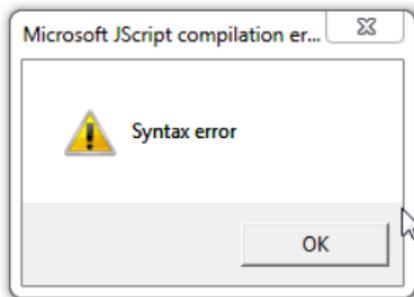


Expliquer les erreurs de syntaxe (LR de rien, c'est pas facile)

François Pottier



OCaml Users in Paris
November 10, 2015



What we do : CompCert's new diagnostic messages

How we do it : Menhir's new features

Show the past, show (some) futures

```
color->y = (sc.kd * amb->y + il.y + sc.ks * is.y * sc.y;
```

```
$ ccomp -c render.c
```

```
render.c:70:57: syntax error after 'y' and before ';'.
```

```
Up to this point, an expression has been recognized:
```

```
'sc.kd * amb->y + il.y + sc.ks * is.y * sc.y'
```

```
If this expression is complete,
```

```
then at this point, a closing parenthesis ')' is expected.
```

Guidelines :

- ▶ Show **the past** : what has been recently understood.
- ▶ Show **the future** : what is expected next...
- ▶ ...but **do not show** every possible future.

Stay where we are

```
multvec_i[i = multvec_j[i] = 0;
```

```
$ ccomp -c subsumption.c
```

```
subsumption.c:71:34: syntax error after '0' and before ';'.
```

```
Ill-formed expression.
```

```
Up to this point, an expression has been recognized:
```

```
'i = multvec_j[i] = 0'
```

```
If this expression is complete,
```

```
then at this point, a closing bracket ']' is expected.
```

Guidelines :

- ▶ Show [where the problem was detected](#),
- ▶ even if the actual error took place earlier.

Show high-level futures ; show enough futures

```
void f (void) { return; }}
```

```
$ gcc -c braces.c
```

```
braces.c:1: error: expected identifier or '(' before '}' token
```

```
$ clang -c braces.c
```

```
braces.c:1:26: error: expected external declaration
```

```
$ ccomp -c braces.c
```

```
braces.c:1:25: syntax error after '}' and before '}'.
```

```
At this point, one of the following is expected:
```

```
  a function definition; or
```

```
  a declaration; or
```

```
  a pragma; or
```

```
  the end of the file.
```

Show high-level futures ; show enough futures

Guidelines :

- ▶ Do not just say what **tokens** are allowed next :
- ▶ instead, say what high-level **constructs** are allowed.
- ▶ List **all** permitted futures, if that is reasonable.

Show enough futures

```
int f(void) { int x;) }
```

```
$ gcc -c indecision.c
```

```
indecision.c: In function 'f':  
indecision.c:1: error: expected statement before ')' token
```

```
$ clang -c indecision.c
```

```
indecision.c:1:7: error: expected expression
```

```
$ ccomp -c indecision.c
```

```
indecision.c:1:20: syntax error after ';' and before ')'.  
At this point, one of the following is expected:
```

- a declaration; or
- a statement; or
- a pragma; or
- a closing brace '}'.

Show the goal(s)

```
int main (void) { static const x; }
```

```
$ ccomp -c staticconstlocal.c
```

```
staticconstlocal.c:1:31: syntax error after 'const' and before 'x'.  
Ill-formed declaration.
```

At this point, one of the following is expected:

- a storage class specifier; or
- a type qualifier; or
- a type specifier.

Guidelines :

- ▶ If possible and useful, show the [goal](#).
- ▶ Here, we definitely hope to recognize a “declaration”.

Show the goal(s)

```
static const x;
```

```
$ ccomp -c staticconstglobal.c
```

```
staticconstglobal.c:1:13: syntax error after 'const' and before 'x'.  
Ill-formed declaration or function definition.
```

At this point, one of the following is expected:

- a storage class specifier; or
- a type qualifier; or
- a type specifier.

Guidelines :

- ▶ Show multiple **goals** when the choice has not been made yet.
- ▶ Here, we hope to recognize a “declaration” **or** a “function definition”.

What we do : CompCert's new diagnostic messages

How we do it : Menhir's new features

How to diagnose syntax errors ?

Jeffery's idea (2005) :

*Choose a diagnostic message based on the LR automaton's **state**, ignoring its **stack** entirely.*

Is this a reasonable idea ?

How to diagnose syntax errors ?

Jeffery's idea (2005) :

*Choose a diagnostic message based on the LR automaton's **state**, ignoring its **stack** entirely.*

Is this a reasonable idea ?

If you are thinking the answer is “yes”, you are wrong.

How to diagnose syntax errors ?

Jeffery's idea (2005) :

*Choose a diagnostic message based on the LR automaton's **state**, ignoring its **stack** entirely.*

Is this a reasonable idea ?

If you are thinking the answer is “yes”, you are wrong.

If you are thinking the answer is “no”, you are wrong.

How to diagnose syntax errors ?

Jeffery's idea (2005) :

*Choose a diagnostic message based on the LR automaton's **state**, ignoring its **stack** entirely.*

Is this a reasonable idea ?

If you are thinking the answer is “yes”, you are wrong.

If you are thinking the answer is “no”, you are wrong.

In fact, this approach **can be made** to work, but

- ▶ one needs to know **which sentences** cause errors ;
- ▶ one needs to know (and control) **in which states** these errors are detected ;
- ▶ which requires **tool support**.

Is this a reasonable idea ? – Yes

Sometimes, **yes**, clearly the **state** alone contains enough information.

```
int f (int x) { do {} while (x--) }
```

The error is detected in a state that looks like this :

```
statement: DO statement WHILE LPAREN expr RPAREN . SEMICOLON [...]
```

It is easy enough to give an accurate message :

```
$ ccomp -c dowhile.c
```

```
dowhile.c:1:34: syntax error after ')' and before '}'.
```

```
Ill-formed statement.
```

```
At this point, a semicolon ';' is expected.
```

Is this a reasonable idea ? – Yes, it seems... ?

Here is another example where things *seem* to work out as hoped :

```
int f (int x) { return x + 1 }
```

The error is detected in a state that looks like this :

```
expr -> expr . COMMA assignment_expr [ SEMICOLON COMMA ]  
expr? -> expr . [ SEMICOLON ]
```

We decide to omit the first possibility, and say a semicolon is expected.

```
$ ccomp -c return.c
```

```
return.c:1:29: syntax error after '1' and before '}'.
```

```
Up to this point, an expression has been recognized:
```

```
'x + 1'
```

```
If this expression is complete,
```

```
then at this point, a semicolon ';' is expected.
```

Yet, ',', and ';' are clearly *not* the only permitted futures ! What is going on ?

Is this a reasonable idea? – Uh, oh...

Let us change **just** the incorrect token in the previous example :

```
int f (int x) { return x + 1 2; }
```

The error is now detected **in a different** state, which looks like this :

```
postfix_expr -> postfix_expr . LBRACK expr RBRACK [ ... ]
postfix_expr -> postfix_expr . LPAREN arg_expr_list? RPAREN [ ... ]
postfix_expr -> postfix_expr . DOT general_identifier [ ... ]
postfix_expr -> postfix_expr . PTR general_identifier [ ... ]
postfix_expr -> postfix_expr . INC [ ... ]
postfix_expr -> postfix_expr . DEC [ ... ]
unary_expr -> postfix_expr . [ SEMICOLON RPAREN and 34 more tokens ]
```

Based on this information, **what diagnostic message** can one propose ?

Is this a reasonable idea ? – No !

Based on this, the diagnostic message could say that :

- ▶ The “postfix expression” $x + 1$ can be continued in 6 different ways ;
- ▶ Or maybe this “postfix expression” forms a complete “unary expression”...
- ▶ ...and in that case, it could be followed with 36 different tokens...
- ▶ among which ‘ ; ’ appears, but also ‘) ’, ‘] ’, ‘ } ’, and others !

So,

- ▶ there is a lot of worthless information,
- ▶ yet there is still **not enough** information :
- ▶ we cannot see that ‘ ; ’ is permitted, while ‘) ’ is not.

The missing information is not encoded in the **state** : it is buried in the **stack**.

Two problems

We face two problems :

- ▶ depending on which incorrect token we look ahead at, the error is detected in **different** states ;
- ▶ in some of these states, there is **not enough information** to propose a good diagnostic message.

What can we do about this ?

We propose two solutions to these problems :

- ▶ **Selective duplication.**

In the grammar, distinguish “expressions that can be followed with a semicolon”, “expressions that can be followed with a closing parenthesis”, etc.

(Uses Menhir’s expansion of parameterized nonterminal symbols.)

This **fixes** the problematic states by building more information into them.

- ▶ **Reduction on error.**

In the automaton, perform one more reduction to get us out of the problematic state before the error is detected.

(Uses Menhir’s new `%on_error_reduce` directive.)

This **avoids** the problematic states.

How do we know what we are doing ?

But :

- ▶ how do we **find** all states where an error can be detected ?
- ▶ after tweaking the grammar or automaton, how do we **know** for sure that we have fixed or avoided the problematic states ?

We need tool support.

Menhir's new features

Menhir can now :

- ▶ **list** all states where an error can be detected, together with **example sentences** that cause these errors ;
- ▶ **check** that a set of invalid sentences is **correct**, **irredundant**, and **complete**.

A few figures

(One version of) CompCert's ISO C99 parser :

- ▶ 145 nonterminal symbols, 93 terminal symbols, 365 productions ;
- ▶ 677 states ;
- ▶ 263 error states found in 43 seconds using 1Gb of memory ;
- ▶ we write 150 distinct diagnostic messages.



You can do it, too !