Tokenization, Indentation-sensitive Grammars and Executable Recognizers

Basil Fürer Dmitriy Traytel

November 17, 2026

Abstract

Two definitional packages for Isabelle/HOL that turn the source text of a language into verified values: The command **lexer** elaborates a flex-style specification, with modes, longest match and rule priority, into an inductive lexing judgment and an executable lexer on precomputed DFA tables. The command **grammar** elaborates an indentation-sensitive context-free grammar with semantic actions into a grammar specification, decoders from parse trees to the values of the semantic actions and an inductive relation. The related command **parser** emits an executable LR parser for it. Both executable constants for lexing and parsing are proved sound and complete with respect to their specifications. The parser completeness, and with it the unambiguity of the grammar, follows from an LALR conflict analysis that the command carries out by evaluation.

Contents

1 Overview **3**

2 Lexers **3**
2.1 Specifications 4
2.2 Patterns 8
2.3 What is defined 8
2.4 Example 10

3 Grammars and parsers **11**
3.1 Specifications 11
3.2 Precedence 15
3.3 Composition 16
3.4 Templates and operators 17
3.5 Parsers 17
3.6 What is defined 18
3.7 Diagnostics and recovery 20
3.8 Example 23

1 Overview

TIGER (Tokenization, Indentation-sensitive Grammars and Executable Recognizers) provides Isabelle/HOL commands that turn a specification of syntax into executable recognizers. They work like the lexer and parser generators `flex` [17], ocamlyacc [14] and Menhir [19]. Unlike Isabelle's mixfix annotations [20, Sect. 9.2] or `Isabelle_Lex-Yacc` [5] they also prove that what they generate meets the specification: a lexing judgment after Verbatim [9], and a grammar with layout constraints after Adams [1].

The TIGER entry provides three main commands:

- **lexer** (Section 2) defines a lexer specification with modes and rules, its lexing judgment, and an executable lexer on DFA tables that computes the judgment.
- **grammar** (Section 3) defines a grammar with semantic actions, decoders from parse trees to values and, on request, an inductive relation between words and values.
- **parser** (Section 3.5) emits an executable LR parser for grammars, proven sound for the particular grammar. Completeness is proven by evaluation when it has no LALR conflicts.

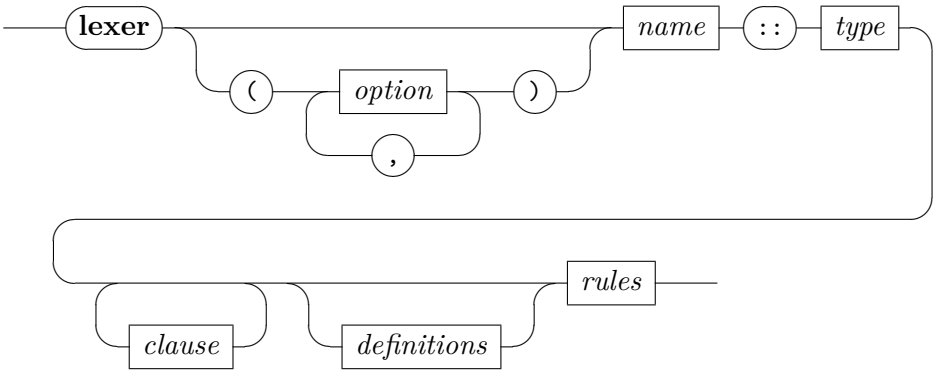
2 Lexers

A lexer specification gives each *mode* a list of *rules*, each a regular expression with an action. Lexing follows flex [17]: in the mode on top of a stack of modes, the longest prefix of the input that a rule matches is the next lexeme, and of the rules that match it the first one wins. Its action gives the token class of the lexeme (or *None* for trivia such as blanks and comments), and the new stack. Trivia go with the token before them, so the source is the concatenation of its tokens and trivia, and each token carries its span: line, column, offset, length and text.

This meaning is an inductive judgment over the specification, not a program (Section 2.3), and the command proves that the lexer it generates computes it. The lexer runs on DFA tables, one per mode. Their states are sets of partial derivatives of the rules' regular expressions, each tagged with its rule: the subset construction over partial derivatives [4, Prop. 4.3] rather than the derivative automata of [6, 16]. The expressions are kept symbolic [12]: their atoms are classes of characters, intervals of codepoints, not single characters, and a transition goes by a minterm of the atoms [7, Sect. 4, Fig. 2].

2.1 Specifications

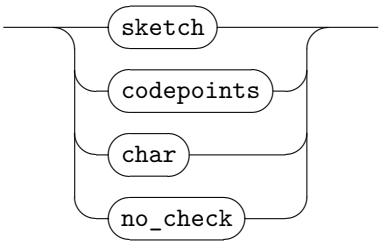
The categories *name*, *string*, *type*, *term* and *cartouche* in the syntax diagrams are those of the Isar reference manual [20, Sects. 4.2–4.3].



lexer L :: ‹τ› ...
: defines the lexer L with token classes of type τ. Each clause is given at most once.

---

option



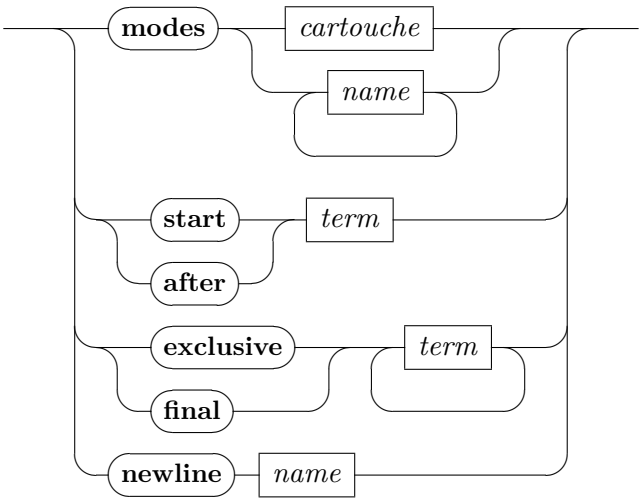
`sketch`
: stops after the specification: generates no tables and no lexer. The other options are ignored.

`codepoints`
: (default) builds a lexer on codepoints, natural numbers: *cp_lexer* on the tables of *L.spec*.

`char`
: builds a lexer on bytes, of type *char*: *char_lexer* on the tables of *char_spec L.spec*, where a class means its part below *0x100*. A class that reaches above *0xFF* is then an error, unless it reaches the last codepoint: `_` and [^...] mean the remaining bytes.

`no_check`
: drops the warning at rules that can never match (Section 2.3).

clause



modes *A B* ...
: names the modes, which become a datatype *L_mode*; **modes** ‹μ› takes an existing datatype μ, whose values the command enumerates. Without the clause there is one mode, of type *unit*.

start *m*
: is the start mode, by default the first.

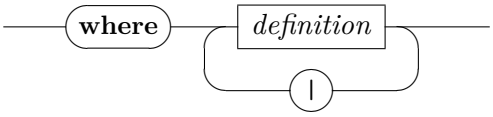
exclusive m_1 ... m_n
: lists modes in which only the rules tagged for them are active (flex's %*x* [17, Sect. 10]).

final m_1 ... m_n
: lists the modes in which the input may end, by default those that are not exclusive: an input that ends in a comment mode is an unterminated comment.

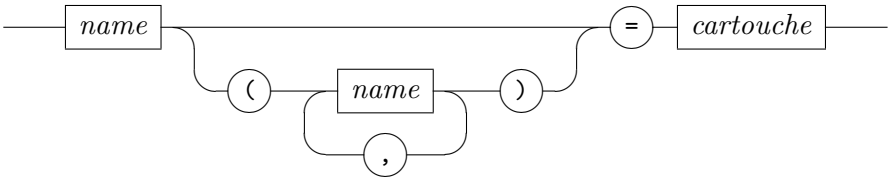
after *h*
: is a hook $h :: \tau\ option \Rightarrow \mu \Rightarrow \mu$ that computes the new top of the stack from the token class of a lexeme, when its rule names no mode (by default the identity). It is lex's user flag [15, Sect. 10] made pure: a flag set in every action is a function of the token.

newline *c*
: names the class of characters that end a line, by default [\n].

definitions



definition

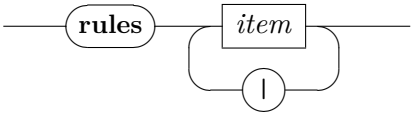


where $c = \langle p \rangle \mid \ldots$

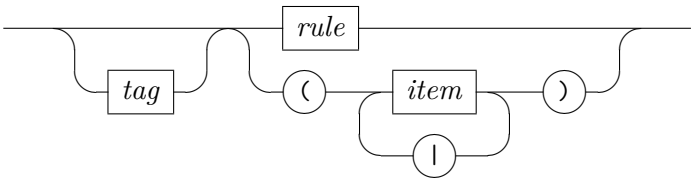
names patterns (lex's definitions [15, Sect. 6]); each becomes a constant *L.c.* A definition with parameters is a template, for example $p+$ may be redefined as $many(p) = \langle p\ p* \rangle$. Templates are substituted where they are used.

---

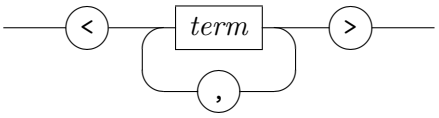
rules



item



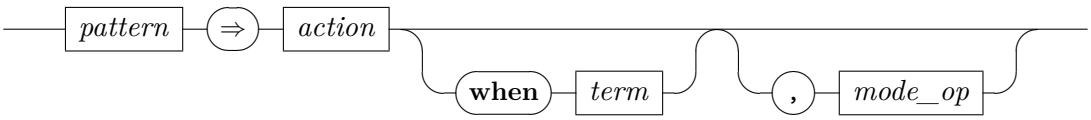
tag



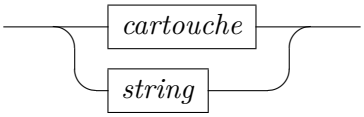
rules $r_1 \mid \ldots \mid r_n$

lists the rules. Each mode has its active rules in the order of the source, so priority is the order of the file. A rule is active in the modes its tag matches, or, without a tag, in every mode that is not exclusive. The terms of a tag are HOL patterns over the mode type, with _ for any part. A scope <*p*> $(r_1 \mid \ldots \mid r_n)$ tags each rule in it; it is written with parentheses where flex has braces, as { opens a proof block in Isar.

rule



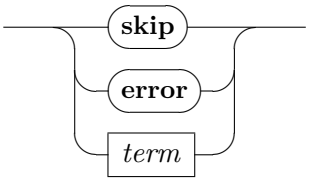
pattern



$p \Rightarrow a$ **when** P, m
: lexes a match of the pattern p with the action a and then applies the mode operation m. The pattern is a cartouche in the syntax of Section 2.2, or a string for the literal text. The guard **when** P makes a lexeme u with $\neg\ P\ u$ an error. $\Rightarrow$ may be written `=>`.

---

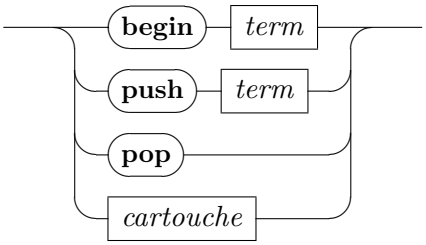
action



skip | **error** | t
: is **skip** for trivia, **error** to stop, or a term: a token class, a function from the lexeme to a token class, or a complete action of type $'ch\ list \Rightarrow 'm\ list \Rightarrow 'k\ option \times 'm\ list$.

---

mode_op



begin m | **push** m | **pop** | $\langle f \rangle$
: change the stack as flex does [17, Sect. 10]: **begin** m replaces the top, **push** m pushes m, **pop** pops, and a cartouche is a function on the stack.

Without a mode operation the stack stays as it is, up to the hook. A pop of the last mode is the empty stack, where lexing stops (flex aborts there).

2.2 Patterns

A pattern elaborates to a class of codepoints, of type *nat cls*, or a regular expression over classes, of type *nat cls rexp*. Its syntax follows flex [17, Sect. 6] and ocamllex [14, Sect. 17.2.4], with the usual precedence: postfix operators bind tightest, then concatenation, then the difference #, then the union |.

"text" is the literal text, with the escapes \n, \t, \r, \\, \" and \u{h} for the codepoint with hexadecimal number *h*.

[*a*−*z*α−ω*0*−*9*] is a class of characters and ranges, a dash at either end for itself, and a backslash before any character for that character. As in flex, a blank in a class is a member: [*a*−*z A*−*Z*] holds the space, an escaped blank is an error, and a newline is written \n. [^...] is its complement, and _ is any codepoint.

A character is a codepoint: an ASCII or UTF-8 character, or a named symbol such as α, read through the *code* column of Isabelle's symbol table.

p∗, *p*+ and *p*? repeat, *p q* concatenates, *p* | *q* is the union, *p* # *q* the difference of two classes, and (*p*) groups.

A name refers to a definition of the specification, *name*(p_1, ..., p_n) instantiates a template, and any other name is a HOL constant of one of the two types.

Classes and regular expressions share one namespace, and the elaborator infers the kind of each pattern: a union, difference or complement of classes is a class, everything else a regular expression.

pattern ‹*p*› **where** ... prints the elaboration of a pattern and its kind, in an optional environment of definitions. In a term, *PAT*‹*p*› is the elaboration of the pattern *p*, whose names are HOL constants: *Plus* (*Times* (*Pred* [(*97*, *122*)]) (*Star* (*Pred* [(*97*, *122*)]))) (*Pred* [(*95*, *95*)]).

2.3 What is defined

Lexing is defined once, for every alphabet, in the locale *symbolic*. It fixes nothing but the reading *sem* :: *'p* ⇒ *'ch* ⇒ *bool* of the atoms: *Pred p* matches the characters *c* with *sem p c*, and *symbolic.lang* is the language of a regular expression over atoms. There is no type class on *'ch* and no finite alphabet, so codepoints and bytes are instances, not assumptions.

A specification *sp* gives each mode its list of rules. One step of lexing follows the judgments *MaxPref* and *FirstToken* of Verbatim [9, Figs. 3–4]: the lexeme is the longest nonempty prefix of the input that a rule of the mode on top of the stack matches, and the winning rule is the first that matches it. Its action, which Verbatim does not have, gives the token class and the new stack.

The judgment *lexes sp is_newline st q s l r* st_f chains steps from the stack *st* and the position *q*, and relates the input *s* to its tokens with their spans and trivia *l*, the residue *r* where no step applies, and the stack st_f it stops in. A lexical error is not a failure but a nonempty residue, and an unterminated comment is an empty residue in a mode that is not final. Each step consumes a nonempty lexeme, so whatever the actions do to the stack, lexing ends:

$\exists\, l\ r\ st_f.$ *symbolic.lexes sem sp is_newline st q s l r* st_f

and with *symbolic.lexes_deterministic* the judgment is a function. Totality is the argument Verbatim gives for the termination of its lexer, whose recursive calls are on proper suffixes of the input [9, Sect. IV]. Determinism is its Uniqueness lemma [9, Sect. V-B], here for a judgment that also carries the stack.

The command takes the codepoints, natural numbers, as the alphabet, and classes of type *nat cls*, read by *in_cls*, as the atoms. For a lexer *L* it defines:

The specification *L.spec* $::\ \mu \Rightarrow \ldots$ *rule list*, by cases on the mode μ, with the equations *L.spec.simps*. A rule pairs a regular expression *match* with an action *act*, built by *mk_act* from the token class, the guard, the effect on the stack and the hook. The constants *L.c* of the definitions, the start mode *L.start*, the final modes *L.final* and the line ends *L.is_newline* come with it.

The lexer *L_lex*, which runs *cp_lexer* on the DFA tables of *L.spec* from the stack [*L.start*] and the first position q_0. The command evaluates the tables once for each distinct list of active rules, as constants *L_lex.tables_k* whose code equations are the evaluated tables, so modes with the same rules share them; *L_lex.index* maps each mode to its tables. The code equations are proved by evaluation, and only the generated code depends on them. *L_lex.ok w* holds if *L_lex* consumes all of *w* and stops in a final mode.

L_lex.correct: on an input of codepoints, *L_lex* computes the judgment,

$(\forall\, c \in$ *set w*. $c \leq$ *0x10FFFF*$) \Longrightarrow$
L_lex w $= (l,\ r,\ st_f) \longleftrightarrow$
symbolic.lexes in_cls L.spec L.is_newline [*L.start*] q_0 *w l r* st_f

the instance of *cp_lexer_iff_lexes_in_range* for *L.spec*. With the option (***verbatim*** ‹*char*›) it is *char_lexer_iff_lexes*, the judgment of *char_spec L.spec* on bytes, with no condition on the input.

The command reports each table with its modes, its size and the time of its

evaluation. It warns at a rule that wins in no state of a mode's tables, as rules of higher priority match all its lexemes; (***verbatim*** ‹*no_check*›) drops the warning.

2.4 Example

Words and numbers, with nested comments in an exclusive mode:

```
datatype ctok = Word | Num

lexer cmt :: ‹ctok›
  modes Code Comment
  exclusive Comment
  where letter = ‹[a−z]›
  rules
    ‹letter+› ⇒ Word
  | ‹[0−9]+› ⇒ Num
  | ‹[ \n]+› ⇒ skip
  | "(*" ⇒ skip, push Comment
  | <Comment> (
      "(*" ⇒ skip, push Comment
    | "*)" ⇒ skip, pop
    | ‹[^*(]+ | [*(]› ⇒ skip
  )
```

The comment rules are active only in *Comment*, and the stack counts the depth of nesting. *cmt.final* holds of *Code* only, so a text that ends inside a comment is not *cmt_lex.ok*.

A hook tells a negation from a subtraction by the token before it:

```
datatype otok = Id | Minus | Neg | LParen | RParen
datatype omode = Start | AfterOperand

fun after_tok :: ‹otok option ⇒ omode ⇒ omode› where
  ‹after_tok (Some Id) _ = AfterOperand›
| ‹after_tok (Some RParen) _ = AfterOperand›
| ‹after_tok _ _ = Start›

lexer ops :: ‹otok›
  modes ‹omode›
  after after_tok
  rules
    ‹[a−z]+› ⇒ Id | "(" ⇒ LParen | ")" ⇒ RParen | ‹[ ]+› ⇒ skip
  | <Start> "−" ⇒ Neg
  | <AfterOperand> "−" ⇒ Minus
```

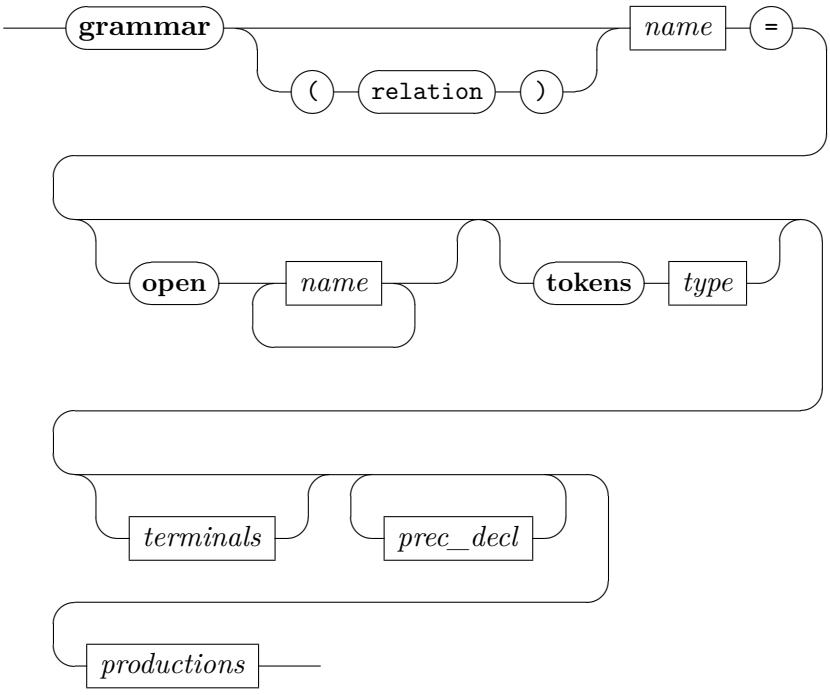
Both examples can be found in `lexer/tests/Lexer_Command_Tests.thy`.

3 Grammars and parsers

A grammar is a value of type (*'n*, *'tok*) *is_cfg*: a list of productions over nonterminals *'n* and token classes *'tok* with a start symbol, in which every symbol on a right-hand side carries an indentation relation *irel* (`grammar/model/Indentation_Sensitive.thy`, after Adams [1]). Its language *ilang G* is a set of token words, lists of records (*'tok*, *'ch*, *'m*) *token_scheme*, of which a parser reads the class *tok* and the column *s_col* of the *span*; the other fields and any extension *'m* of the record pass through to the semantic actions. A grammar may use the nonterminals of an earlier one and parametric nonterminals such as *Sep1*(*X*, *S*), with the postfix operators $*$ $+$ *?* for lists and options; both are expanded before anything is defined, so the result is an ordinary closed grammar.

This meaning is a set of derivations, not a program (Section 3.6), and the command proves the parser it generates sound for it. The parser runs the LR automaton of the grammar under one token of lookahead with layout constraints in its lookahead. This static check of the tables is evaluated when the parser is defined and its result is used to discharge the determinism premise for completeness. Thus parsers for LALR(1) grammars are automatically proven correct.

3.1 Specifications



grammar $G = \ldots$

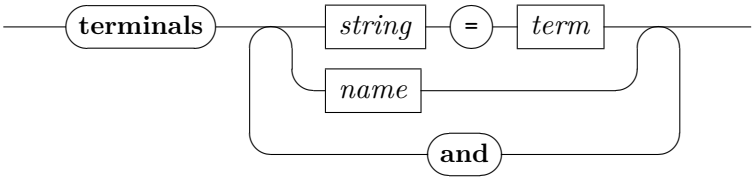
checks the specification and defines the grammar constant G; **grammar** (*relation*) $G = \ldots$ also defines the relation. The first nonterminal is the start symbol. **open** $a_1 \ \ldots \ a_n$ lets the productions refer to the nonterminals of earlier grammars without qualifier (Section 3.3), and the declarations *prec_decl* restrict the trees of an ambiguous grammar (Section 3.2).

tokens T

fixes the token type. T is a ground instance of ($'tok$, $'ch$, $'m$) *token_scheme*: the plain record ($'tok$, $'ch$) *token*, or an extension **record** *my_token* = ‹(*my_tok*, *char*) *token*› + ... Another type T abbreviates (T, *unit*) *token*, and without the clause the token classes are strings.

---

terminals

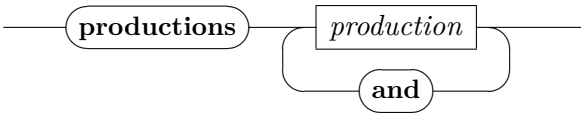


terminals *"s"* = t **and** $c \ \ldots$

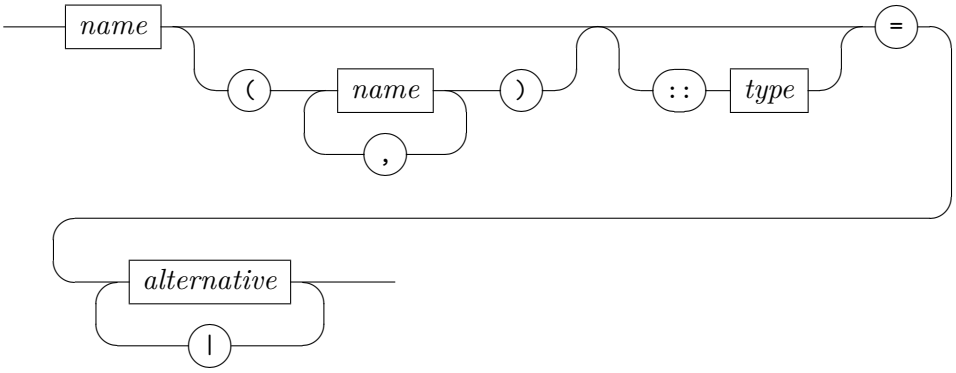
declares the terminals. A *fixed spelling* *"s"* = t lets *"s"* stand for the token class t in the productions, and binds no value. A bare name c declares a *token class*, written by its name in the productions, whose value is the whole token record. A name used in a production that is neither a nonterminal nor declared is read as a term of the token class type.

---

productions



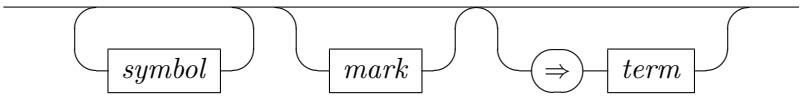
production



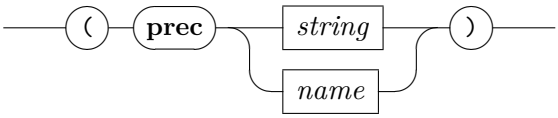
$A :: \langle\tau\rangle = alt_1 \mid \ldots \mid alt_n$
: declares the nonterminal A with *carrier* τ, the type of its values. The ascription can be omitted when the actions determine it. A production with parameters is a template (Section 3.4).

---

alternative



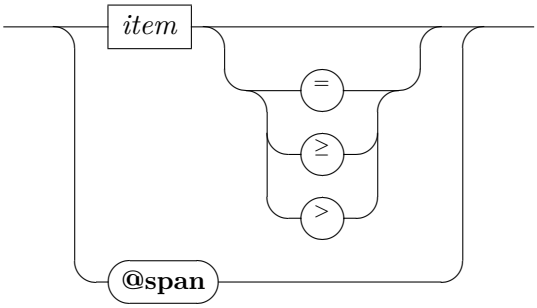
mark



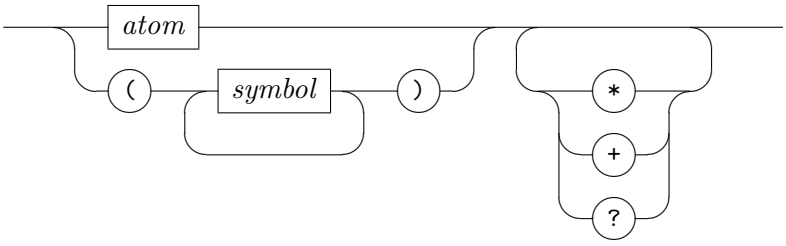
$X_1 \ldots X_k$ (**prec** q) $\Rightarrow f$
: is an alternative: a sequence of symbols, an optional mark and an optional action. The action f computes the value of the alternative from the values of its symbols, in order: $f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_k \Rightarrow \tau_A$. An alternative without action has exactly one symbol with a value, which it passes on; one without symbols needs an action. The mark gives the alternative a precedence level (Section 3.2). $\Rightarrow$ may be written `=>`.

---

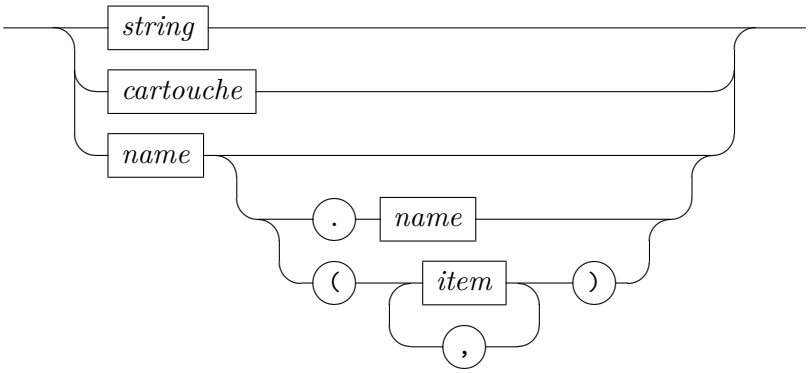
symbol



item



atom



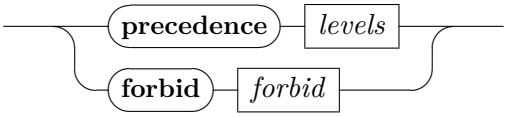
X, $X^{=}$, $X^{\geq}$, $X^{>}$
: is a symbol X with its annotation. X is a nonterminal, a fixed spelling, a token class, an inline token class ‹t›, a nonterminal $a.X$ of an earlier grammar a (Section 3.3), an instance of a template, or a group with an operator. A nonterminal contributes its value, a token class the token record. The annotation of a symbol is its indentation relation: $^{=}$ for *Eq*, $^{\geq}$ for *Ge* and $^{>}$ for *Gt*, written `^=`, `^>=` and `^>` in ASCII; without annotation it is *Any*. This departs from the convention of [1, Sect. 3.4], where an unannotated nonterminal defaults to = and an unannotated terminal to >.

@*span*
: is not a grammar symbol but a value of type *'ch span* among those of

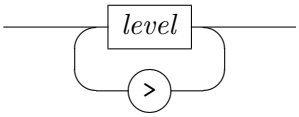
the symbols, where it is written: the span of the alternative's yield, from the line, column and offset of its first token to the end of its last, with their concatenated texts (*word_span*). It needs a symbol that derives no empty word, is written once per alternative, takes no annotation and stands at the top level of the alternative.

3.2 Precedence

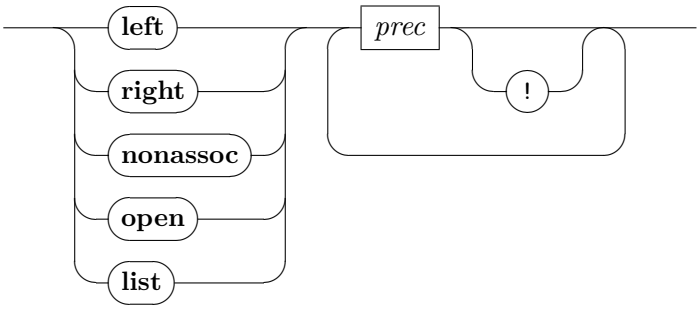
prec_decl



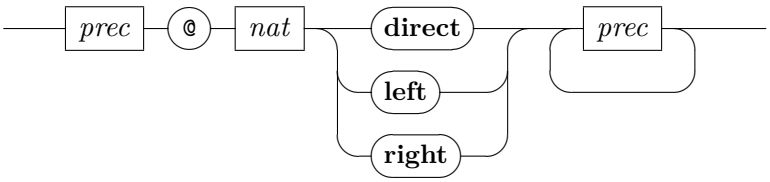
levels



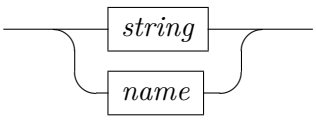
level



forbid



prec



precedence right *"^"* > **left** *"*"* *"/"* > **left** *"+"* *"−"* > **nonassoc** *"=="* declares levels, > binding tighter. A production has the level of its mark

(**prec** *NAME*), else that of its last terminal, written as its spelling or its name; the name of a level need not be a terminal, as in *"−" E* (**prec** *UMINUS*). This is the convention of yacc's *%left* and *%prec* in ocamlyacc [14] and Menhir [19]. Several declarations merge: a name in two of them is one level, the order is their transitive closure, and a cycle is an error, as for the priority chains of [3, Sect. 3.3, Def. 2]. Two levels that no declaration orders stay ambiguous, and the parser reports the conflict.

left, **right** and **nonassoc** have the usual readings. **open** leaves the left operand of its productions unconstrained, for forms that end in an open construct, and **list** does not order the productions of the level against each other, for flat chains. A prefix form is allowed on the right of a tighter operator, as in $a * - b$; ! on a name forbids that. **left** and **right** are the associativity relations of [3, Def. 4], and the levels become patterns after [3, Tables 1–2]; **open**, **list** and ! are not in the paper. **forbid** p @ k **right** c says what no level says: c is not on the right spine of child k of p (**left**: the left spine, **direct**: the child itself). **direct** is a pattern [3, Def. 5]; the spine forms exclude operators at any depth, which one-level patterns cannot [3, Sect. 2.2, Fig. 2].

These declarations restrict the trees, not the grammar: they are a filter on the parse trees of G [13, Defs. 4.1, 4.8], given as a tree automaton [2, Sects. 1, 3.1]. A parser of G is built from *p_gram* = *restricted G p_prec*, and its theorems are stated against G and the trees that satisfy *prec_ok*, or against the language *prec_lang*. Such a theory imports `TIGER_Grammar.LR_Prec_Parse`. The relation of **grammar** (*relation*) does not read the declarations, so no theorem is stated against it.

3.3 Composition

A symbol $a.X$, with a an earlier grammar and X one of its nonterminals, uses X in another grammar b. The productions of a reachable from X are copied into b, renaming each nonterminal Y of a to $a.Y$; a copy $c.Z$ that a made itself keeps its name, so it is copied once however it is reached.

b is then a closed grammar, and $a.X$ one of its nonterminals. In the names of constants and facts the dot is replaced with an underscore, for example *b___dec_a_X*, or *b_a_X*.

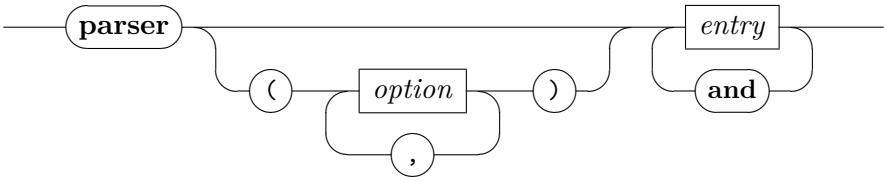
If a and b both have a relation, *b.a_X_iff*: $b_a_X\ i\ w\ x \longleftrightarrow a_X\ i\ w\ x$ links them, so what a parser of b proves about $a.X$ is a fact about a.

grammar b = **open** a lets b write X for $a.X$: a bare name that is not a nonterminal, terminal or template parameter of b itself is $a.X$ if a defines X.

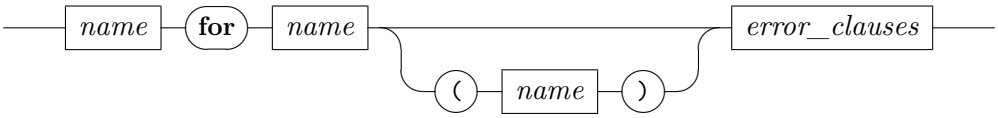
3.4 Templates and operators

A production with parameters, $P(X_1, \ldots, X_n) :: \tau = \ldots$, is a *template* and defines nothing by itself. A use $P(a_1, \ldots, a_n)$, with the parenthesis directly after the name, is an *instance*: the production with each X_i replaced by a_i, an ordinary nonterminal made once for each distinct list of arguments. An argument is a symbol, a symbol with an operator or a group of symbols. The actions of the template see the right-hand side after the replacement, so a parameter bound to a fixed spelling binds no value, and the type variables of the carrier are inferred per instance. A template of an earlier grammar g is used as $g.P(\ldots)$, or as $P(\ldots)$ if no other template has that name.

3.5 Parsers



entry

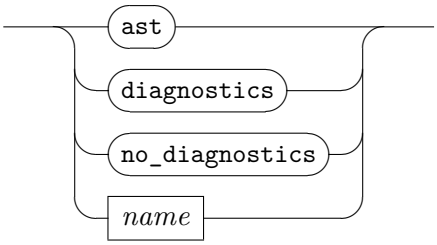


parser *p* **for** *G*
: defines $p :: \mathit{word} \Rightarrow \tau_S\ \mathit{option}$ for the start symbol S of G, with soundness and completeness theorems (Section 3.6).

parser *p* **for** *G* (*A*)
: parses from the nonterminal A instead: $p :: \mathit{word} \Rightarrow \tau_A\ \mathit{option}$, for the grammar $G(\!|\mathit{istart} := ''A''|\!)$ with the same productions and the start A. Its theorems are those of the start symbol with A for S.

parser *p* **for** *G* **and** *q* **for** *G* (*A*)
: defines several parsers for one grammar. They share its decoders, its relation and the lemmas their theorems are proved with; each has its own tables. The clauses *error_clauses* are about rejected words (Section 3.7).

option



`ast`
: defines $p :: word \Rightarrow (\ldots)\ ast\ option$, which returns the raw parse tree and evaluates no semantic action.

`diagnostics` | `no_diagnostics`
: turns the report of a rejected word, *p_diag*, on (default) or off (Section 3.7).

name
: selects the parser generator, by default the most recently declared one (i.e. *LR*).

3.6 What is defined

The language of a grammar is defined by derivations in which every symbol has an indentation. The indentation of a terminal is the column of its token. A nonterminal has no token of its own; its indentation is free up to the annotations. Expanding a nonterminal at indentation i by a production $A = {X_1}^{r1} \ldots {X_n}^{rn}$ chooses for each symbol an indentation j_k with *irel_sat* $r_k\ j_k\ i$:

$$irel_sat\ Eq\ j\ i \longleftrightarrow j = i \qquad irel_sat\ Gt\ j\ i \longleftrightarrow i < j$$
$$irel_sat\ Ge\ j\ i \longleftrightarrow i \leq j \qquad irel_sat\ Any\ j\ i \longleftrightarrow True$$

So $^{=}$ aligns a symbol with the nonterminal being expanded, $^{>}$ indents it further, $^{\geq}$ allows both, and no annotation places no constraint. A block of aligned statements is $Block = Stmt^{=} \mid Block^{=}\ Stmt^{=}$. A grammar without annotations is an ordinary context-free grammar, and *ilang* G is the set of words derived in this way from the start symbol at indentation *0*. This is the rewrite relation of Adams [1, Sect. 4], with his four relations $=$, $>$, $\geq$ and $\star$, the last written as no annotation.

For **grammar** G with nonterminals A of carrier τ_A and start symbol S the command defines:

The grammar $G :: (string, 'tok)\ is_cfg$ with its definition *G_def*. The nonterminals are represented by their names.

A decoder $G___dec_A :: (\ldots)\ ast \Rightarrow \tau_A\ option$ for each nonterminal, with the equations $G___dec_A.simps$, which evaluates the actions over a parse tree.

With (*relation*), an inductive predicate $G_A :: nat \Rightarrow word \Rightarrow \tau_A \Rightarrow bool$ for each nonterminal, with one introduction rule per alternative: $G_A\ i\ w\ x$ holds if w derives from A at indentation i with the value x. The facts are *G.intros*, *G.inducts* and *G.lang*: $(\exists\, x.\ G_S\ 0\ w\ x) \longleftrightarrow w \in ilang\ G$. For a grammar without annotations the indentation argument is dropped.

For **parser** p **for** G it defines:

The parser $p\ w = Option.bind\ (dparse\ G\ Shn\ w)\ G___dec_S$, where *dparse* is the verified driver of `grammar/lr/LR_Driver.thy` and *Shn* a fresh start symbol. The code equation of p runs on tables that are computed once, when the code is loaded, by a faster driver proved equal to *dparse*; for a grammar without annotations this is a table-driven LR driver on a stack of state numbers. With **declare** [[*parser_literal_tables*]] the tables are evaluated when the parser is defined instead.

p.sound and *p.complete* relate the parser to the language:

$$p.sound\text{: } p\ w = Some\ x \Longrightarrow w \in ilang\ G$$
$$p.complete\text{: } w \in ilang\ G \Longrightarrow \exists\, x.\ p\ w = Some\ x$$

For a grammar with a relation, to the relation of the start symbol:

$$p.sound\text{: } p\ w = Some\ x \Longrightarrow G_S\ 0\ w\ x$$
$$p.complete\text{: } G_S\ 0\ w\ x \Longrightarrow p\ w = Some\ x$$

and the language forms are *p.sound_lang* and *p.complete_lang*. With (*ast*) they are about the parse trees of G:

$$p.sound\text{: } p\ w = Some\ t \Longrightarrow ast_of\ G\ (Nt\ (istart\ G))\ (istart_ind\ G)\ t\ w$$
$$p.complete\text{: } wf_ptree\ G\ pt \Longrightarrow proot\ pt = (Nt\ (istart\ G), istart_ind\ G) \Longrightarrow p\ (pyield\ pt) = Some\ (erase\ pt)$$

that is, t is (the erasure of) a parse tree of w, and every parse tree of a word is the one p returns, which is therefore unique.

Soundness holds outright. Completeness needs the determinism of the driver on the grammar, *mdet G Shn*, and the totality of the decoders. For determinism the command evaluates *rlalr_conflicts G Shn*, the LALR(1) conflict analysis extended with the layout constraints. If there are no conflicts, *mdet* follows; otherwise the command lists the conflicting states and keeps *mdet* as a premise of *p.complete*. The check is sufficient but not necessary: an ambiguous grammar always fails it, a deterministic one may fail it too. The parser is defined either way, and **thm** *p.complete* shows what has been established.

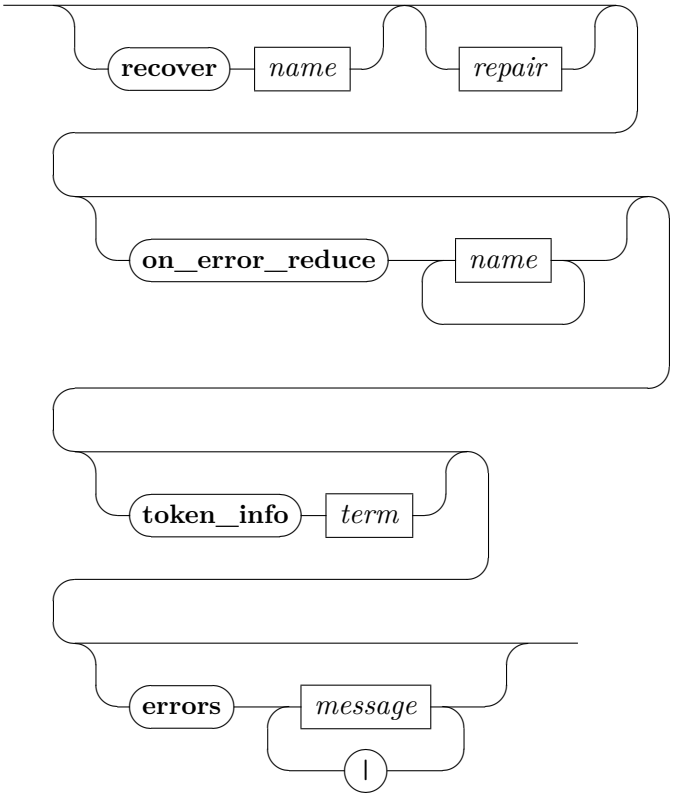
The checks are proved by evaluation (*Code_Runtime.dynamic_holds_conv*), so an unconditional *p.complete* trusts the code generator; *p.sound* does not depend on evaluation. The code equation of *p* does: the checks that let the faster driver stand for *dparse* are evaluated too. Jourdan, Pottier and Leroy also check their automaton by computation, running their validator with Coq's virtual machine [11, Sect. 5].

3.7 Diagnostics and recovery

p says only *None* on a rejected word. Unless **parser** (*no_diagnostics*) turns it off, each parser also defines *p_diag*, with *p_diag w* = *Inr x* exactly when *p w* = *Some x* (*p_diag.Inr_iff*) and otherwise *Inl e*, a report: the token where the word failed, the expected terminals with their columns, the stack with its spans and any repair. *perr_message* renders it. A bracket that is not closed is named as such, and a long list of expected terminals is shortened to the nonterminals that may start there.

No theorem is about the *Inl* side, by design: a report can be wrong, but no word changes its status. Recovery is built the same way. It rewrites the word, unverified, and the verified *p* parses the result, so all its theorems apply to the tree. None of this changes *p* or its tables, and all of it runs only after a failure.

error_clauses

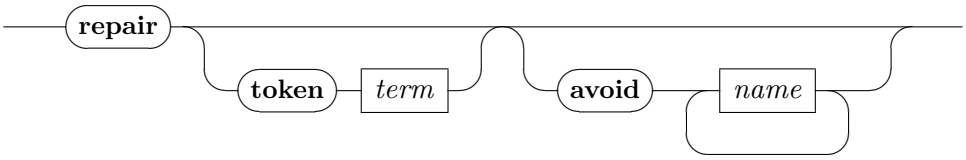


recover *e*
: adds *p_recover* for a grammar in which the terminal *e* stands for broken input, with ordinary productions such as $Expr = \ldots \mid ERR \Rightarrow EErr$. $p_recover\ w = (errs,\ p\ w')$: where w fails, the stack is popped to a state that can shift *e*, tokens are skipped up to one that continues, and the stretch becomes one *e*, as with the *error* token of ocamlyacc [14, Sect. 17.4.4], which skips to three tokens that continue.

on_error_reduce *A B*
: lets the report of a rejected word come after reducing productions of *A* or *B* [18, Sect. 4], so it names the construct that was complete.

token_info *f*
: gives the record extension for a token class, for the sentences of **errors** with extended token records.

repair

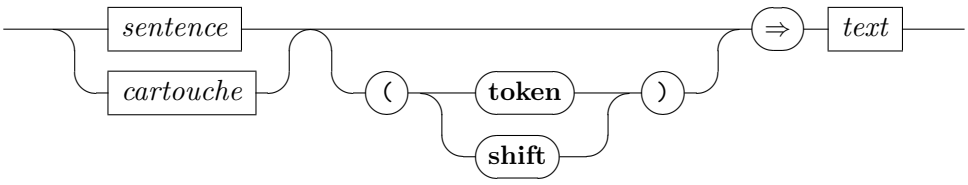


repair

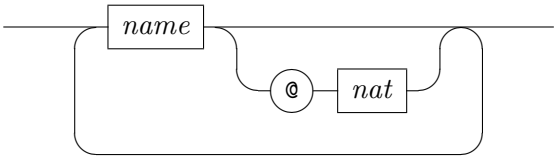
adds *p_recover* for any grammar: at an error it searches for a least-cost sequence of inserted and deleted tokens, ranked by how far parsing goes on, as in CPCT+ [8]. A budget of search nodes bounds it. **repair token** *f* supplies inserted tokens whose text an action reads, and **repair avoid** *A B* ranks repairs that insert *A* or *B* lower.

---

message



sentence



errors *s* ⇒ ‹*m*› | ...

attaches messages to error states by example [10, 18]. The sentence *s* is written with the spellings or names of terminals, each with an optional column after @, or as a cartouche with a term of the word type. It fails at its last token, or at its end. The message replaces the first line of the report of every word that fails in the same state, or with (*token*) in the same state at the same terminal; $*0* to $*9* stand for the text of the stack entries from the top. The keys are computed from the sentences, so an edit of the grammar that renumbers the states needs no change of the block.

3.8 Example

A statement is an identifier or a *while* header followed by a block indented further than the header, the statements of a block aligned. The value is the number of statements:

```
datatype tok = While | Colon | Id

grammar (relation) py =
  tokens ‹tok›
  terminals "while" = ‹While› and ":" = ‹Colon›
  productions
    Prog :: ‹nat› = Block
  and
    Block :: ‹nat›
      = Stmt=
      | Block= Stmt=  ⇒ ‹(+)›
  and
    Stmt :: ‹nat›
      = Id=                    ⇒ ‹λ_. 1›
      | "while"= Id> ":"> Block>  ⇒ ‹λ_ n. n + 1›

parser py_parse for py
```

Id is a token class, so the first action of *Stmt* takes its token record and ignores it. *Block* without annotation leaves the column of the outermost block free, $Block^{>}$ after a header demands a deeper one, and $Block^{=}\ Stmt^{=}$ aligns the members. The introduction rules of *py_Stmt*, writing $\langle t;\ sp\rangle$ for the token $(\!|tok = t,\ span = sp|\!)$, are

$$\frac{s_col\ sp = i}{py_Stmt\ i\ [\langle Id;\ sp\rangle]\ 1}$$

$$\frac{py_Block\ j\ w\ v \qquad s_col\ sp_1 = i \qquad i < s_col\ sp_2 \qquad i < s_col\ sp_3 \qquad i < j}{py_Stmt\ i\ (\langle While;\ sp_1\rangle\ \#\ \langle Id;\ sp_2\rangle\ \#\ \langle Colon;\ sp_3\rangle\ \#\ w)\ (v + 1)}$$

The grammar has no conflicts, so completeness is unconditional:

$$py_parse.complete\colon\ py_Prog\ 0\ w\ x \Longrightarrow py_parse\ w = Some\ x$$

With *tu c t* a token of class *t* at column *c*:

```
lemma ‹py_parse [tu 1 While, tu 7 Id, tu 8 Colon, tu 3 Id, tu 3 Id, tu 1 Id] =
Some 4›
  by eval

lemma ‹py_parse [tu 1 While, tu 7 Id, tu 8 Colon, tu 3 Id, tu 4 Id] = None›
  by eval
```

More examples are in `grammar/tests/Regression_Tests.thy`.

References

[1] M. D. Adams. Principled parsing for indentation-sensitive languages: Revisiting Landin's offside rule. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013)*, pages 511–522. ACM, 2013.

[2] M. D. Adams and M. Might. Restricting grammars with tree automata. *Proc. ACM Program. Lang.*, 1(OOPSLA):82:1–82:25, 2017.

[3] A. Afroozeh, M. van den Brand, A. Johnstone, E. Scott, and J. Vinju. Safe specification of operator precedence rules. In *Software Language Engineering (SLE 2013)*, volume 8225 of *LNCS*, pages 137–156. Springer, 2013.

[4] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoret. Comput. Sci.*, 155:291–319, 1996.

[5] A. D. Brucker and B. Wolff. ML-Lex and ML-Yacc for Isabelle. *Archive of Formal Proofs*, June 2026. https://isa-afp.org/entries/Isabelle_Lex-Yacc.html, Formal proof development.

[6] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[7] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *POPL*, 2014.

[8] L. Diekmann and L. Tratt. Don't panic! Better, fewer, syntax errors for LR parsers. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *LIPIcs*, pages 6:1–6:32, 2020.

[9] D. Egolf, S. Lasser, and K. Fisher. Verbatim: A verified lexer generator. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 92–100. IEEE, 2021.

[10] C. L. Jeffery. Generating LR syntax error messages from examples. *ACM Trans. Program. Lang. Syst.*, 25(5):631–640, 2003.

[11] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *Programming Languages and Systems (ESOP 2012)*, volume 7211 of *LNCS*, pages 397–416. Springer, 2012.

[12] M. Keil and P. Thiemann. Symbolic solving of extended regular expression inequalities. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS 2014)*, volume 29 of *LIPIcs*, pages 175–186, 2014.

[13] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. Technical Report P9426, Programming Research Group, University of Amsterdam, Dec. 1994. Also in Proc. ASMICS Workshop on Parsing Theory, Milano, 1994, https://eelcovisser.org/publications/1994/KlintV94.pdf.

[14] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml System, Release 5.5: Documentation and User's Manual*, June 2026. https://ocaml.org/manual/.

[15] M. E. Lesk and E. Schmidt. Lex – a lexical analyzer generator. Computing Science Technical Report 39, Bell Laboratories, 1975.

[16] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives reexamined. *J. Funct. Program.*, 19(2):173–190, 2009.

[17] V. Paxson, W. Estes, and J. Millaway. *Lexical Analysis With Flex*, 2026. Edition for flex 2.6.9, https://westes.github.io/flex/manual/.

[18] F. Pottier. Reachability and error diagnosis in LR(1) parsers. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*, pages 88–98. ACM, 2016.

[19] F. Pottier and Y. Régis-Gianas. *Menhir Reference Manual*, 2026. Version 20260209, https://gallium.inria.fr/~fpottier/menhir/manual.html.

[20] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2026. Isabelle2026, https://isabelle.in.tum.de/documentation.html.