Reachability and error diagnosis in LR(1) parsers

François Pottier

CC 2016
Barcelona, March 17, 2016

40 years ago

> *While it is at least honest, a compiler that **quits** upon first detecting an error **will not be popular** with its users.*
>
> *– James J. Horning, "What the compiler should tell the user" (1976)*

With limited computer access, error diagnosis, recovery, and repair mattered.

Today, I wish to focus on diagnosis : producing a "good" syntax error message.

(Recovery and repair are still of interest, especially in IDEs.)

A myth ?

In spite of 50+ years of research, one often reads :

> *Good error diagnosis requires a hand-written (recursive descent) parser.*
>
> *– Popular belief*

I suggest that this is a bad reason why one should write a parser by hand.

(There are good reasons, such as expressiveness.)

I take this to mean that we need better tools.

Error diagnosis in LR(1) parsers ?

Suppose one uses an LR(1) parser generator, such as yacc, bison, Menhir, etc.

(Declarative. Expressive. Efficient. Guaranteed unambiguity.)

Why on earth would it be difficult to diagnose a syntax error ?

Error diagnosis in LR(1) parsers ?

Let us look at a simple syntax error in a C program :

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

Error diagnosis in LR(1) parsers ?

Let us look at a simple syntax error in a C program :

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

The error is detected in a state (say, 194) that contains one LR(1) item :

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

Error diagnosis in LR(1) parsers ?

Let us look at a simple syntax error in a C program :

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

The error is detected in a state (say, 194) that contains one LR(1) item :

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

It is easy to associate an accurate message with this state :

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

```
dowhile.c:1:34: syntax error after ')' and before '}'.
Ill-formed statement.
At this point, a semicolon ';' is expected.
```

Jeffery's idea

Jeffery (2003) suggests :

- associating a handwritten diagnostic message...
- ...with an example invalid sentence,
- and letting a tool (**merr**) translate the sentence to a state number (say, 194).

One builds (by hand) a collection of sentence/message pairs,
which the tool turns into a mapping of states to messages.

Jeffery's idea

Jeffery (2003) suggests :

- associating a handwritten diagnostic message...
- ...with an example invalid sentence,
- and letting a tool (**merr**) translate the sentence to a state number (say, 194).

One builds (by hand) a collection of sentence/message pairs,
which the tool turns into a mapping of states to messages.

This approach has (at least) two potential shortcomings...

Shortcoming 1 – incompleteness

Jeffery's Unicon parser has 453 states. He sets up a collection of 70 messages, covering 60 states.

Shortcoming 1 – incompleteness

Jeffery's Unicon parser has 453 states. He sets up a collection of 70 messages, covering 60 states.

However, this automaton has 224 error states, that is, 224 states where an error can occur. This collection of diagnostic messages is very much incomplete.

It is hard for a human to come up with a collection of sentences that reaches all error states.

Shortcoming 1 – incompleteness

Jeffery's Unicon parser has 453 states. He sets up a collection of 70 messages, covering 60 states.

However, this automaton has 224 error states, that is, 224 states where an error can occur. This collection of diagnostic messages is very much incomplete.

It is hard for a human to come up with a collection of sentences that reaches all error states.

How do I know there are 224 error states ? ...

Solution 1 – a reachability algorithm

How does one determine which states are error states ?

For every state s and terminal symbol z such that (s, z) is an error entry, we ask :

- is the configuration (s, z) reachable ? (if so, via which sentence ?)
- in other words : is this an essential error entry ?

Solution 1 – a reachability algorithm

How does one determine which states are error states ?

For every state s and terminal symbol z such that (s, z) is an error entry, we ask :

- is the configuration (s, z) reachable ? (if so, via which sentence ?)
- in other words : is this an essential error entry ?

How does one answer this question ?

- In a canonical automaton for an LR(1) grammar,
  the answer is positive iff s is the target of a terminal transition.

Solution 1 – a reachability algorithm

How does one determine which states are error states ?

For every state s and terminal symbol z such that (s, z) is an error entry, we ask :

- is the configuration (s, z) reachable ? (if so, via which sentence ?)
- in other words : is this an essential error entry ?

How does one answer this question ?

- In a canonical automaton for an LR(1) grammar,
  the answer is positive iff s is the target of a terminal transition.
- Not true otherwise !
  - conflicts, precedence declarations ;
  - merged states ;
  - default reductions ;
  - `%on_error_reduce` declarations (coming up).

A reachability algorithm is needed. The paper gives one. (New.)

Shortcoming 2 – inability to give an accurate diagnostic

In Jeffery's approach, one selects a diagnostic message based solely on the parser's control state, ignoring its stack entirely.

Is this reasonable ? (New question, it seems.)

Shortcoming 2 – inability to give an accurate diagnostic

In Jeffery's approach, one selects a diagnostic message based solely on the parser's control state, ignoring its stack entirely.

Is this reasonable ? (New question, it seems.)

In the previous example ("missing semicolon after do/while"), it was.

Shortcoming 2 – inability to give an accurate diagnostic

In Jeffery's approach, one selects a diagnostic message based solely on the parser's control state, ignoring its stack entirely.

Is this reasonable ? (New question, it seems.)

In the previous example ("missing semicolon after do/while"), it was.

However, there are states for which one cannot produce an accurate message without access to the stack.

Let me demonstrate this...

Too few continuations may be visible in the control state

Here is another “missing-semicolon” situation :

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

Too few continuations may be visible in the control state

Here is another “missing-semicolon” situation :

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

The error is detected in a state that contains two LR(1) items :

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

Too few continuations may be visible in the control state

Here is another “missing-semicolon” situation :

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

The error is detected in a state that contains two LR(1) items :

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

Yet ’,’ and ’;’ are clearly not the only permitted continuations ! What’s going on ?

Too few continuations may be visible in the control state

Here is another “missing-semicolon” situation :

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

The error is detected in a state that contains two LR(1) items :

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

Yet `','` and `';'` are clearly not the only permitted continuations ! What’s going on ?

This particular state was reached because the lookahead token is `'}'`.

The presence of this token allowed certain “spurious” reductions to take place before the error was detected. (This is a noncanonical automaton.)

What if the lookahead token was (say) `'2'` instead of `'}'` ?

Too many continuations may be visible in the control state

This is the same “missing-semicolon” situation, with a different incorrect token :

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

Too many continuations may be visible in the control state

This is the same “missing-semicolon” situation, with a different incorrect token :

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

The error is now detected in a different state :

```
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, can one propose an accurate diagnostic message ?

Too many continuations may be visible in the control state

This is the same “missing-semicolon” situation, with a different incorrect token :

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

The error is now detected in a different state :

```
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, can one propose an accurate diagnostic message ?

No. These items do not allow telling that ’;’ is permitted, while ’)’ is not.

This information is buried in the stack.

Solution 2 – exploiting the stack, without hassle

The easiest way of exploiting the stack is to reduce.

Solution 2 – exploiting the stack, without hassle

The easiest way of exploiting the stack is to reduce.

In the problematic state, a reduction is permitted :

```
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 ]
```

Solution 2 – exploiting the stack, without hassle

The easiest way of exploiting the stack is to reduce.

In the problematic state, a reduction is permitted :

```
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 ]
```

Instead of diagnosing the error in this state, let's perform this reduction first. (New.)

```
%on_error_reduce unary_expr
```

This causes all error entries in this state to be replaced with reduction actions.

By reducing, we go to a different state, where error diagnosis may be easier.

Which state we go to depends on the stack.

Workflow of a Menhir user

The Menhir LR(1) parser generator :

- lists all error states, together with example invalid sentences that lead to them.

The grammar author :

- manually constructs a diagnostic message for each error state ;
- adjusts the grammar or automaton to make this task easier.

Menhir :

- updates the list of example sentences and messages as the grammar evolves ;
- checks that this list remains correct, irredundant, and complete ;
- translates it to a mapping of states to messages.

Application to the CompCert C compiler

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

- 145 nonterminal symbols, 93 terminal symbols, 365 productions ;
- a 677-state LALR(1) automaton ;
- 263 error states, found by Menhir in 43 seconds using 1Gb of memory ;
- 150 distinct hand-written diagnostic messages.

One example message

```
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.
```

Diagnosis takes place where the parser failed.
We do not attempt to locate and explain the "real" error.

We show what has been understood so far (an expression), what is expected next (a closing bracket), and to what end (to form a larger expression).

The corresponding message template

One of the 263 entries in CompCert's **handcrafted.messages** file :

```
INT VAR_NAME EQ VAR_NAME LBRACK VAR_NAME SEMICOLON
##
## Ends in an error in state: 114.
##
## expr -> expr . COMMA assignment_expr [ RBRACK COMMA ]
## postfix_expr -> postfix_expr LBRACK expr . RBRACK [ 42 tokens... ]
```

```
Ill-formed expression.
Up to this point, an expression has been recognized:
  $0
If this expression is complete,
then at this point, a closing bracket ']' is expected.
```

The first half is auto-generated and auto-updated ; the second half is hand-written.

A well-formed expression has been read and is currently stored in stack cell 0.
Its text is substituted for the placeholder `$0`.

Conclusion

Contributions :

- An investigation of, and improvements upon, Jeffery's method.
- All error states are found, thanks to a new reachability algorithm.
- New `%on_error_reduce` declarations allow exploiting the stack.
- An integrated workflow in Menhir.
- A practical application in the CompCert C compiler.

A generated parser can produce good error messages ! (Go tell your Grandpa.)

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.
```

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 enough futures

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

```
$ gcc -c extra.c
```

```
extra.c: In function ‘f’:
extra.c:1: error: expected statement before ‘)’ token
```

```
$ clang -c extra.c
```

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

```
$ ccomp -c extra.c
```

```
extra.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.
```

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.
```

The algorithm's specification : a big-step semantics of LR(1) automata

$$\textsc{Init}\quad s \xrightarrow{\epsilon \,/\, \epsilon}\!\!\!\!\rightarrow\; s\,[z]$$

$$\textsc{Step-Terminal}\quad \frac{s \xrightarrow{\alpha \,/\, w}\!\!\!\!\rightarrow\; s'\,[z] \qquad \mathcal{A} \vdash s' \xrightarrow{z} s''}{s \xrightarrow{\alpha z \,/\, wz}\!\!\!\!\rightarrow\; s''\,[z']}$$

$$\textsc{Step-NonTerminal}\quad \frac{s \xrightarrow{\alpha \,/\, w}\!\!\!\!\rightarrow\; s'\,[z] \qquad \mathcal{A} \vdash s' \xrightarrow{A} s'' \qquad s' \xrightarrow{A \,/\, w'} s''\,[z'] \qquad z = \mathrm{first}(w'z')}{s \xrightarrow{\alpha A \,/\, ww'}\!\!\!\!\rightarrow\; s''\,[z']}$$

$$\textsc{Reduce}\quad \frac{\mathcal{A} \vdash s \xrightarrow{A} s'' \qquad s \xrightarrow{\alpha \,/\, w}\!\!\!\!\rightarrow\; s'\,[z] \qquad \mathcal{A} \vdash s' \text{ reduces } A \to \alpha \text{ on } z}{s \xrightarrow{A \,/\, w} s''\,[z]}$$

FIGURE: Inductive characterization of the predicates $s \xrightarrow{\alpha \,/\, w}\!\!\!\!\rightarrow\; s'\,[z]$ and $s \xrightarrow{A \,/\, w} s'\,[z]$.

The algorithm's performance