

Mapping and Explaining Syntax Errors with LRgrep

Frédéric Bour & François Pottier

January 27, 2026



A syntactically incorrect OCaml program:

```
let x = 3;  
let y = 4  
let z = 5
```

A syntactically incorrect OCaml program:

```
let x = 3;  
let y = 4  
let z = 5
```

Today, OCaml produces this syntax error message:

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

A syntactically incorrect OCaml program:

```
let x = 3; (* The trailing semicolon is the real mistake! *)  
let y = 4 (* This is mistakenly read as a local declaration. *)  
let z = 5 (* An error is detected at the start of this line. *)
```

A syntactically incorrect OCaml program:

```
let x = 3; (* The trailing semicolon is the real mistake! *)
let y = 4 (* This is mistakenly read as a local declaration. *)
let z = 5 (* An error is detected at the start of this line. *)
```

What we would perhaps like to see:

```
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 before this declaration (1:9-10)
would allow it to be interpreted as a global declaration.
```

- ① The setting
- ② Demo: a toy language

Have:

- Deterministic LR(1) parsing.
- Static non-ambiguity check.

Have:

- Deterministic LR(1) parsing.
- Static non-ambiguity check.

Want:

- A tool that helps *visualize the landscape* of syntax error situations.
- A way of expressing a *declarative* and *programmable* mapping of syntax error situations to syntax error messages.
- Support for detecting *useless* and *redundant* entries in this mapping.
- To *separate* this mapping from the description of the grammar.
- To tolerate the *non-determinism* that arises once the next input symbol is considered unknown.

We wish to write a declarative specification:

error situation { *code that produces an error message* }

We wish to write a declarative specification:

error situation { *code that produces an error message* }

What is an error situation?

We wish to write a declarative specification:

error situation { *code that produces an error message* }

What is an error situation?

What state does an LR parser maintain?



We wish to write a declarative specification:

error situation { *code that produces an error message* }

What is an error situation?

What state does an LR parser maintain?



a stack | the remaining input

We wish to write a declarative specification:

error situation { *code that produces an error message* }

What is an error situation?

What state does an LR parser maintain?



a stack | the remaining input
a list of states

We wish to write a declarative specification:

error situation { *code that produces an error message* }

What is an error situation?

What state does an LR parser maintain?



a stack | the remaining input
a list of states
a list of symbols

We wish to write a declarative specification:

error situation { *code that produces an error message* }

What is an error situation?

What state does an LR parser maintain?



a stack | the remaining input
a list of states
a list of symbols
past input (re-interpreted)

We wish to write a declarative specification:

error situation { *code that produces an error message* }

What is an error situation?

What state does an LR parser maintain?



a stack | the remaining input
a list of states
a list of symbols
past input (re-interpreted)

To describe an error situation is to describe *a set of stack* suffixes.

We need a *language* for this purpose.

To describe a set of stacks, we use *regexps* plus a few ad hoc constructs:

$e ::=$	<i>symbol</i>	– <i>terminal or non-terminal</i>
	$(e e) \mid (e \mid e) \mid e^*$	
	$[e]$	– <i>matching up to reduction</i>
	$/item$	– <i>filtering</i>

To describe a set of stacks, we use *regexps* plus a few ad hoc constructs:

$e ::=$	<i>symbol</i>	– <i>terminal or non-terminal</i>
	$(e e) \mid (e \mid e) \mid e^*$	
	$[e]$	– <i>matching up to reduction</i>
	$/item$	– <i>filtering</i>

Examples:

- $[expr]$ matches all stacks that *can* be reduced to $\dots expr$.

To describe a set of stacks, we use *regexps* plus a few ad hoc constructs:

$e ::= symbol$	– <i>terminal or non-terminal</i>
$(e e) \mid (e \mid e) \mid e^*$	
$[e]$	– <i>matching up to reduction</i>
$/item$	– <i>filtering</i>

Examples:

- $[\underbrace{(}_{\text{a terminal symbol}} \quad \underbrace{expr}_{\text{a nonterminal symbol}} \quad \underbrace{/expr : (expr \bullet)}_{\text{an item}}]$

matches all stacks that *can* be reduced to $\dots (expr$
and whose top state contains the item $expr : (expr \bullet)$.

- ① The setting
- ② Demo: a toy language

```

%token <int> INT                (* Tokens. *)
%token <string> IDENT
%token PLUS MINUS TIMES DIV EQUAL LPAREN RPAREN SEMI LET IN EOF
%nonassoc IN                    (* Precedence declarations. *)
%right SEMI
%left PLUS MINUS
%left TIMES DIV
%start <unit> file              (* Entry point. *)
%%
file: declaration* EOF        {} (* Productions. *)
declaration: LET binding {}
binding: IDENT EQUAL expr {}
expr:
| IDENT | INT
| expr PLUS expr | expr MINUS expr | MINUS expr
| expr TIMES expr | expr DIV expr | expr SEMI expr
| LPAREN expr RPAREN
| LET binding IN expr {}

```

A declaration is now expected

Excerpt of the `.lrngrep` file / sample incorrect input / error message:

```
| / . file  
  { "A declaration is now expected." }
```

garbage

```
File "input01.in" (1:0-7):  
Syntax error.  
A declaration is now expected.
```

The pattern `(/. file)` matches a situation where nothing has been read.

An identifier is now expected

The pattern LET matches a situation where LET is on top of the stack:

```
| l=LET
  { read (start decl) $positions(l) ^
    "An identifier is now expected." }
```

```
let
let x = 0
```

```
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.
```

The capture `l=` lets us refer to a position or semantic value.

An equals sign is now expected

A situation where LET followed with IDENT are on top of the stack:

```
| l=LET; i=IDENT
  { read (start decl) ($startpos(l), $endpos(i)) ^
    "An equals sign '=' is now expected." }
```

```
let x + 2 = 2
```

```
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.
```

I like to explain *what has been understood* and *what is expected* next.

What has been read is *described abstractly* and *shown concretely*.

An expression is now expected

A situation where LET, IDENT, EQUAL have been read:

```
| l=LET; IDENT; e=EQUAL
  { read (start decl) ($startpos(l), $endpos(e)) ^
    "An expression is now expected." }
```

```
let x = )
let z = 0
```

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.

I do *not* list all the ways in which an expression might begin.

An expression is now expected

Situations where an expression is expected, in a different context:

```
| e=expr; o=PLUS | e=expr; o=MINUS
| e=expr; o=DIV | e=expr; o=TIMES | e=expr; o=SEMI
  { read (expr ++ binop) ($startpos(e), $endpos(o)) ^
    "An expression is now expected." }
```

```
let a =
  let x = 1+2- in
  0
```

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.

Yet another situation where an expression is expected:

```
| l=LPAREN / .* . expr .*  
  { read lparen $positions(l) ^  
    "An expression is now expected." }
```

```
let x = 1+(+)
```

```
File "input06.in" (1:11-12):
```

```
Syntax error.
```

```
An opening parenthesis has been read (1:10-11):
```

```
(
```

```
An expression is now expected.
```

I do not merge all of the situations where an expression is expected because I want to show and explain what has been recently read.

Another declaration is now expected

A situation where the recent input *can be understood as* a declaration:

```
| d=[declaration]
  { read decl $positions(d) ^
    "If this declaration is complete, then" ^/^
    "another declaration is now expected." }
```

```
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 is now expected.

The keyword `in` is now expected

LET, followed with what can be understood as a binding, has been read:

```
| l=LET; b=[binding]
  { read ldecl ($startpos(l), $endpos(b)) ^
    "The keyword 'in' 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.
```

This declaration must be local: toplevel decls have already been handled.

A special case of the previous situation, with more left context:

```
| [declaration]; s=SEMI; l=LET; b=[binding]
  { read ldecl ($startpos(l), $endpos(b)) ^
    "The keyword 'in' is now expected." ^/^
    sprintf "Suggestion: [...]" (* An extra suggestion. *) }
```

```
let x = 3; (* The trailing semicolon is the real mistake! *)
let y = 4 (* This is mistakenly read as a local declaration. *)
let z = 5 (* An error is detected at the start of this line. *)
```

File "test11.in" (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 before this declaration (1:9-10) would allow it to be interpreted as a global declaration.

A closing parenthesis is now expected

LPAR, followed with what can be viewed as an expression, has been read:

```
| l=LPAREN; e=[expr]
  { read (lparen ++ expr) ($startpos(l), $endpos(e)) ^
    "If this expression is complete," ^/^
    "a closing parenthesis 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.

"If this expression is complete" could be removed. *White lies* are fine.

LRgrep 0.3 has just been released.

This [demo](#) is available online.

LRgrep is still being

- improved by Frédéric so as to better scale,
- applied to [OxCaml](#)'s grammar:
 - 147 terminal symbols, 265 nonterminal symbols,
 - 1485 productions,
 - 3675 states.