Validating $LR(1)$ parsers

Jacques-Henri Jourdan François Pottier Xavier Leroy

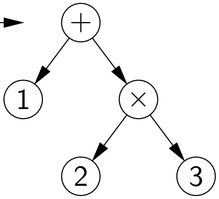
INRIA Paris-Rocquencourt, projet Gallium

IFIP WG 2.8, Nov 2012

Parsing: recap

$$\begin{tikzcd}\text{text or token stream} \arrow[r] & \text{abstract syntax tree}\end{tikzcd}$$

$$1 + 2 \times 3 \longrightarrow$$



Parsing: problem solved?

After 50 years of computer science:

Foundations: Context-Free Grammars, Backus-Naur Form, $LL(k)$, $LR(k)$, Generalized LR, Parsing Expression Grammars, ...

Libraries: parsing combinators, Packrat, ...

Parser generators: Yacc, Bison, ANTLR, Menhir, Elkhound, ...

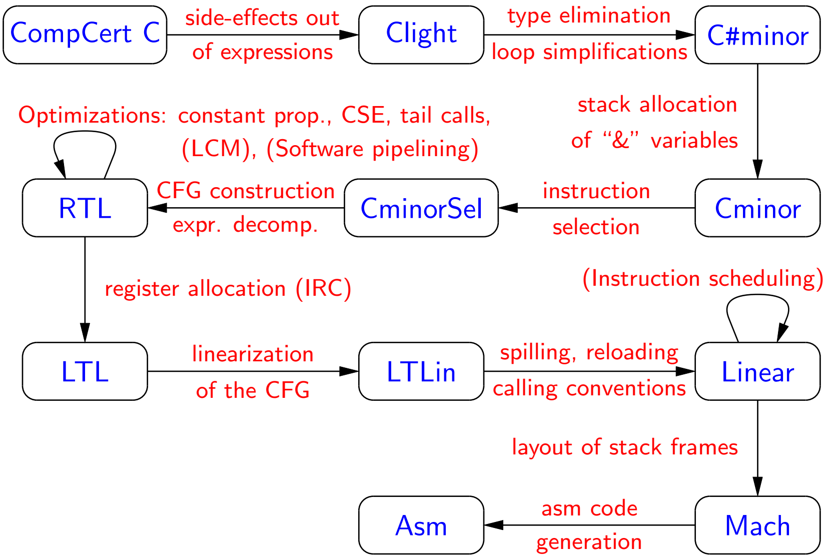
The correctness issue

How can we make sure that a parser (generated or hand-written) is correct?

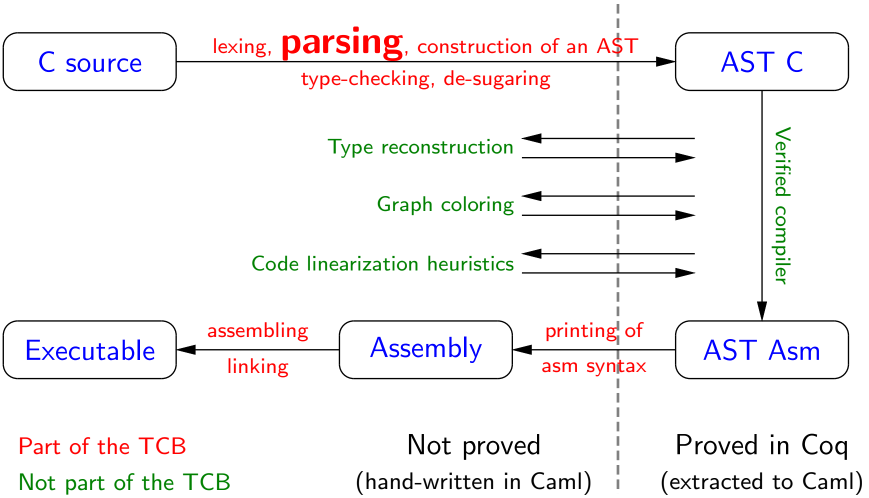
Application areas where it matters:

- Formally-verified compilers, code generators, static analyzers.
- Security-sensitive applications: SQL queries, handling of semi-structured documents (PDF, HTML, XML, . . .).

CompCert: the formally verified part



CompCert: the whole compiler



Correct with respect to what?

Specification of a parser: a context-free grammar with semantic actions.

- Terminal symbols a
- Nonterminal symbols A
- Symbols $X ::= a \mid A$
- Start symbol S
- Productions $A \to X_1 \ldots X_n \ \{f\}$

  $f : T(X_1) \to \cdots \to T(X_n) \to T(A)$ is a semantic action

  $T(X) : \texttt{Type}$ is the type of semantic values for symbol X.

Lovely dependent types!

```
Variable symbol: Type.

Variable T: symbol -> Type.

Fixpoint type_of_sem_action
          (lhs: symbol) (rhs: list symbol) : Type :=
  match rhs with
  | nil => T lhs
  | s :: rhs' => (T s -> type_of_sem_action lhs rhs')
  end.
```

If $T(X) = T(Y) = \texttt{nat}$, we do have that
$\texttt{plus} : \texttt{type\_of\_sem\_action}\ X\ (Y :: Y :: nil)$

Semantics of grammars

$X \to w / v$ (symbol X derives word w producing semantic value v)

$$a \to a \qquad \frac{A \to X_1 \ldots X_n \ \{f\} \text{ is a production} \quad X_i \to w_i / v_i \text{ for } i = 1, \ldots, n}{A \to w_1 \ldots w_n / f(v_1, \ldots, v_n)}$$

Semantics of grammars

$X \to w/v$ (symbol X derives word w producing semantic value v)

$$a \to (a, v)/v \qquad \frac{\begin{array}{c} A \to X_1 \ldots X_n \ \{f\} \text{ is a production} \\ X_i \to w_i/v_i \text{ for } i = 1, \ldots, n \end{array}}{A \to w_1 \ldots w_n / f(v_1, \ldots, v_n)}$$

Correctness of a parser

A parser = a function

$$\text{token stream} \to \texttt{Reject} \mid \texttt{Accept}(\text{semantic value}, \text{token stream})$$

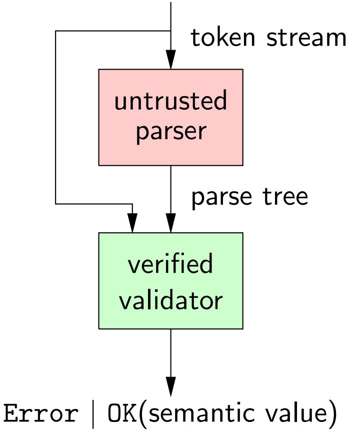
Soundness:
if $\texttt{Parser}(W) = \texttt{Accept}(v, W')$, there exists a word w such that $W = w.W'$ and $S \to w/v$.

Non-ambiguity:
if $\texttt{Parser}(W) = \texttt{Accept}(v, W')$ and and $S \to w/v'$, then $W = w.W'$ and $v' = v$.

Completeness:
if $S \to w/v$ then $\texttt{Parser}(w.W') = \texttt{Accept}(v, W')$.

(Note: completeness + determinism $\Rightarrow$ non-ambiguity.)

Verifying a parser, approach 1: a posteriori validation at every parse



🟩 : proved correct in Coq

🟥 : not verified, untrusted

Validator: trivially checks the parse tree & computes semantic value.

Soundness: guaranteed.
Nonambiguity: no guarantee.
Completeness: no guarantee.

Verifying a parser, approach 2: deductive verification of the parser itself

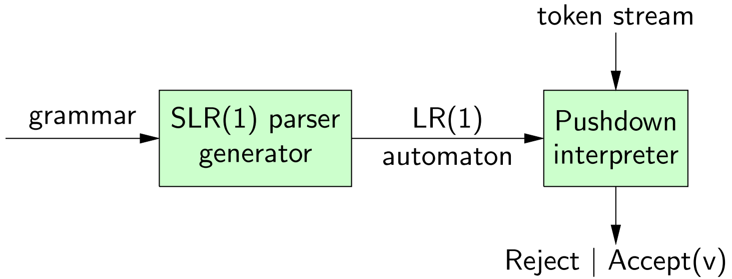
Apply program proof to the parser itself, showing soundness and completeness.

Drawbacks:

- Long and tedious proof,
  especially if parser is generated as an automaton.
- Proof to be re-done every time the grammar changes.

Verifying a parser, approach 3: deductive verification of a parser generator

(A. Barthwal and M. Norrish, *Verified Executable Parsing*, ESOP 2009)

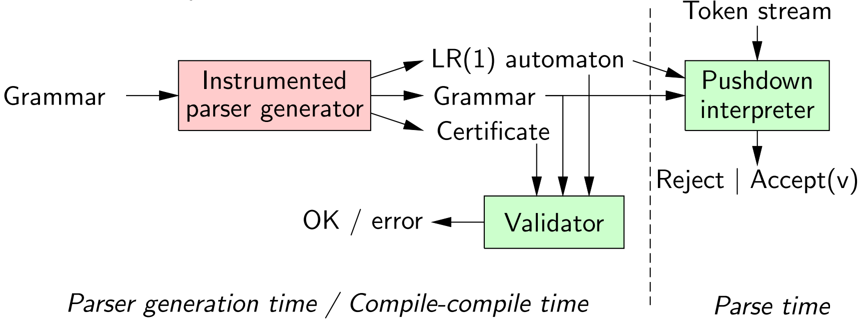


Barthwal & Norrish proved (in HOL) soundness and completeness for every parser successfully generated by their generator.

Limitation: their generator only accepts SLR(1) grammars; the ISO C99 grammar is not SLR(1).

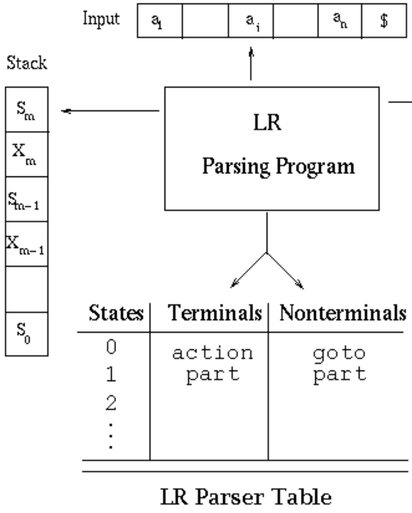
Our approach: verified validation of a parser generator

Given a grammar G and an LR(1) automaton A, check that A is sound and complete w.r.t. G.



The validator supports all flavors of LR(1) parsing: canonical LR(1), SLR(1), LALR(1), Pager's method, ...

Refresher: LR automata



A stack machine with 4 kinds of actions: accept, reject, shift (push the next token), and reduce (by a production) + goto another state.

Interpreting LR(1) automata in Coq

```
Module Parser(G: Grammar) (A: Automaton).

Inductive parse_result :=
  | Accept (v: G.semantic_type G.start_symbol)
           (rem: Stream token)
  | Reject
  | Internal_Error
  | Timeout.

Definition parse (input: Stream token) (fuel: nat)
                                             : parse_result := ...
```

Note `fuel` parameter to guarantee termination
(we can have infinite sequences of reduce actions).

Note `Internal_Error` result caused by e.g. popping from an empty stack.

Soundness

Theorem (Soundness)

If `parse` $W\ N =$ `Accept` $v\ W'$*, there exists a word* w *such that* $W = w.W'$ *and* $S \to w/v$*.*

Note that this theorem holds unconditionally for all automata: the `parse` function performs some dynamic checks and fails with `Internal_Error` in all cases where soundness would be compromised.

Easy Coq proof (200 lines) using an invariant relating the current stack of the automaton with the word read so far.

Safety

Theorem (Safety)

If `safety_validator` $G\ A = \texttt{true}$, *then*
$\texttt{parse}\ W\ N \neq \texttt{Internal\_error}$
for every input stream W *and fuel* N.

`safety_validator` (200 Coq lines) decides a number of properties (next slide) with the help of annotations produced by the parser generator.

Proof of the theorem: 500 Coq lines.

The safety validator

1. For every transition, labeled X, of a state σ to a new state σ',
    - $\textit{pastSymbols}(\sigma')$ is a suffix of $\textit{pastSymbols}(\sigma)\textit{incoming}(\sigma)$,
    - $\textit{pastStates}(\sigma')$ is a suffix of $\textit{pastStates}(\sigma)\{\sigma\}$.
2. For every state σ that has an action of the form $\textit{reduce}\ A \longrightarrow \alpha\ \{f\}$,
    - α is a suffix of $\textit{pastSymbols}(\sigma)\textit{incoming}(\sigma)$,
    - If $\textit{pastStates}(\sigma)\{\sigma\}$ is $\Sigma_n \ldots \Sigma_0$ and if the length of α is k, then for every state $\sigma' \in \Sigma_k$, the goto table is defined at (σ', A). (If k is greater than n, take Σ_k to be the set of all states.)
3. For every state σ that has an *accept* action,
    - $\sigma \neq \textit{init}$,
    - $\textit{incoming}(\sigma) = S$,
    - $\textit{pastStates}(\sigma) = \{\textit{init}\}$.

Completeness

Theorem (Completeness)

If `completeness_validator` $G\ A = \texttt{true}$ *and* $S \rightarrow w/v$,
then there exists a fuel N_0 *such that for all* $N \geq N_0$,
$\texttt{parse}\ (w.W)\ N \in \{\texttt{Accept}(v, W), \texttt{Internal\_Error}\}$.

The proof amounts to taking $N_0 =$ the height of the derivation of $S \rightarrow w/v$, and showing that the automaton performs a depth-first traversal of the parse tree $S \rightarrow w/v$.

`completeness_validator` (next slide): 200 Coq lines.
Proof: 700 Coq lines.

The completeness validator

1. For every state σ, the set *items*(σ) is closed, that is, the following implication holds:

$$\frac{\begin{array}{c} A \longrightarrow \alpha_1 \bullet A'\alpha_2\ [a] \in \mathit{items}(\sigma) \\ A' \longrightarrow \alpha'\ \{f'\} \text{ is a production} \\ a' \in \mathit{first}(\alpha_2 a) \end{array}}{A' \longrightarrow\ \bullet \alpha'\ [a'] \in \mathit{items}(\sigma)}$$

2. For every state σ, if $A \longrightarrow \alpha \bullet\ [a] \in \mathit{items}(\sigma)$, where $A \neq S'$, then the action table maps (σ, a) to *reduce* $A \longrightarrow \alpha\ \{f\}$.
3. For every state σ, if $A \longrightarrow \alpha_1 \bullet a\alpha_2\ [a'] \in \mathit{items}(\sigma)$, then the action table maps (σ, a) to *shift* σ', for some state σ' such that:

$$A \longrightarrow \alpha_1 a \bullet \alpha_2\ [a'] \in \mathit{items}(\sigma')$$

The completeness validator

1. For every state σ, if $A \longrightarrow \alpha_1 \bullet A' \alpha_2\ [a'] \in \mathit{items}(\sigma)$, then the goto table either is undefined at (σ, A') or maps (σ, A') to some state σ' such that:

$$A \longrightarrow \alpha_1 A' \bullet \alpha_2\ [a'] \in \mathit{items}(\sigma')$$

2. For every terminal symbol a, we have $S' \longrightarrow\ \bullet S\ [a] \in \mathit{items}(\mathit{init})$.
3. For every state σ, if $S' \longrightarrow S \bullet\ [a] \in \mathit{items}(\sigma)$, then σ has a default *accept* action.
4. "*first*" and "*nullable*" are fixed points of the standard defining equations.

Towards termination

Completeness shows termination for valid inputs, but what about invalid inputs? (We have examples of non-termination for automata that pass the safety and completeness validators.)

Conjecture (Termination)

Assuming some to-be-determined validation conditions hold, for every finite input W there exists a fuel N_0 such that `parse` $W\ N \neq$ `Timeout` *for all* $N \geq N_0$*.*

A proof sketch in Aho and Ullman, but only for canonical LR(1) automata (which have a peculiar “early failure” property).

Experimental validation: ISO C 1999

Starting point: grammar from Appendix A of ISO C 99 standard.

Removed "old-style" function declarations (unsupported by CompCert).

Fixed / worked around several ambiguities (next slides).

$\rightarrow$ Grammar with 87 terminals, 72 nonterminals, 263 productions.

Modified the Menhir parser generator to produce Coq output + certificates (500 lines of Caml).

$\rightarrow$ Pager's LR(1) automaton with 505 states.
$\rightarrow$ Plus 4.2 Mbytes of certificates (mostly, item sets).

Experimental validation: ISO C 1999

Running the validators on Menhir's Coq output:

- Executed within Coq (`Eval vm_compute`).
- Reading and type-checking Menhir's output: 32 s.
- Safety validator: 4 s.
- Completeness validator: 15 s.

Replacing CompCert's unproved parser with our new parser:

- Parsing: 5 times slower.
- Total compilation time: +20%

Ambiguities in the C grammar
1- Dangling else

```
if (cond1)
    if (cond2)
        x = 1;
  else
      x = 2;
```

A classic problem: which `if` matches the `else`?

ISO standard says “the second `if`”, but not reflected in grammar.

A simple solution: rewrite the grammar to have two *statement* nonterminals, one for statements that can be followed by `else`, the other for statements that cannot.

Ambiguities in the C grammar
2- Type names

```
a * b;
```

In the scope of a `typedef ... a;` declaration, this means

“Declare a variable `b` of type pointer to `a`.”

Otherwise, this means

“Compute `a` times `b` and throw result away.”

→ Must have two different terminals for type names and variable names. The lexer must classify identifiers into type names or variable names taking `typedef` declarations and block scopes into account.

Ambiguities in the C grammar
2- Type names

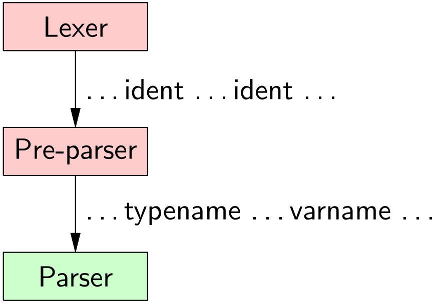
Classic solution: the lexer hack.

Semantic actions of the parser update a symbol table.
The lexer consults this table to classify identifiers.

Our approach: a pre-parser.

The pre-parser keeps track of typedefs that are in scope, and adjusts the stream of tokens accordingly.

In our implementation, the pre-parser is a full, non-verified C parser using the lexer hack.



Ambiguities in the C grammar
3- Binding occurrences

```
typedef double a;

a a;
```

First `a` is a type name, second `a` is a variable name in binding position, subsequent a's are variable names.

Here, no other possible interpretation; but . . .

Ambiguities in the C grammar
3- Binding occurrences

```
typedef double a;

int f(int (a));
```

Could mean either: (in civilized Coq syntax)

1. `f : forall (a: int), int`
2. `f : (a -> int) -> int`

Original ISO C99 standard leaves this ambiguity open.
Technical Corrigendum 2 says interpretation #2 is correct.

Again, we rely on the pre-parser for correct classification.

Conclusions

Once more, the “verified validator” approach is a win:

- Reduced proof effort (2 500 lines versus Barthwal and Norrish’s 20 000).
- Reusable with all known LR(1) constructions (from canonical to Pager’s).
- Can also reuse existing, mature parser generator (e.g. Menhir and its excellent diagnostics).

Possible improvements

Prove termination?

Prove that the parser does not read more tokens than necessary.
(Important for interactive applications, e.g. toplevel loops.)

Speed up the pushdown interpreter by removing dynamic checks.
($\rightarrow$ much more dependent types?)

Take precedences and associativity declarations into account.

Perspectives for CompCert

A similar validation approach should work for the lexer as well. (Perhaps using Brozowski's derivatives.)

Simplify the pre-parser by restricting `typedef` to global scope. (Very few C codes use local `typedef`.)

The elaboration passes (between the parser and the input to the first Coq-proved pass) need work.