

Mapping and explaining syntax errors with LRgrep

Frédéric Bour¹ and François Pottier¹

¹Inria Paris

LR parsers and generated parsers are often criticized for their perceived inability to produce good syntax error messages. Yet, we believe, this inability is not inherent to LR parsing or to the use of a parser generator. Instead, we claim, this perception is caused mainly by the lack of languages and tools that allow the problem of producing good syntax error messages to be addressed at a suitable level of abstraction.

In this paper, we demonstrate how to use the LRgrep language and its compiler to construct good syntax error explanations for an LR parser. We focus on a toy language of arithmetic expressions, with global and local variable definitions, whose parser is built by the Menhir parser generator. The LRgrep tool helps us discover all of the error situations that this parser can encounter. The LRgrep language lets us describe each situation via a succinct pattern and provide code that constructs an explanation message for this situation. Our description of all error situations and messages fits in one page of LRgrep and OCaml code.

1 Introduction

Parsing is by now a well-understood problem for which many techniques and tools have been developed [GJ08]. In spite of this, the selection of a parsing technique and the choice between generated parsers and hand-written parsers remain the subjects of sometimes heated debates. These two questions are distinct, yet related, because most hand-written parsers use recursive descent. A blog post [Cox10] by Russ Cox, one of the main developers of the programming language Go, summarizes them very well: *“One of the great religious debates in compiler writing is whether you should use parser generators like yacc and its many descendants or write parsers by hand, usually using recursive descent. On the one hand, using parser generators means you have a precise definition of the language that you are parsing, and a program does most of the grunt work for you. On the other hand, the proponents of hand-written recursive descent parsers argue that parser generators are overkill, that parsers are easy enough to write by hand, and that the result is easier to understand, more efficient, and can give better error messages when presented with syntactically illegal programs. Like in most religious debates, the choice of side seems to be determined by familiarity more than anything else.”*

Indeed a surprising number of industrial-strength parsers appear to be hand-written [Kla20, Eat21]. A desire to produce better syntax error messages or to perform better error recovery appears to be one of the key reasons why hand-written recursive descent parsers are preferred over generated parsers and over parsers based on techniques other than recursive descent, such as LR parsing.

This state of things might be explained, we believe, by a combination of factors, some of which are caused by the limitations of a particular implementation, some of which are more intimately related with the essence of LR parsing. First, because a generated parser is optimized for efficiency, it uses low-level representations of its data structures and possibly does not offer a programmatic way of inspecting these data structures. Second, the data structures used in an LR parser, most prominently its stack, have a certain essential complexity. Although LR parsing is deterministic (that is, it does not require backtracking), it involves a form of parallel exploration of several possibilities. At a given point in time, the manner in which the input fragment that has been read must eventually be interpreted is not yet known: its interpretation depends on the remaining input. Therefore, the stack simultaneously represents several possible ways of understanding the past, and with each permitted interpretation of the past, comes a set of legal continuations. Therefore, regardless of how the stack is represented in memory and through which API it is accessed, explaining how an LR parser has understood what it has read, and describing what continuations it expects or accepts, can seem a challenging task.

Yet, we claim, if this task is approached at a suitable level of abstraction and with suitable tools, it need not be feared.

Although LR parsing is often taught and understood in terms of LR automata, it can also be explained directly at the level of the context-free grammar. In this high-level view, the current state of the parser is fully described by just two pieces of data, namely:

1. the *stack*, a sequence of terminal and non-terminal symbols;¹
2. the remaining input, a sequence of terminal symbols.

The stack represents the input fragment that has been read so far and the manner in which this fragment has been interpreted. The parser’s state is updated by just two kinds of actions: *shifting* extracts one terminal symbol out of the remaining input and pushes it onto the stack; *reducing a production* transforms the stack by replacing the right-hand side of this production (which must appear at the top of the stack) with its left-hand side.

Although a great deal of care and effort is usually devoted to the problem of determining which action the parser must take at each step of its normal execution, here, we wish to address a different problem: once the parser has detected a syntax error and failed, we wish to produce a “good” syntax error message. We do not aim to determine the true cause of the error or suggest an optimal repair: our purpose is just to produce a message.

We believe that a good error message should usually describe what the parser has understood so far and what continuation or continuations were expected at the point where the parser failed. Occasionally, a message may suggest a likely repair.

To determine which message should be produced, one must analyze the parser’s stack, a string of terminal and non-terminal symbols. Our main insight is that *the desired analysis should and can be expressed in a high-level, declarative manner*, in the traditional form of a case analysis by pattern matching. That is, a case analysis should take the form of a list of *clauses*, where each clause consists of a *pattern* (which recognizes certain stacks) and an *action* (which constructs an error message).

To enable this, we design a domain-specific language of patterns, named LRgrep, and we implement a compiler for this language, also named LRgrep. In addition to the standard constructs of regular expressions, namely literal symbols, concatenation, alternation, and Kleene star,² LRgrep offers several domain-specific constructs. In particular, a *filter pattern*, written */ item* (Section 5.1), tests whether a stack admits a given continuation; and a *reduce pattern*, written *[pattern]* (Section 5.5), tests whether a stack, once transformed by zero, one, or more reduction steps, conforms to a given pattern.

¹In an efficient executable implementation, the stack is usually actually represented as a sequence of states of the LR automaton. Once an LR automaton has been fixed, the two representations convey the same amount of information: in both cases, the stack determines a path through the automaton, from an initial state to some current state. The symbols that label the edges along this path form the stack, viewed as a sequence of symbols. The states traversed by this path form the stack, viewed as a sequence of states.

²Although alternation and Kleene star seem to be rarely useful in practice, LRgrep does support them.

We do *not* automate the task of constructing an error message. The main reasons for this decision are as follows:

- We believe that it is difficult to automate this task while reaching a high level of quality. Although one can imagine several systematic procedures for producing error messages, many of them produce messages that are difficult to read because they contain too much detail, mention too many possible continuations, or mention syntactic categories (non-terminal symbols) whose name is not known to the end user.
- Not everyone agrees on what constitutes a “good” error message. We do not wish to impose our view on the users of our tool.
- In our approach, the author writes a pair of a pattern and a message. The pattern describes a set of error situations; the message must describe and explain all of these situations. Therefore, the wording of the message must depend on the pattern. With a very precise pattern, one can associate a very specific message. With a broader pattern, one must associate a more general message.

For these reasons, we delegate the task of constructing an error message to the user of LRgrep. In each clause, the *action* is a piece of OCaml code, which the user of LRgrep must provide. If the user is satisfied with a fixed error message, this code can be as simple as a string literal. If the user wishes to construct a more elaborate message, which may contain source code locations or source code fragments, it is up to her to write more elaborate code. This paper provides examples, which, for brevity, we do not explain in detail.

This is a tutorial paper. The LRgrep language and compiler are presented from the point of view of an end user. A single extended example, a parser for a toy language, demonstrates how LRgrep is used and illustrates the meaning and expressive power of its patterns. Our toy language, a calculus of arithmetic expressions with global and local variable definitions, is a tiny fragment of the OCaml language, so the questions and difficulties that appear while handling this language are (to a limited extent) representative of those that would arise in an effort to handle all of OCaml. [The full source code of this example](#) is publicly available.

The LRgrep compiler is installed by “`opam update && opam install lrgrep`”. [Its source code](#) is publicly available. LRgrep is designed to work in cooperation with the Menhir parser generator [PRG25]. It is implemented entirely outside Menhir, but exploits some of Menhir’s features, namely its “SDK”, which offers access to descriptions of the grammar and automaton, and its “incremental parsing” and “stack inspection” APIs, which allow undoing the reductions that took place just before the failure³ and inspecting the parser’s stack. LRgrep could be adapted for use with other LR parser generators, such as bison [DS21], LALRPOP [Mm25], or Tree-sitter [Bmc25], if they offer similar facilities—which, as far as we know, is *not* currently the case. LRgrep is compatible with all of the parsers that Menhir can generate: in particular, it can deal with grammars that lie outside of the class LR(1) and whose shift/reduce or reduce/reduce conflicts have been resolved, either in an arbitrary way or by obeying user-provided priority declarations. In fact, our toy parser uses priority declarations. The theory and implementation of LRgrep are described in the first author’s dissertation [Bou24].

³The reduction actions that an LR parser performs during its normal operation are determined by its stack and by the first unconsumed input symbol, also known as the lookahead symbol. A (non-canonical) parser can perform several reductions steps before it detects that the next input symbol is illegal and fails. Such reduction steps are known as *spurious reductions*. When we say that we wish to analyze “the stack at the point where the parser fails”, which stack do we mean? Is it the stack that existed before the spurious reductions took place, the stack that exists after these reductions have taken place, or perhaps some intermediate stack? Before answering this question, let us point out that the spurious reduction steps are possibly misguided: they have been decided based on the current lookahead symbol, which, it turns out, is illegal. Therefore, we prefer to analyze *the stack that existed before any spurious reductions took place*, that is, the stack that existed just after the last legal input symbol was shifted. As we analyze this stack, we may wish to examine the various ways in which this stack *could* be reduced in the presence of lookahead symbols *other* than the illegal symbol that caused the parser to fail. This is made possible by LRgrep’s reduce patterns: indeed, the pattern [*pattern*] matches all of the stacks which can be reduced in zero, one, or more steps to a stack that is matched by *pattern*.

```

1 let x = 3;
2 let y = 4
3 let z = x + y

```

File "foo.ml", line 3, characters 0-3:
3 | **let** z = x + y
 ^^^
Error: Syntax error

File "foo.ml" (3:0-3):
Syntax error.
A local declaration has been read (2:0-9):
 let y = 4
The keyword '**in**' is now expected.
Suggestion: deleting the semicolon
that precedes this declaration (1:9-10)
would allow it to be interpreted as a global declaration.

Figure 1. A potentially confusing syntax error situation

2 A toy language

As an example of a language that we wish to parse, we present a tiny language, named \mathcal{L} , whose structure is as follows. This language is very ordinary; in fact, it is a subset of OCaml.

- A *file* is a sequence of declarations, also known as global declarations.
- A *declaration* binds a variable to an expression. An example is `let x = 21*2.`
- *Expressions* are formed out of
 - Variables, such as `x`.
 - Integer literals, such as `21`.
 - Operations of arity 1, with a prefix notation: integer negation.
 - Operations of arity 2, with an infix notation: integer addition, subtraction, multiplication, division, and sequencing. The semicolon `;` denotes sequencing.
 - Local declarations, whose scope is a subexpression, such as `let x = 21 in x*2.`
 - Expressions enclosed within parentheses.

3 A confusing syntax error

In \mathcal{L} and in OCaml, global declarations and local declarations look alike: both kinds of declarations begin with the keyword `let`, followed with a binding of a variable to an expression. This does not give rise to any ambiguity.⁴ However, it does create a situation where a syntax error has potentially confusing consequences. This situation appears in Figure 1. At first sight, this three-line program seems to contain three global declarations, for the variables

⁴The reader may wonder whether there exists a context where both a global declaration and an expression are acceptable. In the language \mathcal{L} , the answer is negative. In other words, an LR parser for \mathcal{L} is always either in a state where it aims to construct a declaration or in a state where it aims to construct an expression. There is no “undecided” state where the parser has not yet determined whether a declaration or an expression will eventually be constructed. An undecided state appears when one considers an extended language \mathcal{L}' where an expression followed with a double semicolon `;;` forms a global declaration. An LR parser for \mathcal{L}' reaches this undecided state after reading `let x = 0`. Indeed, at this point, it does not yet know whether this input fragment forms a complete global declaration or the beginning of an expression, which itself forms the beginning of a global declaration: say, `let x = 0 in x+1;;`. \mathcal{L}' is an example of a language that lies in the class LR(1) and outside of the class LL(1). We choose to leave this complication out of the language \mathcal{L} . Nevertheless, our methodology is able to deal with \mathcal{L}' . In fact, \mathcal{L}' is a fragment of OCaml, whose LR parser is generated by Menhir.

x , y , and z . However, at the end of line 1, an extraneous semicolon has been mistakenly inserted. An LR parser for the language \mathcal{L} considers this semicolon as a sequencing operator. Therefore, it expects this semicolon to be followed with an expression. For this reason, it interprets line 2 as a *local* declaration, that is, as the beginning of an expression. Upon encountering the **let** keyword on line 3, it fails. Indeed, at this point in a *local* declaration, the keyword **in** is expected; the keyword **let** is not acceptable.

This example illustrates the fact that a parser sometimes detects a failure long *past* the point where the real mistake lies. In this example, the parser fails at the beginning of line 3, whereas the extraneous semicolon lies at the end of line 1. In general, an LR parser fails at the last point where the input fragment that has been read so far is viable, where we say that an input fragment $input[0, i)$ is *viable* if it is a prefix of a syntactically correct file.⁵ In this example, the position of the real mistake and the position where the parser fails are not very far apart. However, by replacing **let** $y = 4$ with a longer declaration, one can construct examples where the two positions are arbitrarily far apart.

The error message that is printed by the OCaml compiler (version 5.3.0) is shown in the middle segment of Figure 1 on a yellow background. The OCaml compiler reports the position where the parser failed but does not attempt to explain why a syntax error occurred.

The error message that is produced by our parser for the language \mathcal{L} is shown in the last segment of Figure 1 on a blue background. Like the OCaml compiler, we report that a syntax error occurred at the beginning of line 3. We always report the position where the parser failed, and do not attempt to infer where the “real” mistake lies, because this is a difficult and ill-defined problem. Then, by exploiting the information contained in the parser’s stack, we explain how the parser has interpreted the input that it has recently read and what input it expects next. Here, in particular, the error message indicates that **let** $y = 4$ has been interpreted by the parser as a *local* declaration, so that the keyword **in** is now expected.

The last part of this message is a white lie: after reading **let** $y = 4$, there exist legal continuations other than the keyword **in**. For example, $*2$ is a legal continuation. By telling the user that the keyword **in** is expected, we simplify reality, and describe a strict subset of the legal continuations. Indeed it would be pointless to describe the many ways in which the expression 4 can be continued to form a larger expression.

As a bonus, we recognize that a possibly undesired semicolon has caused the declaration **let** $y = 4$ to be considered local, and we display a suggestion: deleting this semicolon would allow this declaration to be considered global, which perhaps was the intention of the user. We do not have to do this: we choose to do so because we are aware of the problem that stray semicolons can create and we know that this problem commonly arises in practice. In contrast, although stray arithmetic operators such as $+$, $-$, $*$, $/$ would cause exactly the same problem, we expect them to be less frequent in practice, so we do not treat them in this way.⁶

4 Parsing the toy language

The description of a parser for the language \mathcal{L} appears in Figure 2. This description is stored in the file `parser.mly`. Out of this description, the parser generator Menhir [PRG25] produces OCaml source code for an LR(1) parser, in the files `parser.ml` and `parser.mli`.

Let us briefly review the structure and content of the file `parser.mly`.

The file begins with declarations of *terminal symbols*: `INT`, `IDENT`, `PLUS`, `MINUS`, and so on. A lexical analyzer, or lexer, whose definition is not shown, is in charge of transforming the input text into a sequence of terminal symbols. For example, integer literals, such as 1

⁵If $input$ is a syntactically incorrect file then there exists a unique index i such that $input[0, i)$ is viable and $input[0, i + 1)$ is not viable. The index i is the position where an LR parser fails.

⁶See the example named `input08` in Figure 5.

```

1  (* Tokens. *)
2  %token <int> INT
3  %token <string> IDENT
4  %token PLUS MINUS TIMES DIV EQUAL
5  %token LPAREN RPAREN
6  %token SEMI
7  %token LET IN
8  %token EOF
9  (* Precedence declarations, lowest (first line) to highest (last line). *)
10 %nonassoc IN
11 %right SEMI
12 %left PLUS MINUS
13 %left TIMES DIV
14 (* Entry point. *)
15 %start <unit> file
16 %%
17 (* Productions. *)
18
19 file: declaration* EOF    {}
20
21 declaration: LET binding {}
22
23 binding: IDENT EQUAL expr {}
24
25 expr:
26 | IDENT
27 | INT
28 | expr PLUS expr
29 | expr MINUS expr
30 | expr TIMES expr
31 | expr DIV expr
32 | MINUS expr
33 | expr SEMI expr
34 | LPAREN expr RPAREN
35 | LET binding IN expr
36     {}

```

Figure 2. The parser description file `parser.mly`

and 123, are transformed to the symbol `INT`; identifiers, such as `x` and `length`, are transformed to the symbol `IDENT`; the arithmetic operators `+` and `-` are transformed to the symbols `PLUS` and `MINUS`; and so on.

Next comes a series of *precedence declarations*, or *priority declarations*. Each line in this sequence of declarations represents a distinct priority level. The first line represents the lowest priority level; the last line represents the highest priority level. These declarations are used by the parser generator to statically resolve shift/reduce conflicts, that is, to choose at parser construction time between shifting (reading the next input symbol) and reducing (reinterpreting what has been read so far). This particular LR(1) automaton has 7 states where a shift/reduce conflict appears. For example, the fact that `IN` has lower priority than `TIMES` plays a role in parsing the expression `let x = 2 in 3 * x`. After reading `let x = 2 in 3`, the next input symbol is `*`. One must choose between reducing now (that is, considering `let x = 2 in 3` as an expression) and shifting the input symbol `*` (which later causes `3 * x` to be recognized as an expression). Here, the latter behavior is desired, which is why `TIMES` is given higher priority than `IN`.⁷ Furthermore, each priority level must be decorated with

⁷To avoid ambiguity, instead of using priority declarations, one can stratify the syntax of expressions by

the keyword `%nonassoc`, `%left`, or `%right`. This too influences the manner in which conflicts are resolved. For example, the fact that `PLUS` and `MINUS` are declared “left-associative” means that we wish to interpret the input fragment `3-2-1` as a subtraction whose first operand is `3-2` and whose second operand is `1`. Therefore, when `3-2` has been read and the next input symbol is `-`, reducing is preferred to shifting.

The declaration `%start <unit> file` indicates that the non-terminal symbol `file` is the *start symbol*, that is, the main entry point. The OCaml type `unit` appears because, for the sake of simplicity, we let this parser return a semantic value of type `unit`, as opposed to an abstract syntax tree. In other words, this parser is not quite a parser in the usual sense. Rather, it is a recognizer: either it succeeds (in which case it returns no additional information) or it fails.

The last section of Figure 2 is the context-free grammar. It defines the *non-terminal symbols* `file`, `declaration`, `binding`, and `expr`. These definitions reflect the description of the language \mathcal{L} that was given at the beginning of Section 2. In each right-hand side, the empty curly braces `{}` denote a trivial semantic action, which returns a `unit` value; this is consistent with our decision to *not* construct abstract syntax trees. In the definition of `file`, at line 19, we write `declaration*` as a short-hand for `list(declaration)`. The parameterized non-terminal symbol `list(X)`, which represents a possibly empty list of `X`'s, is defined in Menhir's standard library.

5 Handling syntax errors

When a parser constructed by Menhir detects a syntax error, by default, it just fails. It is then up to the programmer to construct an error message that explains where and why an error was encountered. The *location* of the failure can be obtained by querying the lexer, that is, by reading the fields `lex_curr_p` and `lex_start_p` of the “lexing buffer”, which the lexer maintains.⁸ In contrast, explaining the *reason* for the failure requires analyzing the parser's state—that is, its stack.

As explained earlier, a *stack* can be defined as a sequence of terminal and non-terminal symbols. A stack describes how the input fragment that has been read by the parser is understood at this time. For example, after our parser for the toy language \mathcal{L} has read the character sequence `let x = 21 * 2`, which the lexer transforms into the sequence of terminal symbols `LET IDENT EQUAL INT TIMES INT`, the parser's stack is:

```
LET IDENT EQUAL expr TIMES INT
```

In this stack, one can see that the first occurrence in the input of the terminal symbol `INT`, corresponding to `21`, has been reduced to `expr` already. Indeed, when `INT` appears on top of the stack and when the next input symbol is `TIMES`, this parser reduces `INT` to `expr`. The second occurrence in the input of the symbol `INT`, on the other hand, has not yet been reduced: it has just been shifted and appears at the top of the stack. Depending on the next input symbol, zero, one, or more reductions might now take place:⁹

declaring several non-terminal symbols that represent several levels of expressions. The grammars of the programming languages C [ISO11, Annex A.2] and Java [GJS⁺25, Chapter 15] offer examples of this technique. If desired, this stratification technique can be combined with the use of priority declarations. Avoiding ambiguity by manually transforming the grammar works well in the case of levels of expressions, but can also be quite unnatural and painful: for example, whereas resolving the famous “dangling else” ambiguity via a priority declaration is very easy, avoiding this ambiguity by transforming the grammar requires introducing a syntactic category of “stable expressions”, whose meaning is not affected when they are followed with an `else` keyword. This can require duplicating or parameterizing large parts of the grammar.

⁸<https://ocaml.org/manual/api/Lexing.html>

⁹The following two bullet points show that, depending on the lookahead symbol, two distinct reduction paths, of length 2 and of length 4, can be taken. In this example, the shorter path happens to be a prefix of the longer path. In more complex examples, two reduction paths can truly diverge, that is, can differ not only in their lengths but also in the reduction steps that they involve.

- For example, if the next input symbol is again `TIMES` then two successive reductions take place: first, `INT` is reduced to `expr`; second, `expr TIMES expr` is reduced to `expr`. After these two reductions, the stack is:

```
LET IDENT EQUAL expr
```

and, at this point, the symbol `TIMES` can be shifted.

- As another example, if the next input symbol is `LET` then four successive reductions take place: first, `INT` is reduced to `expr`; second, `expr TIMES expr` is reduced to `expr`; third, `IDENT EQUAL expr` is reduced to `binding`; last, `LET binding` is reduced to `declaration`. After these four reductions, the stack is:

```
declaration
```

and, at this point, the symbol `LET` can be shifted.

- If the next input symbol is illegal at this point then, regardless of which reductions the parser might take, we view these reductions as *spurious*. After the parser fails, we undo these reductions¹⁰ and come back to the stack that existed after the last legal symbol was shifted, that is, `LET IDENT EQUAL expr TIMES INT`. This is the stack that we submit to a case analysis by LRgrep.

In short, a stack is a summary of the input fragment that the parser has read. It shows how the parser has understood this input fragment, that is, how the parser has reduced this input fragment (a sequence of terminal symbols) to a mixed sequence of terminal and non-terminal symbols. The stack also determines what the parser expects or accepts next. This is enough information to craft a good explanation of the reason why the parser failed.¹¹ The LRgrep language lets us describe a mapping of stacks to syntax error messages in the form of a case analysis by pattern matching.

An LRgrep program for the toy language \mathcal{L} appears in Figure 3. This program is placed in a file named `errors.lrgrep`. It is compiled by LRgrep into an OCaml source file named `errors.ml`.

This LRgrep program begins with the line `rule error_message = parse error (file)`. This means that we are interested in analyzing the parser’s stack after it has attempted to recognize the start symbol `file` and detected a syntax error. After this program has been compiled by LRgrep, the file `errors.ml` contains an OCaml function named `error_message` whose type is as follows:

```
val error_message :
  Parser.MenhirInterpreter.env -> (* the parser's stack *)
  Parser.token * Lexing.position * Lexing.position -> (* the illegal symbol *)
  string option (* an error message *)
```

The function `error_message` expects two parameters, namely (1) the stack that we wish to analyze and (2) the illegal terminal symbol that was just read by the parser, together with its start and end positions.¹² This function returns a string, that is, a syntax error message.¹³ This is our decision: in Figure 3, the fragments of OCaml code delimited by curly braces have type `string`. If desired, we could have chosen some other result type.

The case analysis in Figure 3 is structured as a series of *clauses*, where each clause has the form:

$$| \text{ pattern } \{ \text{OCaml code fragment} \}$$

¹⁰This is made easy by the function `loop_handle_undo` in Menhir’s incremental parsing API.

¹¹Once again, we acknowledge that “a good explanation of the reason why the parser failed” is not necessarily “a good explanation of the programmer’s mistake” or “a good suggestion of how to repair the mistake”.

¹²This second parameter is needed because LRgrep lets us examine not only the stack, but also the illegal lookahead symbol. A pattern can be accompanied with a side condition of the form `@z`, where `z` is a terminal symbol, which requires the lookahead symbol to be `z`. This construct is not used in Figure 3.

¹³In fact, it returns an optional string, that is, a value of type `string option`. The value `None` is returned when no pattern matches the parser’s stack. If the static coverage check has succeeded (Section 5.8) then this should never happen.

```

1 rule error_message = parse error (file)
2
3 | / . file
4   { "A declaration is now expected." }
5
6 | l=LET
7   { read (start decl) $positions(l) ^
8     "An identifier is now expected." }
9
10 | l=LET; i=IDENT
11   { read (start decl) ($startpos(l), $endpos(i)) ^
12     "An equals sign '=' is now expected." }
13
14 | l=LET; IDENT; e=EQUAL
15   { read (start decl) ($startpos(l), $endpos(e)) ^
16     "An expression is now expected." }
17
18 | d=[declaration]
19   { read decl $positions(d) ^
20     "If this declaration is complete, then" ^/^
21     "another declaration (or the end of the file) is now expected." }
22
23 | [declaration]; s=SEMI; l=LET; b=[binding]
24   { read ldecl ($startpos(l), $endpos(b)) ^
25     "The keyword 'in' is now expected." ^/^
26     sprintf "Suggestion: deleting the semicolon that precedes this declaration (%s)"
27       (range $positions(s)) ^/^
28     "would allow it to be interpreted as a global declaration." }
29
30 | l=LET; b=[binding]
31   { read ldecl ($startpos(l), $endpos(b)) ^
32     "The keyword 'in' is now expected." }
33
34 | l=LET; b=binding; i=IN
35   { read (ldecl ++ leti) ($startpos(l), $endpos(i)) ^
36     "An expression is now expected." }
37
38 | e=expr; o=PLUS
39 | e=expr; o=MINUS
40 | e=expr; o=DIV
41 | e=expr; o=TIMES
42 | e=expr; o=SEMI
43   { read (expr ++ binop) ($startpos(e), $endpos(o)) ^
44     "An expression is now expected." }
45
46 | l=LPAREN / .* . expr .*
47   { read lparen $positions(l) ^
48     "An expression is now expected." }
49
50 | l=LPAREN; e=[expr]
51   { read (lparen ++ expr) ($startpos(l), $endpos(e)) ^
52     "If this expression is complete," ^/^
53     "a closing parenthesis is now expected." }

```

Figure 3. The syntax error description file `errors.lrgrep`

<pre>garbage</pre>
<pre>File "input01.in" (1:0-7): Syntax error. A declaration is now expected.</pre>
<pre>let let x = 0</pre>
<pre>File "input02.in" (2:0-3): Syntax error. The start of a declaration has been read (1:0-3): let An identifier is now expected.</pre>
<pre>let x + 2 = 2</pre>
<pre>File "input03.in" (1:6-7): Syntax error. The start of a declaration has been read (1:0-5): let x An equals sign '=' is now expected.</pre>
<pre>let x =) let z = 0</pre>
<pre>File "input04.in" (1:8-9): Syntax error. The start of a declaration has been read (1:0-7): let x = An expression is now expected.</pre>
<pre>let a = let x = 1+2- in 0</pre>
<pre>File "input05.in" (2:15-17): Syntax error. An expression and a binary operator have been read (2:10-14): 1+2- An expression is now expected.</pre>
<pre>let x = 1+(+)</pre>
<pre>File "input06.in" (1:11-12): Syntax error. An opening parenthesis has been read (1:10-11): (An expression is now expected.</pre>

Figure 4. Examples of illegal inputs and syntax error messages

```
let x = 1+2)3
let y = 0
```

```
File "input07.in" (1:11-12):
Syntax error.
A declaration has been read (1:0-11):
  let x = 1+2
If this declaration is complete, then
another declaration (or the end of the file) is now expected.
```

```
let x = 1+2-
let y = 0
```

```
File "input08.in" (3:0-0):
Syntax error.
A local declaration has been read (2:0-9):
  let y = 0
The keyword 'in' is now expected.
```

```
let x = (1+2
let y = 0
```

```
File "input09.in" (2:0-3):
Syntax error.
An opening parenthesis and an expression have been read (1:8-12):
  (1+2
If this expression is complete,
a closing parenthesis is now expected.
```

Figure 5. Examples of illegal inputs and syntax error messages (continued)

The semantics of this pattern-matching construct is rather straightforward. Roughly speaking, a pattern either matches (recognizes) or rejects a suffix of a stack.¹⁴ Each pattern is tried in turn until a pattern that matches a suffix of the stack is found. Then, the corresponding fragment of OCaml code is executed, and its result is returned.

This is analogous to pattern-matching in OCaml and in `ocamllex`. The main unusual aspect is that pattern matching is not applied to a sequence of characters, or to a sequence of terminal symbols, but to an LR parser’s stack, which can be viewed as a sequence of terminal and non-terminal symbols. Furthermore, LRgrep’s *patterns* include several original constructs whose meaning is specific to the setting of analyzing the stack of an LR parser.

Let us now look in detail at the program in Figure 3. In the following, we devote one subsection to each clause in Figure 3.

5.1 A declaration is now expected

The pattern `/ . file` (Figure 3, line 3) recognizes a situation where the parser is willing to read an input fragment that conforms to the non-terminal symbol `file`. This is an example of a filter pattern */ item*, which we explain in greater detail in Section 6. If desired, one can write this pattern also under the parenthesized form `(/ . file)`, which is slightly heavier, but makes it easier to see that `. file` is an item.

An *item*, also known as a LR(0) item, is a production into whose right-hand side a dot “.” has been inserted. If the length of the right-hand side is p then there are $p + 1$ places where a dot can be inserted. For example, based on the production `expr: expr PLUS expr`, whose right-hand side has length 3, one can construct 4 items:

¹⁴An accurate formal semantics is given in Section 6.

```

expr: . expr PLUS expr
expr: expr . PLUS expr
expr: expr PLUS . expr
expr: expr PLUS expr .

```

Unless this is ambiguous, in an item, LRgrep allows omitting the left-hand side. Thus, for example, the first item above can also be written `. expr PLUS expr`. The item `. file` is the abbreviated form of the item `file': . file`, where `file'` is a start symbol that Menhir silently creates. If one thinks of a grammar intuitively as a program, then an item is a program point: the position of the dot indicates what symbols have been recently recognized and what symbols the parser expects to recognize next, in order to reduce a certain production. For example, the item `expr: expr PLUS . expr` intuitively means that the parser has recognized an expression followed with the symbol `PLUS` and now expects to recognize another expression, so as to be able to declare that it has recognized an addition.

A filter pattern / *item* imposes a constraint on the stack: it requires the item *item* to be *viable* with respect to this stack. Let us define viability by example; a formal definition is given by the rule `PAT-FILTER` in Section 6. The item `binding: IDENT EQUAL . expr` is viable with respect to a stack α if and only if:

1. α ends with the symbols `IDENT EQUAL`, that is, $\alpha = \alpha' \text{ IDENT EQUAL}$ for some α' ;
2. the stack $\alpha \text{ expr}$ is well-formed; and
3. the stack $\alpha' \text{ binding}$ is well-formed.

In fact, condition (3) above implies condition (2). A stack α is *well-formed* if there exists an input fragment w such that, starting in a state where the stack is α and the remaining input is w , the parser reaches a successful state where the stack is just S and the remaining input is empty.

If the item *item* is viable with respect to a stack, then the filter pattern / *item* matches (consumes) an empty suffix of this stack; otherwise, it rejects this stack.¹⁵

Let us come back to the example of the filter pattern / `. file`. Because `file` is the grammar's start symbol, the item `. file` is viable with respect to only one stack, namely, the empty stack. Therefore, the pattern / `. file` actually match just the empty stack. In other words, it recognizes just the situation where the parser has not read anything yet. If the parser fails in this situation, then the input must begin with an illegal terminal symbol.

In this situation, we choose to produce the message "A declaration is now expected.". One could produce a different message: for example, because a declaration must begin with the keyword `let`, one might wish to say that the keyword `let` is now expected. As a general principle, we prefer to say that the parser expects a declaration (a non-terminal symbol), as opposed to the keyword `let` (a terminal symbol), because this description is higher-level and seems more likely to remain valid as the language evolves: in the future, the language might be extended with new forms of declarations, which do not begin with the keyword `let`.

By convention, before displaying a syntax error message, we always display the name of the input file, the position in the input file where the parser failed, and the fixed string "Syntax error.". The complete message shown to the user appears in Figure 4 as part of the first example, named `input01`.

5.2 An identifier is now expected

The pattern `l=LET` (Figure 3, line 6) matches the stack suffix `LET`. In other words, it recognizes every situation where the parser has just read the keyword `let`.

¹⁵Because of this feature, a pattern can *constrain* a large part of the stack even though it *consumes* only a smaller part of it. More examples of this slightly subtle phenomenon appear at the end of Section 5.2, and a formal semantics of patterns is given in Section 6.

One could write this pattern under the simpler form `LET`. By writing `l=LET`, we introduce `l` as a name for this fragment of the stack. This lets us use the special variable `$positions(l)` inside the OCaml code that follows (line 7) to refer to the start and end positions in the input file of this `LET` symbol.¹⁶ Binding a name in this way is known as a *capture*.

In general, we believe that a good syntax error message should convey two pieces of information, namely (1) what has been recently read and (2) what is expected (or accepted) next. Here, we are in a situation where the keyword `let` has just been read: (1) this keyword is understood as the beginning of a (global or local) declaration, and (2) an identifier is expected to appear next. We construct a message in two lines of OCaml code: on line 7, we indicate that the start of a declaration has been read;¹⁷ on line 8, we indicate that an identifier is now expected. The result is shown in example `input02` in Figure 4.

In this example, we choose to indicate that an identifier (a terminal symbol) is expected, whereas we could have indicated that a binding (a non-terminal symbol) is expected. This is an exception to the general principle that we have stated earlier (Section 5.1). The reason for this decision is that we do not expect an end user of our parser to be aware of the existence of the non-terminal symbol `binding`. We do expect an end user to know what is meant by a “declaration” and by an “expression”, because these are syntactic categories whose importance should be fundamental in the mind of a user of the language \mathcal{L} . On the other hand, `binding` is possibly a syntactic category that we have introduced to make the grammar more concise and of which an end user is not aware.

Here, it was clear to us that, after a `let` keyword has been read, (a) this keyword must eventually become the start of a declaration, (b) a binding is expected next, and (c) a binding must begin with an identifier, therefore an identifier is expected next. These facts were obvious to us because the grammar in Figure 2 is simple enough to be fully grasped at once by our human minds. If the grammar was more complex, so that these facts were not obvious or not true, then instead of the pattern `l=LET`, we could decide to use a more specific pattern, such as one of the following four composite patterns:

```
l=LET / declaration: LET . binding
l=LET / LET . binding
l=LET / binding: . IDENT EQUAL expr
l=LET / .* . IDENT _*
```

In each of these composite patterns, the pattern `l=LET` is followed with a filter pattern */ item*. Thus, each of these patterns requires the symbol `LET` to appear at the top of the stack and (via a filter pattern) imposes a constraint on the stack as a whole. The four examples above impose various constraints on the stack:

- The first item matches a stack where `LET` has been read, `binding` is accepted next, and the sequence `LET binding` can be reduced to `declaration`. This is a non-trivial constraint: it accepts a stack where the `LET` symbol that has just been read is possibly the start of a global declaration, and rejects a stack where this symbol must definitely be the start of a local declaration. This constraint is stronger than is needed here.
- The second item matches a stack where `LET` has been read and `binding` is accepted next, without requiring that the sequence `LET binding` be reducible to a global declaration.
- The third item matches a stack where the sequence `IDENT EQUAL expr` is accepted next and this sequence can be reduced to `binding`.
- The fourth item matches every stack where `IDENT` is accepted next. LRgrep lets us write `.* . IDENT _*` to describe the family of all items where the dot precedes the

¹⁶LRgrep views `$positions(l)` as a short-hand for the pair `($startpos(l), $endpos(l))`. The special variables `$startpos(l)` and `$endpos(l)` can also be used in isolation.

¹⁷For brevity, we omit the definitions of the auxiliary functions and constants `read`, `start`, `decl`, and so on. The reader is referred to the file `src/errors.lrgrep` in our [repository](#).

symbol `IDENT`. Thus, in general, the syntax of filter patterns is `/ items` where *items* is a family of items.

In summary, a filter pattern `/ items` selects just the contexts where a certain continuation is accepted and can be interpreted (reduced) in a certain way.

5.3 An equals sign is now expected

The pattern `l=LET; i=IDENT` (Figure 3, line 10) describes the situation where the parser has read the terminal symbols `LET` and `IDENT`, that is, the keyword `let`, followed with an identifier. We handle this situation in the same way as earlier: we indicate that the start of a declaration has been read and that the terminal symbol `EQUAL` is now expected. This is illustrated in example `input03` in Figure 4.

5.4 An expression is now expected

The pattern `l=LET; i=IDENT; e=EQUAL` (Figure 3, line 14) describes the situation where the parser has read the terminal symbols `LET`, `IDENT`, and `EQUAL`. We indicate that the start of a declaration has been read and that an expression is now expected. Example `input04` in Figure 4 illustrates this situation and the syntax error message that is displayed.

There are other situations where an expression is expected. These include the situation where an expression followed with a binary operator has been read (Figure 3, line 42) and the situation where an opening parenthesis has been read (Figure 3, line 46). These situations are illustrated by examples `input05` and `input06` in Figure 4.

We choose to distinguish these three situations because in each of them we give a different account of what has been recently read. If we were willing to omit this information and indicate merely that an expression is now expected then we could handle all three situations at once by using the filter pattern `/ .* . expr .*`. This pattern matches every situation where an expression is an acceptable continuation, regardless of what has been read earlier.

5.5 Another declaration is now expected

The pattern `d=[declaration]` (Figure 3, line 18) is a reduce pattern, an original construct in LRgrep's domain-specific language of patterns. It recognizes the situations where, *up to a number of reductions*, one can say that a global declaration has been read. Whereas the pattern `declaration` matches literally just the stack segment `declaration`, the pattern `[declaration]` matches all stack segments that can be reduced to `declaration`. This feature provides crucial extra flexibility: instead of examining just one stack, it allows us to virtually examine *all of the stacks that can be derived from this stack* by repeated reductions.

Such a situation is illustrated by example `input07` in Figure 5. There, a stray closing parenthesis at the end of line 1 causes a failure. At this point, the input fragment that has been read by the parser is `let x=1+2`, and the parser's stack is `LET IDENT EQUAL expr PLUS INT`. This stack does not end with the symbol `declaration`, therefore is not matched by the pattern `declaration`. Yet, it *is* matched by the pattern `[declaration]`, because, as per the grammar in Figure 2, the following sequence of reductions is possible:

```
LET IDENT EQUAL expr PLUS INT
LET IDENT EQUAL expr PLUS expr
LET IDENT EQUAL expr
LET binding
declaration
```

The reader might wonder: if this sequence of reductions is possible, why didn't the parser perform these reductions before failing? This is a subtle point. In general, after an LR parser

has read a certain prefix of the input and constructed a certain stack, there may exist *several* ways of reducing this stack, that is, several ways of understanding what has been read so far. An LR(1) parser is deterministic: to choose between several reduction opportunities, it consults the lookahead symbol. Technically, it uses the current state of the LR automaton and the lookahead symbol as indices to look up a two-dimensional *action table*. In the above situation, where the stack is `LET IDENT EQUAL expr PLUS INT` and the lookahead symbol is the closing parenthesis `RPAREN`, the action table does not permit any reduction: it dictates a failure.

In contrast, in our effort to analyze the situation and produce a good syntax error message, we do not wish to tie our hands and restrict our attention to the single action that is normally permitted when the lookahead symbol is `RPAREN`. After all, considering that the parser has just failed, this symbol is somewhat likely to be incorrect. Therefore, in our analysis of the situation, we disregard the lookahead symbol. While testing whether the current stack is matched by the pattern `[declaration]`, we consider *all* of the reduction paths permitted by the action table for *all lookahead symbols*. The sequence of reductions that is shown above is permitted by the action table when the lookahead symbol is `LET` or `EOF`. Therefore, we find that the stack `LET IDENT EQUAL expr PLUS INT` is matched by the pattern `[declaration]`.

In this situation, we choose to report that a declaration has been read (Figure 3, line 18). Furthermore, we choose to indicate that either another declaration or the end of the file is now expected. Indeed, both are legal continuations. One might prefer to lie by omission and mention fewer legal continuations: for example, one could indicate that another declaration is expected, thereby omitting the possibility of ending the file here. We believe that this type of white lie is often good practice.

Because LRgrep lets us extract the source code positions `$startpos(l)` and `$endpos(b)` out of the parser's stack, we are able to explicitly show which declaration has been read: in example `input07` in Figure 5, it is `let x=1+2`. We believe that this can help the user determine whether the parser has correctly interpreted or misinterpreted the input fragment that it has read so far.

Instead of unconditionally asserting that another declaration is now expected, we choose to say: *if this declaration is complete* then another declaration is now expected. Indeed, in example `input07` in Figure 5, we cannot be certain that the declaration `let x=1+2` is complete; perhaps the user intended the declaration to be `let x=1+2-3`, but they mistyped. It can be difficult to decide what is the best way of phrasing an error message. On the one hand, an error message should be simple, so one is tempted to tell a white lie and oversimplify the situation: in this case, by unconditionally claiming that another declaration is expected. On the other hand, one should avoid misrepresenting what the parser has definitely understood. In this case, the parser *could* understand `let x=1+2` as a declaration if the next input symbol was changed to `LET` or `EOF`, but could also view `1+2` as an incomplete expression if the next input symbol was changed to a binary operator such as `PLUS` or `TIMES`.¹⁸

5.6 The keyword ‘in’ is now expected

The pattern `l=LET; b=[binding]` (Figure 3, line 30) recognizes a situation where the keyword `let`, followed with an input fragment that can be understood as a binding, have been read. Furthermore, because the clauses in Figure 3 are examined in order, if this pattern matches the parser's stack, then one can deduce that the earlier pattern `d=[declaration]` did not match it. This means that, even though the stack ends with the sequence `LET binding`, reducing this sequence to `declaration` is not permitted. In other words, the sequence `LET binding` must form the beginning of a local declaration.

If we wanted to explicitly indicate this, instead of the pattern `l=LET; b=[binding]`, we could

¹⁸When the lookahead symbol is `TIMES`, this parser cannot even reduce `expr PLUS expr` to `expr`. This is not permitted by its action table. When the lookahead symbol is any binary operator, it cannot reduce `LET IDENT EQUAL expr` to `declaration`.

use the more complex pattern `l=LET; b=[binding /expr: LET binding . IN expr]`, which makes it clear that, after reducing a suffix of the stack to `binding`, the parser must be in a state where `IN expr` is a legal continuation.

In such a situation, we can safely indicate that a *local* declaration has been read and that the keyword `in` is now expected. This is illustrated by example `input08` in Figure 5. There, the stray `'` sign at the end of the first line causes `let y = 0` on the second line to be interpreted as a local declaration; the parser fails when it reaches the end of the file. This example is very similar to the one that was discussed in detail in Section 3.

Although our claim that “the keyword `in` is now expected” seems unconditional, the correct fix, in this case, is (likely) not to insert the keyword `in` at the end of the file; instead, it is to delete the stray `'` sign at the end of the first line. The user can tell that the parser has been led astray because the error message claims that `let y = 0` is a local declaration, whereas the user (likely) knows that it should be a global declaration. We believe that it is important to explain both what the parser has understood and what the parser expects next, as opposed to just the latter, because the user can then more easily recognize that the parser has misinterpreted what it has read—in other words, that the true mistake lies before the point where the parser has failed.

The pattern `[declaration]; s=SEMI; l=LET; b=[binding]` (Figure 3, line 23) recognizes a special case of the previous situation. Our intent is to recognize precisely the problematic situation, caused by a stray semicolon, that was discussed in Section 3. We wish to single out this specific situation because our expertise with the language (in this paper, the language \mathcal{L} ; in real life, the language OCaml) leads us to believe that this situation frequently arises and that the true origin of the failure can be difficult for the user to diagnose. We choose to not expend similar effort on other special cases such as (say) the case of a stray `'` sign, illustrated by example `input08` in Figure 5.

The pattern `[declaration]; s=SEMI; l=LET; b=[binding]` is somewhat unusual in that it uses *two* reduce patterns. It matches a sequence of four input fragments:

1. an input fragment that *could* be understood as a global declaration, in light of the context where it appears, and without regard for the semicolon that follows it;
2. a semicolon;
3. the keyword `let`;
4. an input fragment that could be reduced to `binding`, without regard for the illegal input symbol that has caused the parser to fail.

In this situation, we (somewhat arbitrarily) decide that this semicolon is the likely cause of the problem. Therefore we suggest deleting this semicolon, and we explain that this would allow the preceding input fragment to be recognized as a global declaration. The bottom part of Figure 1 shows the error message that we produce in this case.

5.7 A closing parenthesis is now expected

The pattern `l=LPAREN; e=[expr]` (Figure 3, line 50) recognizes a situation where an opening parenthesis, followed with an input fragment which (up to reductions) can be understood as an expression, have been read. In this case, assuming that this input fragment should indeed be understood as an expression, and assuming that this expression is complete, a closing parenthesis is now expected. This situation is illustrated by example `input09` in Figure 5.

5.8 Coverage and redundancy

At this point, we are done: the patterns in Figure 3 cover every possible situation in which the parser can fail. We know this because LRgrep performs a coverage check. To do so, the LRgrep compiler computes all of the ways in which the parser can fail. More accurately, it computes a finite description of the set of all pairs (α, z) where α is a well-formed stack, z is a terminal symbol, the combination of the stack α and the lookahead symbol z causes the

parser to fail, and the configuration (α, z) is reachable. Then, the LRgrep compiler checks that every such pair is matched by at least one pattern. In other words, it checks that the case analysis is exhaustive.

The side condition that (α, z) be reachable matters only in the case where the automaton has been altered due to user-provided priority declarations. We do not discuss this condition further, but note that our reachability algorithm [BP21] is used to compute it.

LRgrep also offers a redundancy check: if a clause matches no configurations, because all of the configurations that it recognizes are matched by earlier clauses already, then the LRgrep compiler emits a warning.

5.9 Workflow

Thanks to the static coverage and redundancy checks performed by the LRgrep compiler, we believe that a user of LRgrep can enjoy the familiar comfort of pattern matching. These facilities encourage an incremental style of programming, where one gradually discovers the landscape of the syntax error situations. One typically begins with no patterns at all, or perhaps just a few patterns that one can easily think of. Then, the LRgrep compiler points out which error situations are not covered, and which situations are covered by multiple patterns. One can then improve and simplify the existing clauses, add new clauses that cover more error situations, and repeat the process until both the coverage check and the redundancy check are successful. In the case of the toy language \mathcal{L} , the LRgrep program in Figure 3 reflects the final result of this incremental process.

6 Patterns

We now give a brief description of the syntax and semantics of patterns. A reader who is looking for an informal understanding of LRgrep can skip this section, but may be interested in [the more complete reference guide](#) that is available online. A reader who would like to see a formal definition of the meaning of patterns, but who is not prepared to read the first author’s dissertation [Bou24], should find this section useful.

We assume that a grammar, such as the one shown in Figure 2, has been fixed. For simplicity, we assume that the grammar contains no priority declarations and that the parser generator reports no conflicts, which guarantees that the grammar is in the class LR(1).¹⁹

6.1 Syntax

Let A, B denote non-terminal symbols. A *symbol* x is a terminal or non-terminal symbol. A *sentential form* α, β, γ is a sequence of symbols. A *production* takes the form $A \rightarrow \alpha$, and we write just $A \rightarrow \alpha$ to assert that $A \rightarrow \alpha$ is a production of the grammar. An *item* takes the form $A \rightarrow \beta \cdot \gamma$, where $A \rightarrow \beta\gamma$ is a production: in other words, an item is a distinguished position, denoted by a dot ‘ \cdot ’, inside the right-hand side of a production.

¹⁹When this assumption does not hold, the parser generator must perform “conflict resolution”, that is, forcefully remove certain shift and reduce actions, so as to make the LR automaton deterministic. Then, the semantics of patterns must be based on the more restricted behavior of the automaton, as opposed to the more lenient grammar. From the point of view of an end user, this subtle distinction is essentially invisible: the LRgrep language is unaffected, and the LRgrep compiler just “does the right thing”.

A slightly simplified syntax of *patterns* is as follows:

$p ::= \epsilon$	—	empty
x	—	single symbol
$p_1; p_2$	—	concatenation
$p_1 \mid p_2$	—	alternative
p^*	—	repetition
$(/item)$	—	filter
$[p]$	—	reduction

Reduce patterns cannot be nested. The features *not* described in this simplified syntax include captures, conditions bearing on the lookahead symbol (Footnote 12), the possibility of using wildcards `_` and `_*` inside patterns and inside items, as well as longest-match variants of the repetition and reduction constructs.

6.2 Semantics

A stack is a sequence of symbols, that is, a sentential form α . However, not every sentential form is a well-formed stack. For example, our parser for the toy language \mathcal{L} can construct the stack `LET IDENT`, but will never construct a stack that begins with `LET PLUS` or a stack that begins with `declaration LET binding IN`. When α is a well-formed stack, we write $\alpha \in VP$. The set VP is so named because a stack is well-formed if and only if it is a *viable prefix* of a *rightmost sentential form*²⁰ [Bou24, Definition 7].

The semantics of patterns takes the form of a judgment $(\alpha)\beta \in p$, which means that the pattern p *consumes* a suffix β of the stack $\alpha\beta$. Although the prefix α is not consumed, it is nevertheless possibly inspected and constrained by the pattern p : in other words, $(\alpha)\beta \in p$ does not imply $(\alpha')\beta \in p$.

This judgment is inductively defined by the following rules [Bou24, §3.2.1]. The most unusual rules are the last two rules, which concern LRgrep’s domain-specific constructs. **PAT-FILTER** states that a filter pattern $(/A \rightarrow \beta \cdot \gamma)$ consumes an empty stack suffix, but requires the stack to have the form $\alpha\beta$ where αA is a well-formed stack.²¹ In other words, this pattern requires the stack to end with β and to tolerate the actions of reading γ and reducing $\beta\gamma$ to A . **PAT-REDUCE** states that a reduce pattern $[p]$ consumes a stack suffix β if (in the presence of the unconsumed prefix α) β can be reduced to γ and p consumes γ .

PAT-EMPTY $(\alpha)\epsilon \in \epsilon$	PAT-SYMBOL $(\alpha)x \in x$	PAT-SEQUENCE $\frac{(\alpha)\beta \in p_1 \quad (\alpha\beta)\gamma \in p_2}{(\alpha)\beta\gamma \in p_1; p_2}$	PAT-ALT-LEFT $\frac{(\alpha)\beta \in p_1}{(\alpha)\beta \in p_1 \mid p_2}$
PAT-ALT-RIGHT $\frac{(\alpha)\beta \in p_2}{(\alpha)\beta \in p_1 \mid p_2}$	PAT-STAR $\frac{(\alpha)\beta \in \epsilon \mid (p; p^*)}{(\alpha)\beta \in p^*}$	PAT-FILTER $\frac{A \rightarrow \beta\gamma \quad \alpha A \in VP}{(\alpha\beta)\epsilon \in (/A \rightarrow \beta \cdot \gamma)}$	PAT-REDUCE $\frac{(\alpha)\beta \downarrow \gamma \quad (\alpha)\gamma \in p}{(\alpha)\beta \in [p]}$

The first premise of **PAT-REDUCE** involves the auxiliary judgment $(\alpha)\beta \downarrow \gamma$, which is known as the *reduction relation* [Bou24, §2.2.2]. This judgment means that a stack $\alpha\beta$ can be transformed to $\alpha\gamma$ via a sequence of zero or more reduction steps, which do not touch α , but may require its presence. Thus, $(\alpha)\beta \downarrow \gamma$ can be read as: “in the context α , the stack suffix β can be reduced to γ ”. This judgment is defined by the following rules:

RED-BASE $\frac{\alpha \in VP}{(\alpha)\epsilon \downarrow \epsilon}$	RED-OUTER $\frac{(\alpha\beta)\gamma \downarrow \gamma' \quad A \rightarrow \beta\gamma' \quad \beta \neq \epsilon \quad \alpha A \in VP}{(\alpha)\beta\gamma \downarrow A}$	RED-INNER $\frac{(\alpha)\beta \downarrow \alpha'\gamma' \quad B \rightarrow \gamma' \quad \alpha\alpha'B \in VP}{(\alpha)\beta \downarrow \alpha'B}$
--	---	--

²⁰Here is a definition of the set VP . $\alpha \in VP$ holds iff there exists a sequence of terminal symbols w such that αw can be derived from the start symbol S via a rightmost derivation. In other words, α is a well-formed stack if there exists an input fragment w such that, beginning in a state where the stack is α and the remaining input is w , an LR parser reaches a successful state where the stack is just S and the remaining input is empty. The set VP is a regular language [Bou24, §2.2.1].

²¹This implies that $\alpha\beta\gamma$ is well-formed too.

In these three rules, the parameter α is used in the well-formedness checks, that is, in the last premise of each rule. These checks ensure that a reduction step is considered only if it is permitted by the context. Technically, they ensure that reduction results in a well-formed stack: that is, $(\alpha)\beta \downarrow \gamma$ implies $\alpha\gamma \in VP$.

7 Conclusion

In this work, we have focused on the problem of diagnosing a single syntax error, that is, producing a good human-readable explanation of the reason why an LR parser failed. We do not attempt to automatically construct an explanation message: instead, we rely on a human expert (usually, the author of the parser) to construct a mapping of error situations to error messages. Our main contribution is to equip this expert with a domain-specific language, LRgrep, and its compiler. Together, the language and the tool help this expert discover the landscape of the syntax error situations and describe this landscape using *patterns*. We believe that patterns are fairly abstract and, with some practice, can seem intuitive. In particular, they are phrased in terms of the terminal symbols, non-terminal symbols, and productions of the grammar, and require no knowledge of the underlying LR automaton. The tool checks that the patterns provided by the user are exhaustive and irredundant: that is, every error situation is covered by at least one pattern, and every pattern contributes to covering at least one error situation.

Our approach has limitations. In particular, although we have proposed a few informal guidelines for writing error messages, we do not have a systematic, automatable methodology for constructing “good” error messages. Besides, we currently do not have any way of ensuring that the error message that the user constructs is consistent with the pattern that the user has written.

We view our approach as an improvement over Jeffery’s approach [Jef03], which consists in asking a human expert, or a crowd of non-expert users, to produce a mapping of illegal input sentences to error messages. This mapping is then transformed into a mapping of the states of the LR automaton to error messages. Jeffery’s approach has been implemented in `gc`, a `yacc`-based parser for the language Go [Cox10]. Systematic support for this approach, including efficient coverage and redundancy checks, has been implemented by the authors in Menhir [Pot16, BP21] and has been exploited in several production-grade parsers, including CompCert C [Ler25], Catala [MCP21], and Stan 3 [Tal19]. In comparison with this approach, LRgrep improves in several important ways. Whereas a user of Jeffery’s approach manipulates illegal input sentences which (in an implicit and somewhat fragile way) describe states of the LR automaton, a user of LRgrep works with high-level patterns that describe sets of stacks. Whereas Menhir’s `%on_error_reduce` directive forces the user to statically fix a deterministic reduction strategy, LRgrep’s reduce patterns simulate all of the ways in which the stack could be reduced if the illegal symbol that has been encountered was replaced with some other symbol. Finally, in comparison with Jeffery’s approach, we expect LRgrep programs to be much easier to maintain in the long run as a language and its parser evolve.

The first author [Bou24, Chapter 8] has applied LRgrep to two real-world programming languages, Catala and Elm. He finds that LRgrep allows producing the same error messages as the original Catala and Elm parsers, which respectively rely upon Jeffery’s approach (Catala) and hand-written recursive-descent parser combinators (Elm). In summary, while arguably enabling a more declarative programming style, the use of LRgrep does not lead to a decrease in the quality of error messages.

Our work aims to make it easy for the human expert to obtain information about the context of the failure, thereby enabling her to construct a message that explains why the parser failed. We do not define what form a “good” error message should take: on this topic, we refer the reader to other authors [NPM08, PPM⁺17, WK17, BFMHP18, BDP⁺19, DPB⁺21].

Although our technique sometimes allows proposing a suggested repair, as illustrated

by the example in Figure 1, this is not its main intended use. In general, our technique does not allow suggesting the best or most likely way of repairing the input. Furthermore, we do not attempt to continue parsing beyond the first failure, which would be desirable in an IDE, where one wishes to detect and highlight multiple syntax errors and to construct an approximate parse tree for the entire document, in spite of the syntax errors that it may contain. Merlin [BRS18], a language server for OCaml, contains an error-resilient LR parser for OCaml, which exploits indentation as part of its heuristic error-recovery algorithm. Unfortunately, this algorithm has never been described or made available in a reusable form. In a somewhat similar spirit, de Jonge et al. [dJKVS12] describe an error recovery technique for SGLR parsers, which (we recall) are significantly more expensive than deterministic parsers. Their technique involves extending grammars with additional “recovery” productions and is capable of exploiting indentation. As of 1998, Degano and Priami [DP98] offer a survey of error recovery techniques for LR parsers; it has been followed by more recent work in this area [CPRT02, KY10, DT20]. Outside of the realm of LR parsing, Medeiros and Mascarenhas [MM18] propose an error recovery technique for PEG parsers and evaluate it in a parser for Lua. There has also been much recent interest in exploiting machine learning [ALD23] and large language models [SCP⁺18, JSG⁺23, LHS⁺23] to produce better error messages and suggest repairs. Some researchers have warned that these approaches do not necessarily lead to more effective error messages [SB24].

The first author is currently working on improving the reports that are produced by the LRgrep compiler when the coverage check or the redundancy check detects a problem. Furthermore, a large-scale application of LRgrep to the OCaml and OCaml parsers is in the works.

References

- [ALD23] Toufique Ahmed, Noah Rose Ledesma, and Premkumar Devanbu. [SynShine: Improved fixing of syntax errors](#). *IEEE Transactions on Software Engineering*, 49(4):2169–2181, 2023.
- [BDP⁺19] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. [Compiler error messages considered unhelpful: The landscape of text-based programming error message research](#). In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, pages 177–210, July 2019.
- [BFMHP18] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. [How should compilers explain problems to developers?](#) In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 633–643, 2018.
- [Bmc25] Max Brunsfeld and many contributors. [Tree-sitter](#), December 2025.
- [Bou24] Frédéric Bour. [LRGrep: Selecting Error Messages for LR Parsers](#). PhD thesis, Université Paris Cité, December 2024.
- [BP21] Frédéric Bour and François Pottier. [Faster reachability analysis for LR\(1\) parsers](#). In *Software Language Engineering*, pages 113–125, October 2021.
- [BRS18] Frédéric Bour, Thomas Refis, and Gabriel Scherer. [Merlin: a language server for OCaml \(experience report\)](#). *Proceedings of the ACM on Programming Languages*, 2(ICFP):103:1–103:15, 2018.
- [Cox10] Russ Cox. [Generating good syntax errors](#). Blog post, January 2010.

- [CPRT02] Rafael Corchuelo, José A. Pérez, Antonio Ruiz, and Miguel Toro. [Repairing syntax errors in LR parsers](#). *ACM Transactions on Programming Languages and Systems*, 24(6):698–710, November 2002.
- [dJKVS12] Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, and Emma Söderberg. [Natural and flexible error recovery for generated modular language environments](#). *ACM Transactions on Programming Languages and Systems*, 34(4):15:1–15:50, 2012.
- [DP98] Pierpaolo Degano and Corrado Priami. [LR techniques for handling syntax errors](#). *Computer Languages*, 24(2):73–98, 1998.
- [DPB⁺21] Paul Denny, James Prather, Brett A. Becker, Catherine Mooney, John Homer, Zachary C. Albrecht, and Garrett B. Powell. [On designing programming error messages for novices: Readability and its constituent factors](#). In *Human Factors in Computing Systems*, pages 55:1–55:15, May 2021.
- [DS21] Charles Donnelly and Richard Stallman. *Bison*, September 2021.
- [DT20] Lukas Diekmann and Laurence Tratt. [Don’t panic! better, fewer, syntax errors for LR parsers](#). In *European Conference on Object-Oriented Programming (ECOOP)*, volume 166 of *Leibniz International Proceedings in Informatics*, pages 6:1–6:32, November 2020.
- [Eat21] Phil Eaton. [Parser generators vs. handwritten parsers: surveying major language implementations in 2021](#). Blog post, August 2021.
- [GJ08] Dick Grune and Criel J. H. Jacobs. *Parsing techniques: a practical guide, second edition*. Monographs in computer science. Springer, 2008.
- [GJS⁺25] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. [The Java language specification, Java SE 24 edition](#), February 2025.
- [ISO11] ISO. [ISO/IEC 9899:2011 – programming languages – C](#), 2011.
- [Jef03] Clinton L. Jeffery. [Generating LR syntax error messages from examples](#). *ACM Transactions on Programming Languages and Systems*, 25(5):631–640, 2003.
- [JSG⁺23] Harshit Joshi, José Pablo Cambronero Sánchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radicek. [Repair is nearly generation: Multilingual program repair with LLMs](#). In *Conference on Artificial Intelligence (AAAI)*, pages 5131–5140, February 2023.
- [Kla20] Alex Kladov. [Challenging LR parsing](#). Blog post, September 2020.
- [KY10] Ik-Soon Kim and Kwangkeun Yi. [LR error repair using the A* algorithm](#). *Acta Informatica*, 47(3):179–207, 2010.
- [Ler25] Xavier Leroy. The CompCert C compiler. <http://compcert.org/>, 2025.
- [LHS⁺23] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. [Using large language models to enhance programming error messages](#). In *Computer Science Education*, pages 563–569, 2023.
- [MCP21] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. [Catala: a programming language for the law](#). *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29, 2021.

- [MM18] Sérgio Medeiros and Fabio Mascarenhas. [Syntax error recovery in parsing expression grammars](#). In *Symposium on Applied Computing*, pages 1195–1202, 2018.
- [Mm25] Niko Matsakis and many contributors. [LALRPOP](#), May 2025.
- [NPM08] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. [Compiler error messages: what can help novices?](#) In *Technical Symposium on Computer Science Education*, pages 168–172, March 2008.
- [Pot16] François Pottier. [Reachability and error diagnosis in LR\(1\) parsers](#). In *Compiler Construction (CC)*, pages 88–98, March 2016.
- [PPM⁺17] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani L. Peters, John Homer, Nevan Simone, and Maxine S. Cohen. [On novices’ interaction with compiler error messages: A human factors approach](#). In *International Computing Education Research (ICER)*, pages 74–82, August 2017.
- [PRG25] François Pottier and Yann Régis-Gianas. The Menhir parser generator, 2005–2025. <https://gitlab.inria.fr/fpottier/menhir/>.
- [SB24] Eddie Antonio Santos and Brett A. Becker. [Not the silver bullet: LLM-enhanced programming error messages are ineffective in practice](#). In *Conference on United Kingdom & Ireland Computing Education Research (UKICER)*, 2024.
- [SCP⁺18] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. [Syntax and sensibility: Using language models to detect and correct syntax errors](#). In *Software Analysis, Evolution and Reengineering (SANER)*, pages 311–322, March 2018.
- [Tal19] Sean Talts. [stanc3: rewriting the Stan compiler](#). Blog post, March 2019.
- [WK17] John Wrenn and Shriram Krishnamurthi. [Error messages are classifiers: a process to design and evaluate error messages](#). In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 134–147, October 2017.