A Glimpse and Demo of LRgrep

Frédéric Bour & François Pottier

April 3, 2025

What we have today

A syntactically incorrect OCaml program:

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

What we have today

A syntactically incorrect OCaml program:

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

Today, OCaml produces this syntax error message:

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

What we want

A syntactically incorrect OCaml program:

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

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

The setting

Have:

- Deterministic LR(1) parsing.
- Static non-ambiguity check.
  - Some people in this room will advocate SGLR instead!

The setting

Have:

- Deterministic LR(1) parsing.
- Static non-ambiguity check.
  - Some people in this room will advocate SGLR instead!

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.

Plan of attack

We wish to write a declarative specification:

$$\textit{error situation} \rightarrow \{\, \textit{code that produces an error message} \,\}$$

Plan of attack

We wish to write a declarative specification:

$$\textit{error situation} \rightarrow \{\, \textit{code that produces an error message} \,\}$$

What is an error situation?

Plan of attack

We wish to write a declarative specification:

$$\textit{error situation} \rightarrow \{\ \textit{code that produces an error message}\ \}$$

What is an error situation?

What state does an LR parser maintain?

Plan of attack

We wish to write a declarative specification:

$$\textit{error situation} \rightarrow \{\ \textit{code that produces an error message}\ \}$$

What is an error situation?

What state does an LR parser maintain?

a stack | the remaining input

Plan of attack

We wish to write a declarative specification:

$$\textit{error situation} \rightarrow \{\ \textit{code that produces an error message}\ \}$$

What is an error situation?

What state does an LR parser maintain?

a stack | the remaining input

a list of states

Plan of attack

We wish to write a declarative specification:

$$\textit{error situation} \rightarrow \{\ \textit{code that produces an error message}\ \}$$

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

Plan of attack

We wish to write a declarative specification:

$$\textit{error situation} \rightarrow \{\ \textit{code that produces an error message}\ \}$$

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)

Plan of attack

We wish to write a declarative specification:

$$\textit{error situation} \rightarrow \{\ \textit{code that produces an error message}\ \}$$

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.

LRgrep expressions

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

$$
\begin{array}{rcll}
e & ::= & \mathit{symbol} & \text{– terminal or non-terminal} \\
 & & (e\ e) \mid (e \mid e) \mid e^{\star} & \\
 & & [e] & \text{– matching up to reduction} \\
 & & /\mathit{item} & \text{– filtering}
\end{array}
$$

LRgrep expressions

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

$$
\begin{array}{rcll}
e & ::= & \mathit{symbol} & \text{– terminal or non-terminal} \\
 & & (e\ e) \mid (e \mid e) \mid e^{\star} & \\
 & & [e] & \text{– matching up to reduction} \\
 & & /\mathit{item} & \text{– filtering}
\end{array}
$$

Examples:

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

LRgrep expressions

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

$$
\begin{array}{rcll}
e & ::= & \mathit{symbol} & \text{– terminal or non-terminal} \\
 & & (e\ e) \mid (e \mid e) \mid e^{\star} & \\
 & & [e] & \text{– matching up to reduction} \\
 & & /\mathit{item} & \text{– filtering}
\end{array}
$$

Examples:

- $[\mathit{expr}]$ matches all stacks that can be reduced to $\ldots \mathit{expr}$.
- $[\ (\ \mathit{expr}\ /\ \underbrace{\mathit{expr} : (\ \mathit{expr} \bullet)}_{\text{an LR(0) item}}\ ]$

  matches all stacks that can be reduced to $\ldots (\ \mathit{expr}$
  and whose top state contains the item $\mathit{expr} : (\ \mathit{expr} \bullet)$.

DEMO