rule—recall Figure 2a, line 5. The parser begins by consuming the keyword `"show"`, then attempts each of the possible show commands in order. If parsing is successful, the corresponding constructor is used to create a `Cmd` value; for instance, the sub-parser in line 8 states that if the input contains the keyword `"accounts"`, then the parser should return a `ShowAccounts` value. Other components of the parser follow an analogous pattern. Megaparsec helper parsers

```
1  Left (ParseErrorBundle {
2      bundleErrors =
3        TrivialError 11 (Just EndOfInput)
4          (fromList [Tokens ('a' :| "dmin"),
5                     Tokens ('a' :| "nalyst"),
6                     Tokens ('c' :| "urator")])
7          :| [],
8      bundlePosState =
9        PosState {
10         pstateInput = "grant role ",
11         pstateOffset = 0,
12         pstateSourcePos = SourcePos {
13                             sourceName = "<path/to/file>",
14                             sourceLine = Pos 1,
15                             sourceColumn = Pos 1},
16         pstateTabWidth = Pos 8,
17         pstateLinePrefix = ""}})
```

Fig. 4. The parse error for `"grant role"`

like `keyword` and `identifier` encapsulate common parsing logic, improving readability and maintainability. Similarly, the use of applicative combinators (e.g., `$>` and `<*>`) ensures that parsed values are neatly packaged into the appropriate constructors. With the parser in place, we can now define a function that runs the `cmd` parser, ignoring whitespace and accepting optional semicolons at the end of commands, and returns either the parsed command or an error:

```
1 -- | Parse a command and return the result or an error
2 parseCmd :: FilePath → Text → Either ParseError Cmd
3 parseCmd = runParser
4   (ignore_whitespace *> (cmd <* optional semi) <* eof)
```

For example, calling `parseCmd "<path/to/file>" ("grant role ")` produces the parsing error shown in Figure 4. (While Megaparsec presents errors in a more user-friendly format, we provide the complete content here for clarity.) .

This message provides several key details: (i) *location information:* line 3 tells us that this error occurred at character 11 (0-indexed) of the input, where the parser unexpectedly reaches the end of the input; (ii) *expected tokens:* lines 4–6 show us the list of tokens the parser was expecting to see instead of end of input; and (iii) *parser state:* lines 9–17 show us the internal state of the parser at the point of the error, such as the input so far, the offset, and the source position. From all of these, the expected tokens are the ones that the parser is expecting to see at the point of the error. These are derived directly from the syntax, i.e., from the `keyword` function used in Figure 3. Concretely, in the DPella DSL language, the parser expects to see a `"admin"`, `"analyst"`, or `"curator"` when the user writes a `grant role` command.

4 Suggesting Syntactic Keywords

To transform errors into suggestions, we need to extract the expected tokens from the parser error. We start by defining a data type to represent the expected tokens, as seen in Figure 5. The `ExpectedToken` data type captures tokens that the parser expected when the error occurred. We define function `expectedTokens` to extract the expected tokens from Megaparsec's constructor

```
1 data ExpectedToken = Word String
2
3 expectedTokens   :: ParseError → [ExpectedToken]
4 parseErrorOffset :: ParseError → Int
```

Fig. 5. Capturing expected tokens from parsing errors

```
1  completer :: String → String → [String]
2  completer line word =
3    case parseCmd "<interactive>" (Text.pack line) of
4      Right _ → []
5      Left err →
6        let error_offset = parseErrorOffset err
7            suggestions = expectedTokens err
8            valid_suggestions =
9              filter (isValidSuggestion error_offset (line <> word)
                   word) suggestions
10       in if null valid_suggestions && not (null word)
11            then [word]
12            else show <$> valid_suggestions
13   where
14     isValidSuggestion :: Int → String → String → ExpectedToken →
         Bool
15     isValidSuggestion offset full_line word =
16       (length full_line - length word == offset &&) .
17         \case Word suggest → isPrefixOf word suggest
```

Fig. 6. Autocompleter for suggesting syntactic keywords

`ParseErrorBundle` (Figure 4) of type `ParseError`. The function uses pattern matching to extract the information inside the constructor `TrivialError` (Figure 4, lines 4–6) and returns the expected tokens as a list of values of type `ExpectedToken`—we omit this function for brevity. We also define the `parseErrorOffset` function to extract the offset from the error message (Figure 4, line 3).

Now, we define the function that computes the suggestions in Figure 6. It is a pure function that receives the input line of commands (excluding the word being completed) (`line`) and the current word at the cursor (`word`). To determine the valid suggestions, the line is parsed using our runner function `parseCmd` (described in Section 3). If parsing succeeds, no suggestions are returned (see line 4), indicating that the introduced sequence of commands constitutes a valid expression in the language. Otherwise, the parsing error is captured as it provides an offset and expected tokens that will guide the suggestions. Using `expectedTokens`, the function generates possible completions based on the error message (line 7). The list of suggestions is then filtered using `isValidSuggestion` to ensure the suggestions match the user's input and align with the error offset. A suggestion is valid when two conditions hold: (i) the current word is a prefix of the suggested token, and (ii) the error offset is exactly at the start of the current word. Condition (i) narrows the alternatives to what the user has typed: at `gr`□ (i.e., `line = ""` and `word = "gr"`), the parser expects `create`, `grant`, and `show`, but only `grant` matches. Condition (ii) means the parser consumed everything before the word and failed right where the user is typing, ruling

out errors earlier in the line: given `grant rol □` (i.e., `line = "grant rol "` and `word = ""`), the parser fails at `rol` rather than at the cursor, so nothing is suggested (completing to `grant rol role` would not help). As a case where both conditions cooperate on non-empty inputs, consider `show acc□` (i.e., `line = "show "` and `word = "acc"`): the parser consumes `show` successfully and then fails at `acc`, expecting `account` or `accounts`. The failure offset matches the start of the word, and `"acc"` is a prefix of both keywords, so both are suggested. Finally, the valid suggestions are returned as strings via their `Show` instance. If none are valid and the user has typed something, the current word itself is returned as a fallback suggestion (line 11), leaving the input untouched. For our REPL, we utilize Haskeline, a Haskell library that provides a user-friendly interface for building command-line applications with advanced input features. It offers simple hooks for providing functions for autocompletion. Wiring up our `completer` is done via a thin adapter:

```
autocompleter :: Monad m ⇒ CompletionFunc m
autocompleter = completeWordWithPrev Nothing " " $
  \reversed_line word →
    return (map simpleCompletion
                (completer (reverse reversed_line) word))
```

What is important is that, when a user types a command in the REPL, they receive suggestions based on the expected tokens from parser errors. For example:

```
$> grant role □
admin    analyst  curator
$> grant role □
```

5 Suggesting Syntactic Categories

Using the same principle of exploiting error information, we can go beyond that merely suggesting keywords and suggest syntactic categories required as input by the user. Our strategy has been to *employ parser labels (a feature commonly support by many parsers) to denote such categories.* For example, we would like to indicate when the user needs to provide an account handle. By using the `label` function from Megaparsec, we can attach a label to `rawAccountHandle` parser:

```
rawAccountHandle :: Parser RTS.AccountHandle
rawAccountHandle = label "<account_handle>"
                   (RTS.AccountHandle <$> identifier)
```

When the parser expects an account handle, the error message will now include the label `<account_handle>`:

```
$> show account □
Parse error: <interactive>:1:14:
|
1 | show account
|              ^
unexpected end of input
expected <account_handle>
```

We then extend our `ExpectedToken` type as follows:

```
data ExpectedToken = Word String | Category String
```

Accordingly, we slightly modify the `expectedTokens` function to extract the label information from the parse error message (recall Figure 4), and just update the `isValidSuggestion` function (from the function `completer`) as follows:

```
1 isValidSuggestion :: Int → String → String → ExpectedToken → Bool


2 isValidSuggestion offset line word =
3   (length full_line - length word == offset &&) . \case
4     Word suggest → isPrefixOf word suggest
5     Category _ → null word
```

More specifically, a category suggestion is considered valid when the error occurred at the end of the input (i.e. `length full_line == offset`) at line 3), and the user has not written anything more (i.e., `null word` in line 5). For instance, when parsing `show account <account_handle>`, the error offset is at the end of `show account` (the parser fails to parse `<account_handle>`), while the full line extends further. By checking the offset, we ensure that the autocompleter only suggests `<account_handle>` when there is nothing there, and not multiple times. Now we can see the syntactic categories as expected tokens in the autocompleter:

```
1 $> show account □<tab>
2 $> show account <account_handle>□
```

Note that the completed `<account_handle>` is the name of the syntactic category, and the user is expected to overwrite it with an actual account handle (e.g., `mithrandir`).

6 Context-Aware Suggestions

In this section, we take advantage of the *existing REPL commands to provide valid suggestions for syntactic categories where the user needs to provide input.* Intuitively, the idea is that when the autocompletion needs an input from, for instance, the syntactic category `"<account_handle>"`, then the REPL executes the command `show accounts`, collects its output and proposes autocompletion options based on that data. In this light, we need parsers that, inside error messages, can carry commands responsible to generate suggestions. Luckily, Megaparsec is parametric on user-defined error messages, which are called *fancy error messages*. These fancy errors provide more detailed error messages, whereas *trivial errors* are used for simpler cases, such as unexpected tokens.

By default, Megaparsec uses the empty datatype `Void` as custom error message (`type Parser a = Parsec Void Text a`). So, we change the definition of `Parser` to support error messages capable to carry out commands:

```
1 type Parser a = Parsec LookupCmd Text a
```

where values of type `LookupCmd` embed commands within parser errors.

```
1 data LookupCmd = LookupCmd {
2   lookup_error_reason :: String, -- the error
3   lookup_label :: String, -- the label of the parser
4   lookup_cmd :: Cmd -- the command to execute
5 }
```

To incorporate commands as generators of suggestions, we extend `ExpectedToken` type with a new constructor:

```
data ExpectedToken = Word String | Category String
                   | Command LookupCmd
```

Accordingly, the `expectedTokens` function is then updated to extract the commands from the parser error. In a nutshell, this function pattern matches on the `FancyError` constructor from Megaparsec as well as on the `TrivialError` constructor (recall Figure 4).

What remains is to extend our parser with a combinator that allows us to attach commands into the error messages. For that, we introduce the combinator `labelWithCommand`:

```
labelWithCommand :: String → Cmd → Parser a → Parser a
labelWithCommand lbl cmd parser = do
  observing parser >>= \case
    Right res → return res
    Left err → fancyFailure
                 $ Set.singleton
                 $ ErrorCustom
                 $ LookupCmd (errorReason err) lbl cmd
  where
    errorReason :: ParseError Text LookupCmd → String
    errorReason = ... -- omitted for brevity
```

This function uses the `observing` combinator from Megaparsec that allows us to capture any errors that occur during parsing, and transform them into a `FancyError` (`fancyError`) with the lookup command attached (`LookupCmd (errorReason err) lbl cmd`). In order to keep the error message informative, we also preserve the original error reason (`errorReason err`) and label (`lbl`) as arguments to the constructor `LookupCmd`. With this in place, we can now define new parsers for account handles that include the lookup command `show accounts` (denoted by `ShowAccounts`):

```
accountHandle :: Parser RTS.AccountHandle
accountHandle =
  labelWithCommand "<account_handle>" ShowAccounts
    (RTS.AccountHandle <$> identifier)
```

Note that we must take care to make the distinction between *raw* account handles and account handles coming from the RTS: when we are autocompleting `show account □`, we want to suggest the accounts in the system, but when we are autocompleting `create account □`, we do not want to suggest any accounts, as we are creating a new account. So we update *only* the `showCmd` and `grantCmd` parsers, using the new, lookup enabled, parser `accountHandle`:

```
showCmd :: Parser Cmd
showCmd = do
  keyword "show"
  (keyword "account" *> (ShowAccount <$> accountHandle))
  <|> (keyword "accounts" $> ShowAccounts)

grantCmd :: Parser Cmd
grantCmd = do
  ... -- as before
  handle <- accountHandle -- changed to accountHandle
  return (GrantRole role handle)
```

Another small change (in terms of lines of code), but significant from the conceptual perspective is that the definition of `suggestions` in the function `completer` cannot be side-effect free any longer, i.e., it might need to execute commands (recall Figure 6):

```
1   let suggestions = expectedTokens err
```

We need to introduce `suggestions` as a bind in the same monad [14] as the function `completer`, which in turn runs in the same monad as the REPL.

```
1   suggestions <- f (expectedTokens err)
```

What should be a suitable function `f`? In this case, we need a function that traverses the expected tokens, leaves untouched the suggestions that are keywords (constructor `Word`, Section 4) or syntactic categories (constructor `Category`, Section 5), but executes commands within the current REPL context to generate more suggestions. For that, we introduce a new `completer`'s auxiliary local function as follows.

```
1 expandSuggestions :: Monad m ⇒ ExpectedToken → m [ExpectedToken]
2 expandSuggestions = \case
3  s@(Word _) → return [s]
4  s@(Category _) → return [s]
5  s@(Command (LookupCmd _ _ cmd)) → do
6    output <- executeCommand cmd
7    case output of
8      Nothing → return [s]
9      Just suggestions → return $ Word <$> suggestions
```

In the code above, `executeCommand` is treated as a REPL-supplied callback of type `Cmd → m (Maybe [String])`. Because the callback carries whatever effects the REPL commits to, `expandSuggestions` needs the `Monad m` constraint. When `executeCommand` returns `Nothing` —signalling that the lookup command could not run successfully in the current REPL state (e.g., because no user is logged in)—the original `Command` token is returned. Since the `Show` instance of `ExpectedToken` simply renders a `Command` token as its category label, the completer displays the label as a placeholder—giving the user a hint about the expected input rather than an empty suggestion list.

The interesting case is for the `Command` constructor (lines 5 – 9). The function runs the associated command within the current REPL context to generate suggestions by calling `executeCommand` (line 6). With this function, we can complete the binding for `suggestions`:

```
1   suggestions <- concatMapM expandSuggestions (expectedTokens err)
```

Function `concatMapM` simply applies `expandSuggestions` to each token, sequences the execution of each command that generates suggestions, and flattens the lists of provided suggestions into a single list.

Similarly to what we did in Section 5, we need to extend `isValidSuggestion` to make sure that suggestions based on the result of executing commands trigger when the user has not written anything more (i.e., `null word`).

```
1   isValidSuggestion offset line word = ...
2     -- as before
3     Command _ → null word
```

To conclude, we adapt the `completer` function to accept the *running function* (`executeCommand`) as a parameter provided by the REPL.

```
completer :: Monad m ⇒
             (Cmd → m (Maybe [String])) →
             String → String → m [String]
```

Now, when we press <tab> to request a suggestion for the `show account` command, we get the list of created accounts:

```
$> show account □<tab>
mithrandir pippin
$> show account □
```

Note, however, that for raw handles we still get the same behavior as before:

```
$> create account □<tab>
$> create account <account_handle>□
```

As we have seen in this section, by enriching our parser with custom errors and exploiting the REPL's context, we can provide context-aware suggestions to streamline the user experience with few lines of code and minimal changes to the existing parser, making it a lightweight and effective way to enhance autocompletion in REPLs.

7 Discussion

While the approach we have presented provides a lightweight and adaptable autocomplete for REPL-driven DSL, it is not without limitations. We stress that our goal is not a fully-featured autocomplete in the spirit of mature IDEs, but a low-effort path to useful suggestions for DSL designers. Below we make explicit the trade-offs that come with this choice. In this section, we discuss simple extensions, key challenges and opportunities for future work.

Valid input. Since the suggestions of the autocompleter are based on parser errors, this can be a limitation when users expect suggestions for valid inputs that are *optional* in the grammar. Since the absence of optional inputs does not trigger a parser error, the autocompleter will not provide any suggestions. A straightforward solution consist in intentionally triggering a parser error when an optional input is expected. This can be done by adding a dummy token that is not part of the grammar, thus allowing the autocompleter to generate appropriate suggestions. For instance, to suggest both `account` and `accounts` when the user has already typed `account`□, we inject a dummy token (e.g. `account`⊥) that is guaranteed to fail to parse, and account for it in the offset; this way the parser still produces the expected-token set even though the prefix is itself a valid keyword.

Parser restrictions. The autocompleter's capabilities are closely tied to the parser's design. For instance, if the parser uses the common `oneOf` combinator, the autocompleter will not be able to suggest completions for the different options. This is because `oneOf` is implemented as `oneOf cs = satisfy (\c → elem c cs)` which prevents introspection of the list of valid characters, as the function passed to `satisfy` cannot be inspected to retrieve the valid characters—observe that the body of the function could be an arbitrary side-effect free Haskell expression. This limitation can be mitigated by rewriting the parser to use more granular combinators, such as `try` and `keyword`, which explicitly define valid tokens. However, this approach may impact performance, particularly for large grammars, so balancing efficiency with autocomplete functionality remains an area for further exploration.

Constants. Another challenge is suggesting constants, particularly when the set of valid values is large or infinite. Unlike keywords or identifiers, constants such as integers cannot be exhaustively listed. In the domain of automatic program repair, techniques like *jungloid mining* [8] address this

issue by extracting frequently used constants from existing code. Although such techniques are beyond the scope of our current approach, future work could explore lightweight adaptations of these methods to improve autocomplete suggestions without sacrificing simplicity.

Bindings. The language described in this paper does not support bindings, but if it did, it would be useful to suggest such identifiers from the REPL's context. Special care must be taken to respect the scope of the bindings and to avoid suggesting identifiers that are not in scope. This could be achieved by extending the autocompleter to query the context, and look for bindings that match the word the user has typed so far. However, a more sophisticated REPL context would be required — one that tracks both the scope and category of identifiers (derived from the labels) to prevent suggesting identifiers of the wrong category. A similar approach to suggesting valid hole-fits for typed holes in Haskell [7] could then be applied, utilizing these categories.

Extending to general DSLs. Extending our technique to general DSLs has its challenges. Firstly, our technique only triggers on the parser's first error, and is most useful when the cursor is at the end of the user's input—which matches how a REPL is typically used. This means that, in a longer file, a syntax error far before the cursor will steal the suggestions, and that errors detected later than where they occur (the classic "missing closing brace" detected at end of input) cannot be diagnosed precisely. Combining our approach with parser *error recovery* (e.g., resuming parsing after a syntax error to continue producing diagnostics) is also incompatible with our pure-error-driven view: once the parser swallows the error, the information that drives the suggestions is gone. Secondly, we do not claim semantic context awareness. For instance, in a query language we cannot, by ourselves, restrict the column suggestions for `from people select` □ to columns of the `people` table; the lookup command we trigger for the syntactic category does not know which table is in scope. Our suggestions are context-aware only in the sense that they depend on the parser's state at the cursor. Closing the gap to semantic awareness requires a richer protocol between the parser, the autocompleter, and the host language's runtime—an interesting direction for future work that would step outside the lightweight regime of this pearl.

Cost on parser design. The technique imposes a few small costs on the DSL author. Identifier-style parsers should contain a label indicating the syntactic category (e.g., `<account_handle>`), otherwise the lookup command has nothing to dispatch on. These costs, however, are modest—our entire change to the production parser fits in a few dozen lines—but they are part of the bargain.

Making the parser-errors-as-suggestions pattern explicit. Given the code we present, a natural question is whether `ExpectedToken`, `expandSuggestions`, and `isValidSuggestion` could be extracted into a reusable library, with per-parser adaptors for Parsec, Attoparsec, Megaparsec, and beyond. In principle, yes: once the DSL's command AST is fixed, most of this code is largely generic—only the REPL commands wired into `LookupCmd` are DSL-specific. We leave that endeavour as future work; our focus here is instead on showing how conceptually simple, and how few lines of code, it takes to obtain useful autocomplete for a REPL-driven DSL directly from parser's error messages.

Applicability beyond Megaparsec. The minimal feature set we rely on is: (i) parser combinators, (ii) errors that expose expected tokens, (iii) attachable labels, and (iv) enrichable errors with custom data. We ported the completer of Section 4–Section 6 to three non-Haskell hosts: Parsy[5] (Python), chumsky[6] (Rust), and FastParse[7] (Scala). Properties (i)–(iii) hold in every library. Only

[5] https://github.com/python-parsy/parsy
[6] https://codeberg.org/zesterer/chumsky
[7] https://github.com/com-lihaoyi/fastparse

(iv) discriminates: none of Parsy's `ParseError`, chumsky's `Rich`, nor FastParse's `Parsed.Failure` carries arbitrary payloads. These libraries can only attach *label strings* to errors—not the command that should be run to expand a label into concrete suggestions. We recover the same effect with a *label-to-command lookup table*: a module-scope dictionary mapping each labelled category to the `Cmd` that produces its suggestions. Concretely, when the user types `show account ␣`, the parser fails with the label `<account_handle>` in its expected set; the completer looks up that string in the table, retrieves `ShowAccounts`, executes it against the REPL state, and turns the returned list of handles into suggestions. In Megaparsec the same information rides directly on the error via `ErrorCustom`; without it, the side-table plays the messenger. Each port defines this table in around 25 lines. One trade-off follows from keying by the label string: the same label can no longer encode two different lookup intents, because both would collide on the same table key. Megaparsec's `ErrorCustom` avoids this by discriminating structurally on the error tree; the lookup table forces *distinct labels for distinct lookup intents* instead (`<account_handle>` when an existing handle should be suggested versus `<new_account_handle>` when a fresh one should not). In effect, the label ceases to be a pure name for a syntactic category and doubles as a marker of the completer's intent—a loss of conceptual crispness that DSL authors have to accept in exchange for the workaround. A second cross-library distinction surfaces in FastParse, whose `traced.expected` reports expectations *anywhere along the failure path* rather than at the cursor; this is reconciled with a small post-processing pass that drops tokens already consumed in the prefix and atomises combinator fragments. The technique itself is unaffected.

8 Related work

Tool-supported DSL. Modern programming tools can derive syntax highlighting, parsing, and completion from a grammar specification, but using it ties the DSL to the tool's ecosystem and requires learning its frameworks (e.g., Spoofax [10], Langium [11], Xtext [5], and ANTLR4 [16]). These approaches excel in functionality but can be heavyweight—they often demand a steep learning curve or reimplementation of the DSL in the workbench's format. Langium and Xtext, in particular, derive a language server from a grammar written in their own DSL, including completion, scoping, and validation; they are excellent when one is willing to commit to that ecosystem, but require maintaining the DSL grammar twice (once in the runtime parser, once in the workbench). ANTLR offers a similar story: a generated parser plus auxiliary libraries (e.g. the "ANTLR4 autocomplete" family of projects) can drive completion from the parser's expected-token set, with varying degrees of integration with error recovery. In contrast, our work follows a more lightweight philosophy by leveraging the *existing DSL parser*—thus avoids duplicate language implementations and lets REPL-driven DSLs bootstrap completion from a handful of labels and lookup commands, rather than a full grammar workbench setup.

Grammar-driven code completion. The work by Amorim et al. [2] introduce a *placeholder* mechanism for syntactic completion: the parser treats incomplete code by inserting placeholder nodes, and their system automatically generates all syntax fragments that could validly replace a placeholder. Our completion strategy is conceptually similar but realized in a lightweight way, i.e., directly using parser combinators rather than extending the language's grammar to include placeholder constructs and implement the full mechanism in Spoofax.

Statistical approaches. Some autocompletion aspects has been addressed by both statistical and semantic approaches. Allamanis et al. [1] propose a technique to suggest method and class names based on code context. Bruch et al. [4] demonstrate that learning from existing code can improve completion by suggesting method calls based on different heuristics (e.g., frequency based). Zhang et al. [19] combine various features (code semantics, naming conventions, historical renames) in

a machine learning model to recommend better names for inconsistent identifiers. All the work above assumes large repositories of code to learn from. In the context of DSLs, especially a new or niche one, such resources are usually unavailable, which makes ML-based name suggestions not feasible.

Type-driven suggestions. Introduced in the Haskell compiler (GHC), typed holes allow developers to write incomplete programs by marking unknown expressions with an underscore (_) [7]. During compilation, the type checker infers the expected type at each hole and reports a list of in-scope expressions that match the required type. This feature turns types into interactive development guides, offering lightweight yet powerful support for incremental programming. Type-directed autocompletion is also present in languages like Agda [15] and Idris [3], where holes are considered as proof goals. The environment then can search for expressions (including function calls or combinations of them) that have the required type to fill the hole. Pelsmaeker et al. [17] provide sound and complete code suggestions based on formal language definitions and static semantics. The authors leverage Statix static semantics specification language (part of Spoofax) to automatically derive code completion services for statically typed programming languages. Implementing something similar to type-driven suggestions requires a full formalization of the DSL's typing and scoping rules. Our approach does not aim to such expressiveness—we prioritize lightweight implementation over absolute correctness of suggestions.

9 Conclusion

In this paper, we have shown that autocomplete can be implemented with simplicity by leveraging parsers—in around 55 lines of Haskell code. By extracting expected tokens from parser errors, we can generate syntactically and semantically relevant suggestions. We have also shown that integrating autocomplete into a REPL enables context-aware suggestions, providing more accurate and helpful completions to users.

We expect this fresh perspective on repurposing parsers for autocomplete brings benefits to REPL-driven DSLs designers and potential users. By deriving an autocomplete system directly from the parser, developers can provide robust tooling from the earliest stages of language development. This not only improves the user experience but also makes new languages more approachable, fostering faster adoption and a more productive use. We hope this pearl encourages the authors of such tools to reach for the parser they already have when their users start asking for `<tab>`.

Acknowledgments

We thank the anonymous reviewers for their thoughtful feedback, which helped improve this paper. We are grateful to Josef Svenningsson for encouraging us to write this pearl, and to both Josef Svenningsson and Andres Löh for their comments on early versions of the paper. We also thank Agustín Mista, who drafted the initial autocomplete code that seeded this work. This work was partially funded by the Swedish Research Council (VR).

References

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery. doi:10.1145/2786805.2786849

[2] Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. 2016. Principled syntactic code completion using placeholders. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016)*. Association for Computing Machinery. doi:10.1145/2997364.2997374

[3] Edwin C. Brady. 2011. IDRIS —: systems programming meets full dependent types. In *Proceedings of the Workshop on Programming Languages Meets Program Verification (PLPV '11)*. Association for Computing Machinery, 43–54. doi:10.1145/1929529.1929536

[4] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from examples to improve code completion systems. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. Association for Computing Machinery, 213–222. doi:10.1145/1595696.1595728

[5] Eclipse Foundation. 2022. Eclipse Xtext. Accessed February 2025. www.eclipse.org/Xtext

[6] C. N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, and Daniel L. Stock. 1984. The Poe language-based editor project. *SIGSOFT Softw. Eng. Notes* 9, 3 (1984). doi:10.1145/800020.808245

[7] Matthías Páll Gissurarson. 2018. Suggesting Valid Hole Fits for Typed-Holes (Experience R eport). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 179–185. doi:10.1145/3242744.3242760

[8] Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands. 2022. PropR: property-based automatic program repair. In *Proceedings of the 44th International Conference on Software Engineering*. 1768–1780. doi:10.1145/3510003.3510620

[9] Richard D Hipp. 2025. SQLite. https://www.sqlite.org/index.html

[10] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. doi:10.1145/1869459.1869497

[11] Langium Contributors. 2022. Langium: Next-gen language engineering framework. Accessed February 2025. https://langium.org

[12] Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. 2020. A Programming Framework for Differential Privacy with Accuracy Concentration Bounds. In *IEEE Symposium on Security and Privacy (SP)*. doi:10.1109/SP40000.2020.00086

[13] Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. 2021. A Programming Language for Data Privacy with Accuracy Estimations. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 6 (June 2021), 42 pages. doi:10.1145/3452096

[14] Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92. doi:10.1016/0890-5401(91)90052-4

[15] Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD Thesis. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

[16] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf. https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference/

[17] Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. 2022. Language-parametric static semantic code completion. *Proc. ACM Program. Lang.* OOPSLA1 (2022). doi:10.1145/3527329

[18] S. Doaitse Swierstra and Luc Duponcheel. 1996. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming*. Springer Berlin Heidelberg. doi:10.1007/3-540-61628-4_7

[19] Jingxuan Zhang, Junpeng Luo, Jiahui Liang, Lina Gong, and Zhiqiu Huang. 2023. An Accurate Identifier Renaming Prediction and Suggestion Approach. *ACM Trans. Softw. Eng. Methodol.* (2023). doi:10.1145/3603109

Received 2026-05-20; accepted 2026-06-30