# Menhir Reference Manual
**(version 20160526)**

François Pottier   Yann Régis-Gianas
INRIA
{Francois.Pottier, Yann.Regis-Gianas}@inria.fr

## Contents

## 1. Foreword

Menhir is a parser generator. It turns high-level grammar specifications, decorated with semantic actions expressed in the OCaml programming language [15], into parsers, again expressed in OCaml. It is based on Knuth's LR(1) parser construction technique [12]. It is strongly inspired by its precursors: `yacc` [10], `ML-Yacc` [19], and `ocamlyacc` [15], but offers a large number of minor and major improvements that make it a more modern tool.

This brief reference manual explains how to use Menhir. It does not attempt to explain context-free grammars, parsing, or the LR technique. Readers who have never used a parser generator are encouraged to read about these ideas first [1, 2, 7]. They are also invited to have a look at the `demos` directory in Menhir's distribution.

Potential users of Menhir should be warned that Menhir's feature set is not completely stable. There is a tension between preserving a measure of compatibility with `ocamlyacc`, on the one hand, and introducing new ideas, on the other hand. Some aspects of the tool, such as the error handling mechanism, are still potentially subject to incompatible changes: for instance, in the future, the current error handling mechanism (which is based on the **error** token, see §10) could be removed and replaced with an entirely different mechanism.

There is room for improvement in the tool and in this reference manual. Bug reports and suggestions are welcome!

## 2. Usage

Menhir is invoked as follows:

`menhir` *option* ... *option filename* ... *filename*

Each of the file names must end with `.mly` and denotes a partial grammar specification. These partial grammar specifications are joined (§5.1) to form a single, self-contained grammar specification, which is then processed. A number of optional command line switches allow controlling many aspects of the process.

`--base` *basename*. This switch controls the base name of the `.ml` and `.mli` files that are produced. That is, the tool will produce files named *basename*`.ml` and *basename*`.mli`. Note that *basename* can contain occurrences of the / character, so it really specifies a path and a base name. When only one *filename* is provided on the command line, the default *basename* is obtained by depriving *filename* of its final `.mly` suffix. When multiple file names are provided on the command line, no default base name exists, so that the `--base` switch *must* be used.

`--comment`. This switch causes a few comments to be inserted into the OCaml code that is written to the `.ml` file.

`--compare-errors` *filename1* `--compare-errors` *filename2*. Two such switches must always be used in conjunction so as to specify the names of two `.messages` files, *filename1* and *filename2*. Each file is read and internally translated to a mapping of states to messages. Menhir then checks that the left-hand mapping is a subset of the right-hand mapping. This feature is typically used in conjunction with `--list-errors` to check that *filename2* is complete (that is, covers all states where an error can occur). For more information, see §11.

`--compile-errors` *filename*. This switch causes Menhir to read the file *filename*, which must obey the `.messages` file format, and to compile it to an OCaml function that maps a state number to a message. The OCaml code is sent to the standard output channel. At the same time, Menhir checks that the collection of input sentences in the file *filename* is correct and irredundant. For more information, see §11.

`--coq`. This switch causes Menhir to produce Coq code. See §12.

`--coq-no-actions`. (Used in conjunction with `--coq`.) This switch causes the semantic actions present in the `.vy` file to be ignored and replaced with `tt`, the unique inhabitant of Coq's `unit` type. This feature can be used to test the Coq back-end with a standard grammar, that is, a grammar that contains OCaml semantic actions. Just rename the file from `.mly` to `.vy` and set this switch.

`--coq-no-complete`. (Used in conjunction with `--coq`.) This switch disables the generation of the proof of completeness of the parser (§12). This can be necessary because the proof of completeness is possible only if the grammar has no conflict (not even a benign one, in the sense of §6.1). This can be desirable also because, for a complex grammar, completeness may require a heavy certificate and its validation by Coq may take time.

`--depend`. This switch causes Menhir to generate dependency information for use in conjunction with `make`. When invoked in this mode, Menhir does not generate a parser. Instead, it examines the grammar specification and prints a list of prerequisites for the targets *basename*`.cm[iox]`, *basename*`.ml`, and *basename*`.mli`. This list is intended to be textually included within a `Makefile`. It is important to note that *basename*`.ml` and *basename*`.mli` can have `.cm[iox]` prerequisites. This is because, when the `--infer` switch is used, Menhir infers types by invoking `ocamlc`, and `ocamlc` itself requires the OCaml modules that the grammar specification depends upon to have been compiled first. An end user who uses `ocamlbuild` does not need this switch.

When in `--depend` mode, Menhir computes dependencies by invoking `ocamldep`. The command that is used to run `ocamldep` is controlled by the `--ocamldep` switch.

`--dump`. This switch causes a description of the automaton to be written to the file *basename*`.automaton`.

`--echo-errors` *filename*. This switch causes Menhir to read the `.messages` file *filename* and to produce on the standard output channel just the input sentences. (That is, all messages, blank lines, and comments are filtered out.) For more information, see §11.

`--explain`. This switch causes conflict explanations to be written to the file *basename*`.conflicts`. See also §6.

`--external-tokens` *T*. This switch causes the definition of the `token` type to be omitted in *basename*`.ml` and *basename*`.mli`. Instead, the generated parser relies on the type $T$`.token`, where $T$ is an OCaml module name. It is up to the user to define module $T$ and to make sure that it exports a suitable `token` type. Module $T$ can be hand-written. It can also be automatically generated out of a grammar specification using the `--only-tokens` switch.

`--fixed-exception`. This switch causes the exception `Error` to be internally defined as a synonym for `Parsing.Parse_error`. This means that an exception handler that catches `Parsing.Parse_error` will also catch the generated parser's `Error`. This helps increase Menhir's compatibility with `ocamlyacc`. There is otherwise no reason to use this switch.

`--graph`. This switch causes a description of the grammar's dependency graph to be written to the file *basename*`.dot`. The graph's vertices are the grammar's nonterminal symbols. There is a directed edge from vertex $A$ to vertex $B$ if the definition of $A$ refers to $B$. The file is in a format that is suitable for processing by the *graphviz* toolkit.

`--unused-token` *symbol*. This switch suppresses the warning that is normally emitted when Menhir finds that the terminal symbol *symbol* is unused.

`--unused-tokens`. This switch suppresses all of the warnings that are normally emitted when Menhir finds that some terminal symbols are unused.

`--infer`. This switch causes the semantic actions to be checked for type consistency *before* the parser is generated. This is done by invoking the OCaml compiler. Use of `--infer` is **strongly recommended**, because it helps obtain consistent, well-located type error messages, especially when advanced features such as Menhir's standard library or **%inline** keyword are exploited. One downside of `--infer` is that the OCaml compiler usually needs to consult a few `.cm[iox]` files. This means that these files must have been created first, requiring `Makefile` changes and use of the `--depend` switch. The file `demos/obsolete/Makefile.shared` suggests how to deal with this difficulty. A better option is to avoid `make` altogether and use `ocamlbuild`, which has built-in knowledge of Menhir. Using `ocamlbuild` is **strongly recommended**!

`--inspection`. This switch requires `--table`. It causes Menhir to generate not only the monolithic and incremental APIs (§9.1, §9.2), but also the inspection API (§9.3). Activating this switch causes a few more tables to be produced, resulting in somewhat larger code size.

`--interpret`. This switch causes Menhir to act as an interpreter, rather than as a compiler. No OCaml code is generated. Instead, Menhir reads sentences off the standard input channel, parses them, and displays outcomes. This switch can be usefully combined with `--trace`. For more information, see §8.

`--interpret-error`. This switch is analogous to `--interpret`, except Menhir expects every sentence to cause an error on its last token, and displays information about the state in which the error is detected, in the `.messages` file format. For more information, see §11.

`--interpret-show-cst`. This switch, used in conjunction with `--interpret`, causes Menhir to display a concrete syntax tree when a sentence is successfully parsed. For more information, see §8.

`--list-errors`. This switch causes Menhir to produce (on the standard output channel) a complete list of input sentences that cause an error, in the `.messages` file format. For more information, see §11.

`--log-automaton` *level*. When *level* is nonzero, this switch causes some information about the automaton to be logged to the standard error channel.

`--log-code` *level*. When *level* is nonzero, this switch causes some information about the generated OCaml code to be logged to the standard error channel.

`--log-grammar` *level*. When *level* is nonzero, this switch causes some information about the grammar to be logged to the standard error channel. When *level* is 2, the *nullable*, *FIRST*, and *FOLLOW* tables are displayed.

`--no-inline`. This switch causes all **%inline** keywords in the grammar specification to be ignored. This is especially useful in order to understand whether these keywords help solve any conflicts.

`--no-stdlib`. This switch causes the standard library *not* to be implicitly joined with the grammar specifications whose names are explicitly provided on the command line.

`--ocamlc` *command*. This switch controls how `ocamlc` is invoked (when `--infer` is used). It allows setting both the name of the executable and the command line options that are passed to it.

`--ocamldep` *command*. This switch controls how `ocamldep` is invoked (when `--depend` is used). It allows setting both the name of the executable and the command line options that are passed to it.

`--only-preprocess`. This switch causes the grammar specifications to be transformed up to the point where the automaton's construction can begin. The grammar specifications whose names are provided on the command line are joined (§5.1); all parameterized nonterminal symbols are expanded away (§5.2); type inference is performed, if `--infer` is enabled; all nonterminal symbols marked **%inline** are expanded away (§5.3). This yields a single, monolithic grammar specification, which is printed on the standard output channel.

`--only-tokens`. This switch causes the **%token** declarations in the grammar specification to be translated into a definition of the `token` type, which is written to the files *basename*`.ml` and *basename*`.mli`. No code is generated. This is useful when a single set of tokens is to be shared between several parsers. The directory `demos/calc-two` contains a demo that illustrates the use of this switch.

`--raw-depend`. This switch is analogous to `--depend`, except that `ocamldep`'s output is not postprocessed by Menhir; it is echoed without change. This switch is not suitable for direct use with `make`; it is intended for use with `omake` or `ocamlbuild`, which perform their own postprocessing. An end user who uses `ocamlbuild` does not need to mention this switch: `ocamlbuild` uses it automatically.

`--strict`. This switch causes several warnings about the grammar and about the automaton to be considered errors. This includes warnings about useless precedence declarations, non-terminal symbols that produce the empty language, unreachable non-terminal symbols, productions that are never reduced, conflicts that are not resolved by precedence declarations, and end-of-stream conflicts.

`--suggest-comp-flags`. This switch causes Menhir to print a set of suggested compilation flags, and exit. These flags are intended to be passed to the OCaml compilers (`ocamlc` or `ocamlopt`) when compiling and linking the parser generated by Menhir. What are these flags? In the absence of the `--table` switch, they are empty. When `--table` is set, these flags ensure that `MenhirLib` is visible to the OCaml compiler. If the support library `MenhirLib` was installed via `ocamlfind`, a `-package` directive is issued; otherwise, a `-I` directive is used.

`--suggest-link-flags-byte`. This switch causes Menhir to print a set of suggested link flags, and exit. These flags are intended to be passed to `ocamlc` when producing a bytecode executable. What are these flags? In the absence of the `--table` switch, they are empty. When `--table` is set, these flags ensure that `MenhirLib` is linked in. If the support library `MenhirLib` was installed via `ocamlfind`, a `-linkpkg` directive is issued; otherwise, the object file `menhirLib.cmo` is named.

`--suggest-link-flags-opt`. This switch causes Menhir to print a set of suggested link flags, and exit. These flags are intended to be passed to `ocamlopt` when producing a native code executable. What are these flags? In the absence of the `--table` switch, they are empty. When `--table` is set, these flags ensure that `MenhirLib` is linked in. If the support library `MenhirLib` was installed via `ocamlfind`, a `-linkpkg` directive is issued; otherwise, the object file `menhirLib.cmx` is named.

`--suggest-menhirLib`. This switch causes Menhir to print (the absolute path of) the directory where `MenhirLib` was installed. If `MenhirLib` was installed via `ocamlfind`, this is equivalent to calling `ocamlfind query menhirLib`.

`--suggest-ocamlfind`. This switch causes Menhir to print a Boolean value (i.e., either `true` or `false`), which indicates whether `MenhirLib` was installed via `ocamlfind`.

`--stdlib` *directory*. This switch controls the directory where the standard library is found. It allows overriding the default directory that is set at installation time. The trailing `/` character is optional.

`--table`. This switch causes Menhir to use its table-based back-end, as opposed to its (default) code-based back-end. When `--table` is used, Menhir produces significantly more compact and somewhat slower parsers. See §14 for a speed comparison.

The table-based back-end produces rather compact tables, which are analogous to those produced by `yacc`, `bison`, or `ocamlyacc`. These tables are not quite stand-alone: they are exploited by an interpreter, which is shipped as part of the support library `MenhirLib`. For this reason, when `--table` is used, `MenhirLib` must be made visible to the OCaml compilers, and must be linked into your executable program. The `-suggest-*` switches, described above, help do this.

The code-based back-end compiles the LR automaton directly into a nest of mutually recursive OCaml functions. In that case, `MenhirLib` is not required.

The incremental API (§9.2) and the inspection API (§9.3) are made available only by the table-based back-end.

`--timings`. This switch causes internal timing information to be sent to the standard error channel.

`--trace`. This switch causes tracing code to be inserted into the generated parser, so that, when the parser is run, its actions are logged to the standard error channel. This is analogous to `ocamlrun`'s `p=1` parameter, except this switch must be enabled at compile time: one cannot selectively enable or disable tracing at runtime.

`--update-errors` *filename*. This switch causes Menhir to read the `.messages` file *filename* and to produce on the standard output channel a new `.messages` file that is identical, except the auto-generated comments have been re-generated. For more information, see §11.

`--version`. This switch causes Menhir to print its own version number and exit.

$$
\begin{array}{rcl}
\textit{specification} & ::= & \textit{declaration} \ \ldots \ \textit{declaration} \ \textbf{\%\%} \ \textit{rule} \ \ldots \ \textit{rule} \ [\, \textbf{\%\%} \ \textit{OCaml code} \,] \\
\textit{declaration} & ::= & \textbf{\%\{} \ \textit{OCaml code} \ \textbf{\%\}} \\
 & & \textbf{\%parameter} < \textit{uid} : \textit{OCaml module type} > \\
 & & \textbf{\%token} \ [\, < \textit{OCaml type} > \,] \ \textit{uid} \ \ldots \ \textit{uid} \\
 & & \textbf{\%nonassoc} \ \textit{uid} \ \ldots \ \textit{uid} \\
 & & \textbf{\%left} \ \textit{uid} \ \ldots \ \textit{uid} \\
 & & \textbf{\%right} \ \textit{uid} \ \ldots \ \textit{uid} \\
 & & \textbf{\%type} < \textit{OCaml type} > \textit{lid} \ \ldots \ \textit{lid} \\
 & & \textbf{\%start} \ [\, < \textit{OCaml type} > \,] \ \textit{lid} \ \ldots \ \textit{lid} \\
 & & \textbf{\%on\_error\_reduce} \ \textit{lid} \ \ldots \ \textit{lid} \\
\textit{rule} & ::= & [\, \textbf{\%public} \,] \ [\, \textbf{\%inline} \,] \ \textit{lid} \ [\, \textbf{(} \ \textit{id}, \ \ldots, \textit{id} \,\textbf{)} \,] \ \textbf{:} \ [\, \textbf{|} \,] \ \textit{group} \ \textbf{|} \ \ldots \ \textbf{|} \ \textit{group} \\
\textit{group} & ::= & \textit{production} \ \textbf{|} \ \ldots \ \textbf{|} \ \textit{production} \ \textbf{\{} \ \textit{OCaml code} \ \textbf{\}} \ [\, \textbf{\%prec} \ \textit{id} \,] \\
\textit{production} & ::= & \textit{producer} \ \ldots \ \textit{producer} \ [\, \textbf{\%prec} \ \textit{id} \,] \\
\textit{producer} & ::= & [\, \textit{lid} = \,] \ \textit{actual} \\
\textit{actual} & ::= & \textit{id} \ [\, \textbf{(} \ \textit{actual}, \ \ldots, \textit{actual} \,\textbf{)} \,] \\
 & & \textit{actual} \ [\, \textbf{?} \mid \textbf{+} \mid \textbf{*} \,] \\
 & & \textit{group} \ \textbf{|} \ \ldots \ \textbf{|} \ \textit{group}
\end{array}
$$

**Figure 1.** Syntax of grammar specifications

## 3. Lexical conventions

The semicolon character (;) is treated as insignificant, just like white space. Thus, rules and producers (for instance) can be separated with semicolons if it is thought that this improves readability. Semicolons can be omitted otherwise.

Identifiers (*id*) coincide with OCaml identifiers, except they are not allowed to contain the quote (') character. Following OCaml, identifiers that begin with a lowercase letter (*lid*) or with an uppercase letter (*uid*) are distinguished.

Comments are C-style (surrounded with **/*** and ***/**, cannot be nested), C++-style (announced by **//** and extending until the end of the line), or OCaml-style (surrounded with **(*** and ***)**, can be nested). Of course, inside OCaml code, only OCaml-style comments are allowed.

OCaml type expressions are surrounded with $<$ and $>$. Within such expressions, all references to type constructors (other than the built-in *list*, *option*, etc.) must be fully qualified.

## 4. Syntax of grammar specifications

The syntax of grammar specifications appears in Figure 1. (For compatibility with `ocamlyacc`, some specifications that do not fully adhere to this syntax are also accepted.)

### 4.1 Declarations

A specification file begins with a sequence of declarations, ended by a mandatory **%%** keyword.

#### 4.1.1 Headers

A header is a piece of OCaml code, surrounded with **%{** and **%}**. It is copied verbatim at the beginning of the `.ml` file. It typically contains OCaml **open** directives and function definitions for use by the semantic actions. If a single grammar specification file contains multiple headers, their order is preserved. However, when two headers originate in distinct grammar specification files, the order in which they are copied to the `.ml` file is unspecified.

### 4.1.2 Parameters

A declaration of the form:

**%parameter** $<$ *uid* : *OCaml module type* $>$

causes the entire parser to become parameterized over the OCaml module *uid*, that is, to become an OCaml functor. The directory `demos/calc-param` contains a demo that illustrates the use of this switch.

If a single specification file contains multiple **%parameter** declarations, their order is preserved, so that the module name *uid* introduced by one declaration is effectively in scope in the declarations that follow. When two **%parameter** declarations originate in distinct grammar specification files, the order in which they are processed is unspecified. Last, **%parameter** declarations take effect before **%{** ... **%}**, **%token**, **%type**, or **%start** declarations are considered, so that the module name *uid* introduced by a **%parameter** declaration is effectively in scope in *all* **%{** ... **%}**, **%token**, **%type**, or **%start** declarations, regardless of whether they precede or follow the **%parameter** declaration. This means, in particular, that the side effects of an OCaml header are observed only when the functor is applied, not when it is defined.

### 4.1.3 Tokens

A declaration of the form:

**%token** [ $<$ *OCaml type* $>$ ] $uid_1, \ldots, uid_n$

defines the identifiers $uid_1, \ldots, uid_n$ as tokens, that is, as terminal symbols in the grammar specification and as data constructors in the *token* type. If an OCaml type $t$ is present, then these tokens are considered to carry a semantic value of type $t$, otherwise they are considered to carry no semantic value.

### 4.1.4 Priority and associativity

A declaration of one of the following forms:

**%nonassoc** $uid_1 \ldots uid_n$
**%left** $uid_1 \ldots uid_n$
**%right** $uid_1 \ldots uid_n$

attributes both a *priority level* and an *associativity status* to the symbols $uid_1, \ldots, uid_n$. The priority level assigned to $uid_1, \ldots, uid_n$ is not defined explicitly: instead, it is defined to be higher than the priority level assigned by the previous **%nonassoc**, **%left**, or **%right** declaration, and lower than that assigned by the next **%nonassoc**, **%left**, or **%right** declaration. The symbols $uid_1, \ldots, uid_n$ can be tokens (defined elsewhere by a **%token** declaration) or dummies (not defined anywhere). Both can be referred to as part of **%prec** annotations. Associativity status and priority levels allow shift/reduce conflicts to be silently resolved (§6).

### 4.1.5 Types

A declaration of the form:

**%type** $<$ *OCaml type* $>$ $lid_1 \ldots lid_n$

assigns an OCaml type to each of the nonterminal symbols $lid_1, \ldots, lid_n$. For start symbols, providing an OCaml type is mandatory, but is usually done as part of the **%start** declaration. For other symbols, it is optional. Providing type information can improve the quality of OCaml's type error messages.

A **%type** declaration may concern not only a nonterminal symbol, such as, say, `expression`, but also a fully applied parameterized nonterminal symbol, such as `list(expression)` or `separated_list(COMMA, option(expression))`.

### 4.1.6 Start symbols

A declaration of the form:

**%start** [ < *OCaml type* > ] $lid_1 \ldots lid_n$

declares the nonterminal symbols $lid_1, \ldots, lid_n$ to be start symbols. Each such symbol must be assigned an OCaml type either as part of the **%start** declaration or via separate **%type** declarations. Each of $lid_1, \ldots, lid_n$ becomes the name of a function whose signature is published in the `.mli` file and that can be used to invoke the parser.

### 4.1.7 Extra reductions on error

A declaration of the form:

**%on_error_reduce** $lid_1 \ldots lid_n$

marks the nonterminal symbols $lid_1, \ldots, lid_n$ as potentially eligible for reduction when an invalid token is found.

More precisely, this declaration affects the automaton as follows. Let us say that a production $lid \rightarrow \ldots$ is "reducible on error" if its left-hand symbol *lid* appears in a **%on_error_reduce** declaration. After the automaton has been constructed and after any conflicts have been resolved, in every state $s$, the following rule is applied:

> If the set of all productions that are ready to be reduced in state $s$ and are reducible on error is a singleton set $\{p\}$, then in state $s$ every error action is replaced with a reduction of the production $p$.

In other words, for every terminal symbol $t$, if the automaton's action table says: "in state $s$, when the next input symbol is $t$, fail", then this table entry is replaced with: "in state $s$, when the next input symbol is $t$, reduce production $p$".

If this rule fires in state $s$, then an error can never be detected in state $s$, since all error actions in state $s$ are replaced with reduce actions. Error detection is deferred: at least one reduction takes place before the error is detected. It is a "spurious" reduction: in a canonical LR(1) automaton, it would not take place.

An **%on_error_reduce** declaration does not affect the language that is accepted by the automaton. It does not affect the location where an error is detected. It is used to control in which state an error is detected. If used wisely, it can make errors easier to report, because they are detected in a state for which it is easier to write an accurate diagnostic message (§11.3).

Like a **%type** declaration, an **%on_error_reduce** declaration may concern not only a nonterminal symbol, such as, say, `expression`, but also a fully applied parameterized nonterminal symbol, such as `list(expression)` or `separated_list(COMMA, option(expression))`.

## 4.2 Rules

Following the mandatory **%%** keyword, a sequence of rules is expected. Each rule defines a nonterminal symbol *id*. (It is recommended that the name of a nonterminal symbol begin with a lowercase letter, so it falls in the category *lid*. This is in fact mandatory for the start symbols.) In its simplest form, a rule begins with the nonterminal symbol *id*, followed by a colon character (:), and continues with a sequence of production groups (§4.2.1). Each production group is preceded with a vertical bar character (|); the very first bar is optional. The meaning of the bar is choice: the nonterminal symbol *id* develops to either of the production groups. We defer explanations of the keyword **%public** (§5.1), of the keyword **%inline** (§5.3), and of the optional formal parameters **(** *id*, ..., *id* **)** (§5.2).

### 4.2.1 Production groups

In its simplest form, a production group consists of a single production (§4.2.2), followed by an OCaml semantic action (§4.2.1) and an optional **%prec** annotation (§4.2.1). A production specifies a sequence of terminal and nonterminal symbols that should be recognized, and optionally binds identifiers to their semantic values.

***Semantic actions*** A semantic action is a piece of OCaml code that is executed in order to assign a semantic value to the nonterminal symbol with which this production group is associated. A semantic action can refer to the (already computed) semantic values of the terminal or nonterminal symbols that appear in the production via the semantic value identifiers bound by the production.

For compatibility with `ocamlyacc`, semantic actions can also refer to unnamed semantic values via positional keywords of the form **$1**, **$2**, etc. This style is discouraged. Furthermore, as a positional keyword of the form **$i** is internally rewritten as *_i*, the user should not use identifiers of the form *_i*.

**%prec *annotations*** An annotation of the form **%prec** *id* indicates that the precedence level of the production group is the level assigned to the symbol *id* via a previous **%nonassoc**, **%left**, or **%right** declaration (§4.1.4). In the absence of a **%prec** annotation, the precedence level assigned to each production is the level assigned to the rightmost terminal symbol that appears in it. It is undefined if the rightmost terminal symbol has an undefined precedence level or if the production mentions no terminal symbols at all. The precedence level assigned to a production is used when resolving shift/reduce conflicts (§6).

***Multiple productions in a group*** If multiple productions are present in a single group, then the semantic action and precedence annotation are shared between them. This short-hand effectively allows several productions to share a semantic action and precedence annotation without requiring textual duplication. It is legal only when every production binds exactly the same set of semantic value identifiers and when no positional semantic value keywords (**$1**, etc.) are used.

### 4.2.2 Productions

A production is a sequence of producers (§4.2.3), optionally followed by a **%prec** annotation (§4.2.1). If a precedence annotation is present, it applies to this production alone, not to other productions in the production group. It is illegal for a production and its production group to both carry **%prec** annotations.

### 4.2.3 Producers

A producer is an actual (§4.2.4), optionally preceded with a binding of a semantic value identifier, of the form *lid* =. The actual specifies which construction should be recognized and how a semantic value should be computed for that construction. The identifier *lid*, if present, becomes bound to that semantic value in the semantic action that follows. Otherwise, the semantic value can be referred to via a positional keyword (**$1**, etc.).

### 4.2.4 Actuals

In its simplest form, an actual is just a terminal or nonterminal symbol *id*. If it is a parameterized non-terminal symbol (see §5.2), then it should be applied: *id***(** *actual*, ..., *actual* **)**.

An actual may be followed with a modifier (**?**, **+**, or *****). This is explained further on (see §5.2 and Figure 2).

An actual may also be an "anonymous rule". In that case, one writes just the rule's right-hand side, which takes the form *group* **|** ... **|** *group*. (This form is allowed only as an argument in an application.) This form is expanded on the fly to a definition of a fresh non-terminal symbol, which is declared **%inline**. For instance, providing an anonymous rule as an argument to *list*:

*list* **(** *e* = *expression*; *SEMICOLON* **{** *e* **}** **)**

is equivalent to writing this:

*list* **(** *expression_SEMICOLON* **)**

where the non-terminal symbol *expression_SEMICOLON* is chosen fresh and is defined as follows:

**%inline** *expression_SEMICOLON*:
    | *e* = *expression*; *SEMICOLON* **{** *e* **}**

# 5. Advanced features

## 5.1 Splitting specifications over multiple files

***Modules*** Grammar specifications can be split over multiple files. When Menhir is invoked with multiple argument file names, it considers each of these files as a *partial* grammar specification, and *joins* these partial specifications in order to obtain a single, complete specification.

This feature is intended to promote a form a modularity. It is hoped that, by splitting large grammar specifications into several "modules", they can be made more manageable. It is also hoped that this mechanism, in conjunction with parameterization (§5.2), will promote sharing and reuse. It should be noted, however, that this is only a weak form of modularity. Indeed, partial specifications cannot be independently processed (say, checked for conflicts). It is necessary to first join them, so as to form a complete grammar specification, before any kind of grammar analysis can be done.

This mechanism is, in fact, how Menhir's standard library (§5.4) is made available: even though its name does not appear on the command line, it is automatically joined with the user's explicitly-provided grammar specifications, making the standard library's definitions globally visible.

A partial grammar specification, or module, contains declarations and rules, just like a complete one: there is no visible difference. Of course, it can consist of only declarations, or only rules, if the user so chooses. (Don't forget the mandatory **%%** keyword that separates declarations and rules. It must be present, even if one of the two sections is empty.)

***Private and public nonterminal symbols*** It should be noted that joining is *not* a purely textual process. If two modules happen to define a nonterminal symbol by the same name, then it is considered, by default, that this is an accidental name clash. In that case, each of the two nonterminal symbols is silently renamed so as to avoid the clash. In other words, by default, a nonterminal symbol defined in module $A$ is considered *private*, and cannot be defined again, or referred to, in module $B$.

Naturally, it is sometimes desirable to define a nonterminal symbol $N$ in module $A$ and to refer to it in module $B$. This is permitted if $N$ is public, that is, if either its definition carries the keyword **%public** or $N$ is declared to be a start symbol. A public nonterminal symbol is never renamed, so it can be referred to by modules other than its defining module.

In fact, it is even permitted to split the definition of a public nonterminal symbol over multiple modules. That is, a public nonterminal symbol $N$ can have multiple definitions in distinct modules. When the modules are joined, the definitions are joined as well, using the choice (|) operator. This feature allows splitting a grammar specification in a manner that is independent of the grammar's structure. For instance, in the grammar of a programming language, the definition of the nonterminal symbol *expression* could be split into multiple modules, where one module groups the expression forms that have to do with arithmetic, one module groups those that concern function definitions and function calls, one module groups those that concern object definitions and method calls, and so on.

***Tokens aside*** Another use of modularity consists in placing all **%token** declarations in one module, and the actual grammar specification in another module. The module that contains the token definitions can then be shared, making it easier to define multiple parsers that accept the same type of tokens. (On this topic, see `demos/calc-two`.)

## 5.2 Parameterizing rules

A rule (that is, the definition of a nonterminal symbol) can be parameterized over an arbitrary number of symbols, which are referred to as formal parameters.

***Example*** For instance, here is the definition of the parameterized nonterminal symbol *option*, taken from the standard library (§5.4):

| | | |
|---|---|---|
| *actual*? | is syntactic sugar for | *option*(*actual*) |
| *actual*+ | is syntactic sugar for | *nonempty_list*(*actual*) |
| *actual** | is syntactic sugar for | *list*(*actual*) |

**Figure 2.** Syntactic sugar for simulating regular expressions

```
%public option(X):
    | { None }
    | x = X { Some x }
```

This definition states that *option*(*X*) expands to either the empty string, producing the semantic value *None*, or to the string *X*, producing the semantic value *Some x*, where *x* is the semantic value of *X*. In this definition, the symbol *X* is abstract: it stands for an arbitrary terminal or nonterminal symbol. The definition is made public, so *option* can be referred to within client modules.

A client that wishes to use *option* simply refers to it, together with an actual parameter – a symbol that is intended to replace *X*. For instance, here is how one might define a sequence of declarations, preceded with optional commas:

```
declarations:
    | { [] }
    | ds = declarations; option(COMMA); d = declaration { d :: ds }
```

This definition states that *declarations* expands either to the empty string or to *declarations* followed by an optional comma followed by *declaration*. (Here, *COMMA* is presumably a terminal symbol.) When this rule is encountered, the definition of *option* is instantiated: that is, a copy of the definition, where *COMMA* replaces *X*, is produced. Things behave exactly as if one had written:

```
optional_comma:
    | { None }
    | x = COMMA { Some x }
declarations:
    | { [] }
    | ds = declarations; optional_comma; d = declaration { d :: ds }
```

Note that, even though *COMMA* presumably has been declared as a token with no semantic value, writing *x* = *COMMA* is legal, and binds *x* to the unit value. This design choice ensures that the definition of *option* makes sense regardless of the nature of *X*: that is, *X* can be instantiated with a terminal symbol, with or without a semantic value, or with a nonterminal symbol.

***Parameterization in general*** In general, the definition of a nonterminal symbol $N$ can be parameterized with an arbitrary number of formal parameters. When $N$ is referred to within a production, it must be applied to the same number of actuals. In general, an actual is:

- either a single symbol, which can be a terminal symbol, a nonterminal symbol, or a formal parameter;
- or an application of such a symbol to a number of actuals.

For instance, here is a rule whose single production consists of a single producer, which contains several, nested actuals. (This example is discussed again in §5.4.)

```
plist(X):
    | xs = loption(delimited(LPAREN, separated_nonempty_list(COMMA, X), RPAREN)) { xs }
```

Applications of the parameterized nonterminal symbols *option*, *nonempty_list*, and *list*, which are defined in the standard library (§5.4), can be written using a familiar, regular-expression like syntax (Figure 2).

***Higher-order parameters*** A formal parameter can itself expect parameters. For instance, here is a rule that defines the syntax of procedures in an imaginary programming language:

```
procedure(list):
    | PROCEDURE ID list(formal) SEMICOLON block SEMICOLON { ... }
```

This rule states that the token *ID*, which represents the name of the procedure, should be followed with a list of formal parameters. (The definitions of the nonterminal symbols *formal* and *block* are not shown.) However, because *list* is a formal parameter, as opposed to a concrete nonterminal symbol defined elsewhere, this definition does not specify how the list is laid out: which token, if any, is used to separate, or terminate, list elements? is the list allowed to be empty? and so on. A more concrete notion of procedure is obtained by instantiating the formal parameter *list*: for instance, *procedure*(*plist*), where *plist* is the parameterized nonterminal symbol defined earlier, is a valid application.

***Consistency*** Definitions and uses of parameterized nonterminal symbols are checked for consistency before they are expanded away. In short, it is checked that, wherever a nonterminal symbol is used, it is supplied with actual arguments in appropriate number and of appropriate nature. This guarantees that expansion of parameterized definitions terminates and produces a well-formed grammar as its outcome.

### 5.3 Inlining

It is well-known that the following grammar of arithmetic expressions does not work as expected: that is, in spite of the priority declarations, it has shift/reduce conflicts.

```
%token < int > INT
%token PLUS TIMES
%left PLUS
%left TIMES

%%

expression:
    | i = INT { i }
    | e = expression; o = op; f = expression { o e f }
op:
    | PLUS { ( + ) }
    | TIMES { ( * ) }
```

The trouble is, the precedence level of the production *expression* → *expression op expression* is undefined, and there is no sensible way of defining it via a **%prec** declaration, since the desired level really depends upon the symbol that was recognized by *op*: was it *PLUS* or *TIMES*?

The standard workaround is to abandon the definition of *op* as a separate nonterminal symbol, and to inline its definition into the definition of *expression*, like this:

```
expression:
    | i = INT { i }
    | e = expression; PLUS; f = expression { e + f }
    | e = expression; TIMES; f = expression { e * f }
```

This avoids the shift/reduce conflict, but gives up some of the original specification's structure, which, in realistic situations, can be damageable. Fortunately, Menhir offers a way of avoiding the conflict without manually transforming the grammar, by declaring that the nonterminal symbol *op* should be inlined:

*expression*:
  | *i* = *INT* { *i* }
  | *e* = *expression*; *o* = *op*; *f* = *expression* { *o e f* }
**%inline** *op*:
  | *PLUS* { ( + ) }
  | *TIMES* { ( * ) }

The **%inline** keyword causes all references to *op* to be replaced with its definition. In this example, the definition of *op* involves two productions, one that develops to *PLUS* and one that expands to *TIMES*, so every production that refers to *op* is effectively turned into two productions, one that refers to *PLUS* and one that refers to *TIMES*. After inlining, *op* disappears and *expression* has three productions: that is, the result of inlining is exactly the manual workaround shown above.

In some situations, inlining can also help recover a slight efficiency margin. For instance, the definition:

**%inline** *plist*(*X*):
  | *xs* = *loption*(*delimited*(*LPAREN*, *separated_nonempty_list*(*COMMA*, *X*), *RPAREN*)) { *xs* }

effectively makes *plist*(*X*) an alias for the right-hand side *loption*(. . .). Without the **%inline** keyword, the language recognized by the grammar would be the same, but the LR automaton would probably have one more state and would perform one more reduction at run time.

The **%inline** keyword does not affect the computation of positions (§7). The same positions are computed, regardless of where **%inline** keywords are placed.

If the semantic actions have side effects, the **%inline** keyword *can* affect the order in which these side effects take place. In the example of *op* and *expression* above, if for some reason the semantic action associated with *op* has a side effect (such as updating a global variable, or printing a message), then, by inlining *op*, we delay this side effect, which takes place *after* the second operand has been recognized, whereas in the absence of inlining it takes place as soon as the operator has been recognized.

### 5.4 The standard library

Once equipped with a rudimentary module system (§5.1), parameterization (§5.2), and inlining (§5.3), it is straightforward to propose a collection of commonly used definitions, such as options, sequences, lists, and so on. This *standard library* is joined, by default, with every grammar specification. A summary of the nonterminal symbols offered by the standard library appears in Figure 3. See also the short-hands documented in Figure 2.

By relying on the standard library, a client module can concisely define more elaborate notions. For instance, the following rule:

**%inline** *plist*(*X*):
  | *xs* = *loption*(*delimited*(*LPAREN*, *separated_nonempty_list*(*COMMA*, *X*), *RPAREN*)) { *xs* }

causes *plist*(*X*) to recognize a list of *X*'s, where the empty list is represented by the empty string, and a non-empty list is delimited with parentheses and comma-separated.

## 6. Conflicts

When a shift/reduce or reduce/reduce conflict is detected, it is classified as either benign, if it can be resolved by consulting user-supplied precedence declarations, or severe, if it cannot. Benign conflicts are not reported. Severe conflicts are reported and, if the `--explain` switch is on, explained.

### 6.1 When is a conflict benign?

A shift/reduce conflict involves a single token (the one that one might wish to shift) and one or more productions (those that one might wish to reduce). When such a conflict is detected, the precedence level (§4.1.4, §4.2.1) of these entities are looked up and compared as follows:

| Name | Recognizes | Produces | Comment |
|---|---|---|---|
| *option*(*X*) | $\epsilon$ \| *X* | $\alpha$ *option*, if $X : \alpha$ | |
| *ioption*(*X*) | $\epsilon$ \| *X* | $\alpha$ *option*, if $X : \alpha$ | (inlined) |
| *boption*(*X*) | $\epsilon$ \| *X* | *bool* | |
| *loption*(*X*) | $\epsilon$ \| *X* | $\alpha$ *list*, if $X : \alpha$ *list* | |
| *pair*(*X*, *Y*) | *X Y* | $\alpha \times \beta$, if $X : \alpha$ and $Y : \beta$ | |
| *separated_pair*(*X*, *sep*, *Y*) | *X sep Y* | $\alpha \times \beta$, if $X : \alpha$ and $Y : \beta$ | |
| *preceded*(*opening*, *X*) | *opening X* | $\alpha$, if $X : \alpha$ | |
| *terminated*(*X*, *closing*) | *X closing* | $\alpha$, if $X : \alpha$ | |
| *delimited*(*opening*, *X*, *closing*) | *opening X closing* | $\alpha$, if $X : \alpha$ | |
| *list*(*X*) | a possibly empty sequence of *X*'s | $\alpha$ *list*, if $X : \alpha$ | |
| *nonempty_list*(*X*) | a nonempty sequence of *X*'s | $\alpha$ *list*, if $X : \alpha$ | |
| *separated_list*(*sep*, *X*) | a possibly empty sequence of *X*'s separated with *sep*'s | $\alpha$ *list*, if $X : \alpha$ | |
| *separated_nonempty_list*(*sep*, *X*) | a nonempty sequence of *X*'s separated with *sep*'s | $\alpha$ *list*, if $X : \alpha$ | |

**Figure 3.** Summary of the standard library

1. if only one production is involved, and if it has higher priority than the token, then the conflict is resolved in favor of reduction.
2. if only one production is involved, and if it has the same priority as the token, then the associativity status of the token is looked up:
   (a) if the token was declared nonassociative, then the conflict is resolved in favor of neither action, that is, a syntax error will be signaled if this token shows up when this production is about to be reduced;
   (b) if the token was declared left-associative, then the conflict is resolved in favor of reduction;
   (c) if the token was declared right-associative, then the conflict is resolved in favor of shifting.
3. if multiple productions are involved, and if, considered one by one, they all cause the conflict to be resolved in the same way (that is, either in favor in shifting, or in favor of neither), then the conflict is resolved in that way.

In either of these cases, the conflict is considered benign. Otherwise, it is considered severe. Note that a reduce/reduce conflict is always considered severe, unless it happens to be subsumed by a benign multi-way shift/reduce conflict (item 3 above).

### 6.2 How are severe conflicts explained?

When the `--dump` switch is on, a description of the automaton is written to the `.automaton` file. Severe conflicts are shown as part of this description. Fortunately, there is also a way of understanding conflicts in terms of the grammar, rather than in terms of the automaton. When the `--explain` switch is on, a textual explanation is written to the `.conflicts` file.

*Not all conflicts are explained* in this file: instead, *only one conflict per automaton state is explained*. This is done partly in the interest of brevity, but also because Pager's algorithm can create artificial conflicts in a state that already contains a true LR(1) conflict; thus, one cannot hope in general to explain all of the conflicts that appear in the automaton. As a result of this policy, once all conflicts explained in the `.conflicts` file have been fixed, one might need to run Menhir again to produce yet more conflict explanations.

```
%token IF THEN ELSE
%start < expression > expression

%%

expression:
      | ...
      | IF b = expression THEN e = expression { ... }
      | IF b = expression THEN e = expression ELSE f = expression { ... }
      | ...
```

**Figure 4.** Basic example of a shift/reduce conflict

***How the conflict state is reached*** Figure 4 shows a grammar specification with a typical shift/reduce conflict. When this specification is analyzed, the conflict is detected, and an explanation is written to the `.conflicts` file. The explanation first indicates in which state the conflict lies by showing how that state is reached. Here, it is reached after recognizing the following string of terminal and nonterminal symbols—the *conflict string*:

*IF expression THEN IF expression THEN expression*

Allowing the conflict string to contain both nonterminal and terminal symbols usually makes it shorter and more readable. If desired, a conflict string composed purely of terminal symbols could be obtained by replacing each occurrence of a nonterminal symbol $N$ with an arbitrary $N$-sentence.

The conflict string can be thought of as a path that leads from one of the automaton's start states to the conflict state. When multiple such paths exist, the one that is displayed is chosen shortest. Nevertheless, it may sometimes be quite long. In that case, artificially (and temporarily) declaring some existing nonterminal symbols to be start symbols has the effect of adding new start states to the automaton and can help produce shorter conflict strings. Here, *expression* was declared to be a start symbol, which is why the conflict string is quite short.

In addition to the conflict string, the `.conflicts` file also states that the *conflict token* is *ELSE*. That is, when the automaton has recognized the conflict string and when the lookahead token (the next token on the input stream) is *ELSE*, a conflict arises. A conflict corresponds to a choice: the automaton is faced with several possible actions, and does not know which one should be taken. This indicates that the grammar is not LR(1). The grammar may or may not be inherently ambiguous.

In our example, the conflict string and the conflict token are enough to understand why there is a conflict: when two *IF* constructs are nested, it is ambiguous which of the two constructs the *ELSE* branch should be associated with. Nevertheless, the `.conflicts` file provides further information: it explicitly shows that there exists a conflict, by proving that two distinct actions are possible. Here, one of these actions consists in *shifting*, while the other consists in *reducing*: this is a *shift/reduce* conflict.

A *proof* takes the form of a *partial derivation tree* whose *fringe* begins with the conflict string, followed by the conflict token. A derivation tree is a tree whose nodes are labeled with symbols. The root node carries a start symbol. A node that carries a terminal symbol is considered a leaf, and has no children. A node that carries a nonterminal symbol $N$ either is considered a leaf, and has no children; or is not considered a leaf, and has $n$ children, where $n \geq 0$, labeled $x_1, \ldots, x_n$, where $N \rightarrow x_1, \ldots, x_n$ is a production. The fringe of a partial derivation tree is the string of terminal and nonterminal symbols carried by the tree's leaves. A string of terminal and nonterminal symbols that is the fringe of some partial derivation tree is a *sentential form*.

***Why shifting is legal*** In our example, the proof that shifting is possible is the derivation tree shown in Figures 5 and 6. At the root of the tree is the grammar's start symbol, *expression*. This symbol develops into the string *IF expression THEN expression*, which forms the tree's second level. The second occurrence of *expression* in

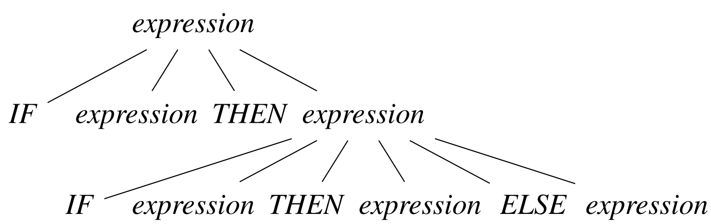


**Figure 5.** A partial derivation tree that justifies shifting

*expression*
*IF expression THEN expression*
*IF expression THEN expression . ELSE expression*

**Figure 6.** A textual version of the tree in Figure 5

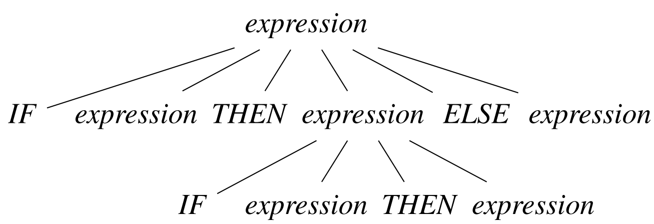


**Figure 7.** A partial derivation tree that justifies reducing

*expression*
*IF expression THEN expression ELSE expression* ***// lookahead token appears***
*IF expression THEN expression .*

**Figure 8.** A textual version of the tree in Figure 7

that string develops into *IF expression THEN expression ELSE expression*, which forms the tree's last level. The tree's fringe, a sentential form, is the string *IF expression THEN IF expression THEN expression ELSE expression*. As announced earlier, it begins with the conflict string *IF expression THEN IF expression THEN expression*, followed with the conflict token *ELSE*.

In Figure 6, the end of the conflict string is materialized with a dot. Note that this dot does not occupy the rightmost position in the tree's last level. In other words, the conflict token (*ELSE*) itself occurs on the tree's last level. In practical terms, this means that, after the automaton has recognized the conflict string and peeked at the conflict token, it makes sense for it to *shift* that token.

***Why reducing is legal*** In our example, the proof that shifting is possible is the derivation tree shown in Figures 7 and 8. Again, the sentential form found at the fringe of the tree begins with the conflict string, followed with the conflict token.

Again, in Figure 8, the end of the conflict string is materialized with a dot. Note that, this time, the dot occupies the rightmost position in the tree's last level. In other words, the conflict token (*ELSE*) appeared on an earlier level (here, on the second level). This fact is emphasized by the comment ***// lookahead token appears*** found at the second level. In practical terms, this means that, after the automaton has recognized the conflict string

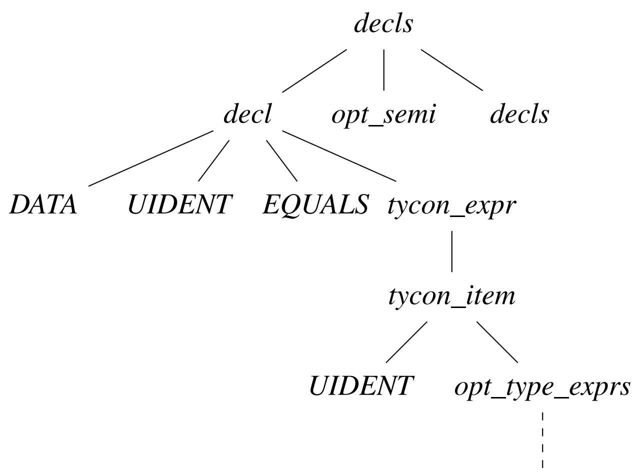


**Figure 9.** A partial derivation tree that justifies reducing

*decls*
*decl opt_semi decls* ***// lookahead token appears because opt_semi can vanish and decls can begin with LIDENT***
*DATA UIDENT EQUALS tycon_expr* ***// lookahead token is inherited***
*tycon_item* ***// lookahead token is inherited***
*UIDENT opt_type_exprs* ***// lookahead token is inherited***
.

**Figure 10.** A textual version of the tree in Figure 9

and peeked at the conflict token, it makes sense for it to *reduce* the production that corresponds to the tree's last level—here, the production is *expression* $\rightarrow$ *IF expression THEN expression*.

***An example of a more complex derivation tree*** Figures 9 and 10 show a partial derivation tree that justifies reduction in a more complex situation. (This derivation tree is relative to a grammar that is not shown.) Here, the conflict string is *DATA UIDENT EQUALS UIDENT*; the conflict token is *LIDENT*. It is quite clear that the fringe of the tree begins with the conflict string. However, in this case, the fringe does not explicitly exhibit the conflict token. Let us examine the tree more closely and answer the question: following *UIDENT*, what's the next terminal symbol on the fringe?

First, note that *opt_type_exprs* is *not* a leaf node, even though it has no children. The grammar contains the production *opt_type_exprs* $\rightarrow \epsilon$: the nonterminal symbol *opt_type_exprs* develops to the empty string. (This is made clear in Figure 10, where a single dot appears immediately below *opt_type_exprs*.) Thus, *opt_type_exprs* is not part of the fringe.

Next, note that *opt_type_exprs* is the rightmost symbol within its level. Thus, in order to find the next symbol on the fringe, we have to look up one level. This is the meaning of the comment ***// lookahead token is inherited***. Similarly, *tycon_item* and *tycon_expr* appear rightmost within their level, so we again have to look further up.

This brings us back to the tree's second level. There, *decl* is *not* the rightmost symbol: next to it, we find *opt_semi* and *decls*. Does this mean that *opt_semi* is the next symbol on the fringe? Yes and no. *opt_semi* is a *nonterminal* symbol, but we are really interested in finding out what the next *terminal* symbol on the fringe could be. The partial derivation tree shown in Figures 9 and 10 does not explicitly answer this question. In order to answer it, we need to know more about *opt_semi* and *decls*.

Here, *opt_semi* stands (as one might have guessed) for an optional semicolon, so the grammar contains a production *opt_semi* $\rightarrow \epsilon$. This is indicated by the comment ***// opt_semi can vanish***. (Nonterminal symbols that generate $\epsilon$ are also said to be *nullable*.) Thus, one could choose to turn this partial derivation tree into a larger one by developing *opt_semi* into $\epsilon$, making it a non-leaf node. That would yield a new partial derivation tree where the next symbol on the fringe, following *UIDENT*, is *decls*.

Now, what about *decls*? Again, it is a *nonterminal* symbol, and we are really interested in finding out what the next *terminal* symbol on the fringe could be. Again, we need to imagine how this partial derivation tree could be turned into a larger one by developing *decls*. Here, the grammar happens to contain a production of the form *decls* $\rightarrow$ *LIDENT* ... This is indicated by the comment ***// decls can begin with LIDENT***. Thus, by developing *decls*, it is possible to construct a partial derivation tree where the next symbol on the fringe, following *UIDENT*, is *LIDENT*. This is precisely the conflict token.

To sum up, there exists a partial derivation tree whose fringe begins the conflict string, followed with the conflict token. Furthermore, in that derivation tree, the dot occupies the rightmost position in the last level. As in our previous example, this means that, after the automaton has recognized the conflict string and peeked at the conflict token, it makes sense for it to *reduce* the production that corresponds to the tree's last level—here, the production is *opt_type_exprs* $\rightarrow \epsilon$.

***Greatest common factor among derivation trees*** Understanding conflicts requires comparing two (or more) derivation trees. It is frequent for these trees to exhibit a common factor, that is, to exhibit identical structure near the top of the tree, and to differ only below a specific node. Manual identification of that node can be tedious, so Menhir performs this work automatically. When explaining a $n$-way conflict, it first displays the greatest common factor of the $n$ derivation trees. A question mark symbol *(?)* is used to identify the node where the trees begin to differ. Then, Menhir displays each of the $n$ derivation trees, *without their common factor* – that is, it displays $n$ sub-trees that actually begin to differ at the root. This should make visual comparisons significantly easier.

### 6.3 How are severe conflicts resolved in the end?

It is unspecified how severe conflicts are resolved. Menhir attempts to mimic `ocamlyacc`'s specification, that is, to resolve shift/reduce conflicts in favor of shifting, and to resolve reduce/reduce conflicts in favor of the production that textually appears earliest in the grammar specification. However, this specification is inconsistent in case of three-way conflicts, that is, conflicts that simultaneously involve a shift action and several reduction actions. Furthermore, textual precedence can be undefined when the grammar specification is split over multiple modules. In short, Menhir's philosophy is that

severe conflicts should not be tolerated,

so you should not care how they are resolved.

### 6.4 End-of-stream conflicts

Menhir's treatment of the end of the token stream is (believed to be) fully compatible with `ocamlyacc`'s. Yet, Menhir attempts to be more user-friendly by warning about a class of so-called "end-of-stream conflicts".

***How the end of stream is handled*** In many textbooks on parsing, it is assumed that the lexical analyzer, which produces the token stream, produces a special token, written $\#$, to signal that the end of the token stream has been reached. A parser generator can take advantage of this by transforming the grammar: for each start symbol *S* in the original grammar, a new start symbol *S'* is defined, together with the production $S' \rightarrow S\#$. The symbol $S$ is no longer a start symbol in the new grammar. This means that the parser will accept a sentence derived from $S$ only if it is immediately followed by the end of the token stream.

This approach has the advantage of simplicity. However, `ocamlyacc` and Menhir do not follow it, for several reasons. Perhaps the most convincing one is that it is not flexible enough: sometimes, it is desirable to recognize a sentence derived from $S$, *without* requiring that it be followed by the end of the token stream: this is the case, for instance, when reading commands, one by one, on the standard input channel. In that case, there is no end of stream: the token stream is conceptually infinite. Furthermore, after a command has been recognized, we do *not* wish to examine the next token, because doing so might cause the program to block, waiting for more input.

In short, `ocamlyacc` and Menhir's approach is to recognize a sentence derived from $S$ and to *not look*, if possible, at what follows. However, this is possible only if the definition of $S$ is such that the end of an $S$-sentence is identifiable without knowledge of the lookahead token. When the definition of $S$ does not satisfy this criterion, and *end-of-stream conflict* arises: after a potential $S$-sentence has been read, there can be a tension between consulting the next token, in order to determine whether the sentence is continued, and *not* consulting the next token, because the sentence might be over and whatever follows should not be read. Menhir warns about end-of-stream conflicts, whereas `ocamlyacc` does not.

***A definition of end-of-stream conflicts*** Technically, Menhir proceeds as follows. A $\#$ symbol is introduced. It is, however, only a *pseudo*-token: it is never produced by the lexical analyzer. For each start symbol *S* in the original grammar, a new start symbol *S'* is defined, together with the production $S' \to S$. The corresponding start state of the LR(1) automaton is composed of the LR(1) item $S' \to\ .\ S\ [\#]$. That is, the pseudo-token $\#$ initially appears in the lookahead set, indicating that we expect to be done after recognizing an $S$-sentence. During the construction of the LR(1) automaton, this lookahead set is inherited by other items, with the effect that, in the end, the automaton has:

- *shift* actions only on physical tokens; and
- *reduce* actions either on physical tokens or on the pseudo-token $\#$.

A state of the automaton has a reduce action on $\#$ if, in that state, an $S$-sentence has been read, so that the job is potentially finished. A state has a shift or reduce action on a physical token if, in that state, more tokens potentially need to be read before an $S$-sentence is recognized. If a state has a reduce action on $\#$, then that action should be taken *without* requesting the next token from the lexical analyzer. On the other hand, if a state has a shift or reduce action on a physical token, then the lookahead token *must* be consulted in order to determine if that action should be taken.

An end-of-stream conflict arises when a state has distinct actions on $\#$ and on at least one physical token. In short, this means that the end of an $S$-sentence cannot be unambiguously identified without examining one extra token. Menhir's default behavior, in that case, is to suppress the action on $\#$, so that more input is *always* requested.

***Example*** Figure 11 shows a grammar that has end-of-stream conflicts. When this grammar is processed, Menhir warns about these conflicts, and further warns that *expr* is never accepted. Let us explain.

Part of the corresponding automaton, as described in the `.automaton` file, is shown in Figure 12. Explanations at the end of the `.automaton` file (not shown) point out that states 6 and 2 have an end-of-stream conflict. Indeed, both states have distinct actions on $\#$ and on the physical token *TIMES*. It is interesting to note that, even though state 4 has actions on $\#$ and on physical tokens, it does not have an end-of-stream conflict. This is because the action taken in state 4 is always to reduce the production *expr* $\to$ *expr TIMES expr*, regardless of the lookahead token.

By default, Menhir produces a parser where end-of-stream conflicts are resolved in favor of looking ahead: that is, the problematic reduce actions on $\#$ are suppressed. This means, in particular, that the *accept* action in state 2, which corresponds to reducing the production *expr* $\to$ *expr'*, is suppressed. This explains why the symbol *expr* is never accepted: because expressions do not have an unambiguous end marker, the parser will always request one more token and will never stop.

In order to avoid this end-of-stream conflict, the standard solution is to introduce a new token, say *END*, and to use it as an end marker for expressions. The *END* token could be generated by the lexical analyzer when it encounters the actual end of stream, or it could correspond to a piece of concrete syntax, say, a line feed character, a semicolon, or an `end` keyword. The solution is shown in Figure 13.

```
%token < int > INT
%token PLUS TIMES
%left PLUS
%left TIMES
%start < int > expr
%%
expr:
    | i = INT { i }
    | e1 = expr PLUS e2 = expr { e1 + e2 }
    | e1 = expr TIMES e2 = expr { e1 * e2 }
```

**Figure 11.** Basic example of an end-of-stream conflict

```
State 6:
expr -> expr . PLUS expr [ # TIMES PLUS ]
expr -> expr PLUS expr . [ # TIMES PLUS ]
expr -> expr . TIMES expr [ # TIMES PLUS ]
-- On TIMES shift to state 3
-- On # PLUS reduce production expr -> expr PLUS expr

State 4:
expr -> expr . PLUS expr [ # TIMES PLUS ]
expr -> expr . TIMES expr [ # TIMES PLUS ]
expr -> expr TIMES expr . [ # TIMES PLUS ]
-- On # TIMES PLUS reduce production expr -> expr TIMES expr

State 2:
expr' -> expr . [ # ]
expr -> expr . PLUS expr [ # TIMES PLUS ]
expr -> expr . TIMES expr [ # TIMES PLUS ]
-- On TIMES shift to state 3
-- On PLUS shift to state 5
-- On # accept expr
```

**Figure 12.** Part of an LR automaton for the grammar in Figure 11

```
...
%token END
%start < int > main        // instead of expr
%%
main:
    | e = expr END { e }
expr:
    | ...
```

**Figure 13.** Fixing the grammar specification in Figure 11

| | |
|---|---|
| `$startpos` | start position of the first symbol in the production's right-hand side, if there is one; end position of the most recently parsed symbol, otherwise |
| `$endpos` | end position of the first symbol in the production's right-hand side, if there is one; end position of the most recently parsed symbol, otherwise |
| `$startpos(` `$`*i* \| *id* `)` | start position of the symbol named `$`*i* or *id* |
| `$endpos(` `$`*i* \| *id* `)` | end position of the symbol named `$`*i* or *id* |
| `$symbolstartpos` | start position of the leftmost symbol *id* such that `$startpos(`*id*`) != $endpos(`*id*`)`; if there is no such symbol, `$endpos` |
| `$startofs` | |
| `$endofs` | |
| `$startofs(` `$`*i* \| *id* `)` | same as above, but produce an integer offset instead of a position |
| `$endofs(` `$`*i* \| *id* `)` | |
| `$symbolstartofs` | |

**Figure 14.** Position-related keywords

| | | |
|---|---|---|
| `symbol_start_pos()` | `$symbolstartpos` | |
| `symbol_end_pos()` | `$endpos` | |
| `rhs_start_pos i` | `$startpos($i)` | $(1 \leq i \leq n)$ |
| `rhs_end_pos i` | `$endpos($i)` | $(1 \leq i \leq n)$ |
| `symbol_start()` | `$symbolstartofs` | |
| `symbol_end()` | `$endofs` | |
| `rhs_start i` | `$startofs($i)` | $(1 \leq i \leq n)$ |
| `rhs_end i` | `$endofs($i)` | $(1 \leq i \leq n)$ |

**Figure 15.** Translating position-related incantations from `ocamlyacc` to Menhir

## 7. Positions

When an `ocamllex`-generated lexical analyzer produces a token, it updates two fields, named `lex_start_p` and `lex_curr_p`, in its environment record, whose type is `Lexing.lexbuf`. Each of these fields holds a value of type `Lexing.position`. Together, they represent the token's start and end positions within the text that is being scanned. A position consists mainly of an offset (the position's `pos_cnum` field), but also holds information about the current file name, the current line number, and the current offset within the current line. (Not all `ocamllex`-generated analyzers keep this extra information up to date. This must be explicitly programmed by the author of the lexical analyzer.)

This mechanism allows associating pairs of positions with terminal symbols. If desired, Menhir automatically extends it to nonterminal symbols as well. That is, it offers a mechanism for associating pairs of positions with terminal or nonterminal symbols. This is done by making a set of keywords available to semantic actions (Figure 14). Note that these keywords are *not* available outside of a semantic action: in particular, they cannot be used within an OCaml header. Note also that OCaml's standard library module `Parsing` is deprecated. The functions that it offers *can* be called, but will return dummy positions.

We remark that, if the current production has an empty right-hand side, then `$startpos` and `$endpos` are equal, and (by convention) are the end position of the most recently parsed symbol (that is, the symbol that happens to be on top of the automaton's stack when this production is reduced). If the current production has a nonempty right-hand side, then `$startpos` is the same as `$startpos($1)` and `$endpos` is the same as `$endpos($`*n*`)`, where *n* is the length of the right-hand side.

More generally, if the current production has matched a sentence of length zero, then `$startpos` and `$endpos` will be equal, and conversely.

The position `$startpos` is sometimes "further towards the left" than one would like. For example, in the following production:

```
declaration: modifier? variable { $startpos }
```

the keyword `$startpos` represents the start position of the optional modifier `modifier?`. If this modifier turns out to be absent, then its start position is (by definition) the end position of the most recently parsed symbol. This may not be what is desired: perhaps the user would prefer in this case to use the start position of the symbol `variable`. This is achieved by using `$symbolstartpos` instead of `$startpos`. By definition, `$symbolstartpos` is the start position of the leftmost symbol whose start and end positions differ. In this example, the computation of `$symbolstartpos` skips the absent `modifier`, whose start and end positions coincide, and returns the start position of the symbol `variable` (assuming this symbol has distinct start and end positions).

There is no keyword `$symbolendpos`. Indeed, the problem with `$startpos` is due to the asymmetry in the definition of `$startpos` and `$endpos` in the case of an empty right-hand side, and does not affect `$endpos`.

The positions computed by Menhir are exactly the same as those computed by `ocamlyacc`[1]. More precisely, Figure 15 sums up how to translate a call to the `Parsing` module, as used in an `ocamlyacc` grammar, to a Menhir keyword.

We note that Menhir's `$startpos` does not appear in the right-hand column in Figure 15. In other words, Menhir's `$startpos` does not correspond exactly to any of the `ocamlyacc` function calls. An exact `ocamlyacc` equivalent of `$startpos` is `rhs_start_pos 1` if the current production has a nonempty right-hand side and `symbol_start_pos()` if it has an empty right-hand side.

Finally, we remark that Menhir's **%inline** keyword (§5.3) does not affect the computation of positions. The same positions are computed, regardless of where **%inline** keywords are placed.

## 8. Using Menhir as an interpreter

When `--interpret` is set, Menhir no longer behaves as a compiler. Instead, it acts as an interpreter. That is, it repeatedly:

- reads a sentence off the standard input channel;
- parses this sentence, according to the grammar;
- displays an outcome.

This process stops when the end of the input channel is reached.

### 8.1 Sentences

The syntax of sentences is as follows:

$$sentence ::= [\, lid \, \texttt{:} \,]\ uid\ \ldots\ uid\ \backslash\mathbf{n}$$

Less formally, a sentence is a sequence of zero or more terminal symbols (*uid*'s), separated with whitespace, terminated with a newline character, and optionally preceded with a non-terminal start symbol (*lid*). This non-terminal symbol can be omitted if, and only if, the grammar only has one start symbol.

For instance, here are four valid sentences for the grammar of arithmetic expressions found in the directory `demos/calc`:

```
main: INT PLUS INT EOL
```

[1] The computation of `$symbolstartpos` is optimized by Menhir under two assumptions about the lexer. First, Menhir assumes that the lexer never produces a token whose start and end positions are equal. Second, Menhir assumes that two positions produced by the lexer are equal if and only if they are physically equal. If the lexer violates either of these assumptions, the computation of `$symbolstartpos` could produce a result that differs from `Parsing.symbol_start_pos()`.

```
INT PLUS INT
INT PLUS PLUS INT EOL
INT PLUS PLUS
```

In the first sentence, the start symbol `main` was explicitly specified. In the other sentences, it was omitted, which is permitted, because this grammar has no start symbol other than `main`. The first sentence is a stream of four terminal symbols, namely `INT`, `PLUS`, `INT`, and `EOL`. These terminal symbols must be provided under their symbolic names. Writing, say, "`12+32\n`" instead of `INT PLUS INT EOL` is not permitted. Menhir would not be able to make sense of such a concrete notation, since it does not have a lexer for it.

### 8.2 Outcomes

As soon as Menhir is able to read a complete sentence off the standard input channel (that is, as soon as it finds the newline character that ends the sentence), it parses the sentence according to whichever grammar was specified on the command line, and displays an outcome.

An outcome is one of the following:

- `ACCEPT`: a prefix of the sentence was successfully parsed; a parser generated by Menhir would successfully stop and produce a semantic value;
- `OVERSHOOT`: the end of the sentence was reached before it could be accepted; a parser generated by Menhir would request a non-existent "next token" from the lexer, causing it to fail or block;
- `REJECT`: the sentence was not accepted; a parser generated by Menhir would raise the exception `Error`.

When `--interpret-show-cst` is set, each `ACCEPT` outcome is followed with a concrete syntax tree. A concrete syntax tree is either a leaf or a node. A leaf is either a terminal symbol or **error**. A node is annotated with a non-terminal symbol, and carries a sequence of immediate descendants that correspond to a valid expansion of this non-terminal symbol. Menhir's notation for concrete syntax trees is as follows:

$$\begin{array}{rl} cst ::= & uid \\ & \textbf{error} \\ & [\ lid : cst\ \ldots\ cst\ ] \end{array}$$

For instance, if one wished to parse the example sentences of §8.1 using the grammar of arithmetic expressions in `demos/calc`, one could invoke Menhir as follows:

```
$ menhir --interpret --interpret-show-cst demos/calc/parser.mly
main: INT PLUS INT EOL
ACCEPT
[main: [expr: [expr: INT] PLUS [expr: INT]] EOL]
INT PLUS INT
OVERSHOOT
INT PLUS PLUS INT EOL
REJECT
INT PLUS PLUS
REJECT
```

(Here, Menhir's input—the sentences provided by the user on the standard input channel— is shown intermixed with Menhir's output—the outcomes printed by Menhir on the standard output channel.) The first sentence is valid, and accepted; a concrete syntax tree is displayed. The second sentence is incomplete, because the grammar specifies that a valid expansion of `main` ends with the terminal symbol `EOL`; hence, the outcome is `OVERSHOOT`. The third sentence is invalid, because of the repeated occurrence of the terminal symbol `PLUS`; the outcome is `REJECT`. The fourth sentence, a prefix of the third one, is rejected for the same reason.

## 8.3 Remarks

Using Menhir as an interpreter offers an easy way of debugging your grammar. For instance, if one wished to check that addition is considered left-associative, as requested by the %left directive found in the file demos/calc/parser.mly, one could submit the following sentence:

```
$ ./menhir --interpret --interpret-show-cst ../demos/calc/parser.mly
INT PLUS INT PLUS INT EOL
ACCEPT
[main:
  [expr: [expr: [expr: INT] PLUS [expr: INT]] PLUS [expr: INT]]
  EOL
]
```

The concrete syntax tree displayed by Menhir is skewed towards the left, as desired.

The switches --interpret and --trace can be used in conjunction. When --trace is set, the interpreter logs its actions to the standard error channel.

# 9. Generated API

When Menhir processes a grammar specification, say parser.mly, it produces one OCaml module, Parser, whose code resides in the file parser.ml and whose signature resides in the file parser.mli. We now review this signature. For simplicity, we assume that the grammar specification has just one start symbol main, whose OCaml type is thing.

## 9.1 Monolithic API

The monolithic API defines the type token, the exception Error, and the parsing function main, named after the start symbol of the grammar.

The type token is an algebraic data type. A value of type token represents a terminal symbol and its semantic value. For instance, if the grammar contains the declarations %token A and %token<int> B, then the generated file parser.mli contains the following definition:

```
type token =
| A
| B of int
```

If --only-tokens is specified on the command line, the type token is generated, and the rest is omitted. On the contrary, if --external-tokens is used, the type token is omitted, but the rest (described below) is generated.

The exception Error carries no argument. It is raised by the parsing function main (described below) when a syntax error is detected.

```
exception Error
```

Next comes one parsing function for each start symbol of the grammar. Here, we have assumed that there is one start symbol, named main, so the generated file parser.mli contains the following declaration:

```
val main: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> thing
```

This function expects two arguments, namely: a lexer, which typically is produced by ocamllex and has type Lexing.lexbuf -> token; and a lexing buffer, which has type Lexing.lexbuf. This API is compatible with ocamlyacc. (For information on using Menhir without ocamllex, please consult §14.) This API is “monolithic” in the sense that there is just one function, which does everything: it pulls tokens from the lexer, parses, and eventually returns a semantic value (or fails by throwing the exception Error).

## 9.2 Incremental API

If `--table` is set, Menhir offers an incremental API in addition to the monolithic API. In this API, control is inverted. The parser does not have access to the lexer. Instead, when the parser needs the next token, it stops and returns its current state to the user. The user is then responsible for obtaining this token (typically by invoking the lexer) and resuming the parser from that state. The directory `demos/calc-incremental` contains a demo that illustrates the use of the incremental API.

This API is "incremental" in the sense that the user has access to a sequence of the intermediate states of the parser. Assuming that semantic values are immutable, a parser state is a persistent data structure: it can be stored and used multiple times, if desired. This enables applications such as "live parsing", where a buffer is continuously parsed while it is being edited. The parser can be re-started in the middle of the buffer whenever the user edits a character. Because two successive parser states share most of their data in memory, a list of $n$ successive parser states occupies only $O(n)$ space in memory.

In this API, the parser is started by invoking `Incremental.main`. (Recall that we assume `main` is the name of the start symbol.) The generated file `parser.mli` contains the following declaration:

```
module Incremental : sig
  val main: Lexing.position -> thing MenhirInterpreter.checkpoint
end
```

The argument is the initial position. If the lexer is based on an OCaml lexing buffer, this argument should be `lexbuf.lex_curr_p`. We emphasize that the function `Incremental.main` does not parse anything. It constructs a checkpoint which serves as a *starting* point. The functions `offer` and `resume`, described below, are used to drive the parser.

The sub-module `MenhirInterpreter` is also part of the incremental API. Its declaration, which appears in the generated file `parser.mli`, is as follows:

```
module MenhirInterpreter : MenhirLib.IncrementalEngine.INCREMENTAL_ENGINE
  with type token = token
```

The signature `INCREMENTAL_ENGINE`, defined in the module `MenhirLib.IncrementalEngine`, contains the following elements. Please keep in mind that, from the outside, these elements should be referred to with an appropriate prefix: e.g., the type `checkpoint` should be referred to as `MenhirInterpreter.checkpoint`, or `Parser.MenhirInterpreter.checkpoint`, depending on which modules the user chooses to open.

```
type env
```

The abstract type `env` represents the current state of the parser. (That is, it contains the current state and stack of the LR automaton.) Assuming that semantic values are immutable, it is a persistent data structure: it can be stored and used multiple times, if desired.

```
type production
```

The abstract type `production` represents a production of the grammar.

```
type 'a checkpoint = private
  | InputNeeded of env
  | Shifting of env * env * bool
  | AboutToReduce of env * production
  | HandlingError of env
  | Accepted of 'a
  | Rejected
```

The type `'a checkpoint` represents an intermediate or final state of the parser. An intermediate checkpoint is a suspension: it records the parser's current state, and allows parsing to be resumed. The parameter `'a` is the type of the semantic value that will eventually be produced if the parser succeeds.

`Accepted` and `Rejected` are final checkpoints. `Accepted` carries a semantic value.

`InputNeeded` is an intermediate checkpoint. It means that the parser wishes to read one token before continuing.

`Shifting` is an intermediate checkpoint. It means that the parser is taking a shift transition. It exposes the state of the parser before and after the transition. The Boolean parameter tells whether the parser intends to request a new token after this transition. (It always does, except when it is about to accept.)

`AboutToReduce` is an intermediate checkpoint: it means that the parser is about to perform a reduction step. `HandlingError` is also an intermediate checkpoint: it means that the parser has detected an error and is about to handle it. (Error handling is typically performed in several steps, so the next checkpoint is likely to be `HandlingError` again.) In these two cases, the parser does not need more input. The parser suspends itself at this point only in order to give the user an opportunity to observe the parser's transitions and possibly handle errors in a different manner, if desired.

```
val offer:
  'a checkpoint ->
  token * Lexing.position * Lexing.position ->
  'a checkpoint
```

The function `offer` allows the user to resume the parser after the parser has suspended itself with a checkpoint of the form `InputNeeded env`. This function expects the previous checkpoint `checkpoint` as well as a new token (together with the start and end positions of this token). It produces a new checkpoint, which again can be an intermediate checkpoint or a final checkpoint. It does not raise any exception. (The exception `Error` is used only in the monolithic API.)

```
val resume:
  'a checkpoint ->
  'a checkpoint
```

The function `resume` allows the user to resume the parser after the parser has suspended itself with a checkpoint of the form `AboutToReduce (env, prod)` or `HandlingError env`. This function expects just the previous checkpoint `checkpoint`. It produces a new checkpoint. It does not raise any exception.

The incremental API subsumes the monolithic API. Indeed, `main` can be (and is in fact) implemented by first using `Incremental.main`, then calling `offer` and `resume` in a loop, until a final checkpoint is obtained.

Although the type `env` is opaque, a parser state can be inspected via a few accessor functions, which we are about to describe. Before we do so, we give a few more type definitions.

```
type supplier =
  unit -> token * Lexing.position * Lexing.position
```

A token supplier is a function of no arguments which delivers a new token (together with its start and end positions) every time it is called. The function `loop` and its variants, described below, expect a supplier as an argument.

```
val lexer_lexbuf_to_supplier:
  (Lexing.lexbuf -> token) ->
  Lexing.lexbuf ->
  supplier
```

The function `lexer_lexbuf_to_supplier`, applied to a lexer and to a lexing buffer, produces a fresh supplier.

The functions `offer` and `resume`, documented above, are sufficient to write a parser loop. One can imagine many variations of such a loop, which is why we expose `offer` and `resume` in the first place!. Nevertheless, some variations are so common that it is worth providing them, ready for use.

```
val loop: supplier -> 'a checkpoint -> 'a
```

`loop supplier checkpoint` begins parsing from `checkpoint`, reading tokens from `supplier`. It continues parsing until it reaches a checkpoint of the form `Accepted v` or `Rejected`. In the former case, it returns `v`. In the latter case, it raises the exception `Error`. (By the way, this is how we implement the monolithic API on top of the incremental API.)

```
val loop_handle:
  ('a -> 'answer) ->
  ('a checkpoint -> 'answer) ->
  supplier -> 'a checkpoint -> 'answer
```

`loop_handle succeed fail supplier checkpoint` begins parsing from `checkpoint`, reading tokens from `supplier`. It continues parsing until it reaches a checkpoint of the form `Accepted v` or `HandlingError env` (or `Rejected`, but that should not happen, as `HandlingError _` will be observed first). In the former case, it calls `succeed v`. In the latter case, it calls `fail` with this checkpoint. It cannot raise `Error`.

This means that Menhir's traditional error-handling procedure (which pops the stack until a state that can act on the **error** token is found) does not get a chance to run. Instead, the user can implement her own error handling code, in the `fail` continuation.

```
type 'a lr1state
```

The abstract type `'a lr1state` describes a (non-initial) state of the LR(1) automaton. If `s` is such a state, then `s` should have at least one incoming transition, and all of its incoming transitions carry the same (terminal or non-terminal) symbol, say $A$. We say that $A$ is the *incoming symbol* of the state `s`. The index `'a` is the type of the semantic values associated with $A$. The role played by `'a` is clarified in the definition of the type `element`, which follows.

```
val number: _ lr1state -> int
```

The states of the LR(1) automaton are numbered (from 0 and up).

```
type element =
  | Element: 'a lr1state * 'a * Lexing.position * Lexing.position -> element
```

The type `element` describes one entry in the stack of the LR(1) automaton. In a stack element of the form `Element (s, v, startp, endp)`, `s` is a (non-initial) state and `v` is a semantic value. The value `v` is associated with the incoming symbol $A$ of the state `s`. In other words, the value `v` was pushed onto the stack just before the state `s` was entered. Thus, for some type `'a`, the state `s` has type `'a lr1state` and the value `v` has type `'a`. The positions `startp` and `endp` delimit the fragment of the input text that was reduced to the symbol $A$.

In order to do anything useful with the value `v`, one must gain information about the type `'a`, by inspection of the state `s`. So far, the type `'a lr1state` is abstract, so there is no way of inspecting `s`. The inspection API (§9.3) offers further tools for this purpose.

```
type stack =
  element stream
```

A parser stack can be viewed as a stream of elements, where the first element of the stream is the topmost element of the stack. (The type `'a stream` is defined in the module `MenhirLib.General`.) This stream is empty if the parser is in an initial state, and non-empty otherwise. In the latter case, the current state of the LR(1) automaton is found in the topmost stack element.

```
val stack: env -> stack
```

The function `stack` offers a view of the parser's stack as a stream of elements. This stream is computed on-demand. (The internal representation of the stack may be different, so a conversion is necessary.) Invoking the function `stack`, and demanding the next element of the stream, takes constant time.

```
val positions: env -> Lexing.position * Lexing.position
```

The function `positions` returns the start and end positions of the current lookahead token. If invoked in an initial state, this function returns a pair of twice the initial position that was passed as an argument to `main`.

```
val has_default_reduction: env -> bool
```

The function `has_default_reduction` tells whether the parser is about to perform a default reduction. In particular, when applied to an environment `env` taken from a result of the form `AboutToReduce (env, prod)`, it tells whether the reduction that is about to take place is a default reduction.

### 9.3 Inspection API

If `--inspection` is set, Menhir offers an inspection API in addition to the monolithic and incremental APIs. Like the incremental API, the inspection API is found in the sub-module `MenhirInterpreter`. It offers the following types and functions.

The type `'a terminal` is a generalized algebraic data type (GADT). A value of type `'a terminal` represents a terminal symbol (without a semantic value). The index `'a` is the type of the semantic values associated with this symbol. For instance, if the grammar contains the declarations `%token A` and `%token<int> B`, then the generated module `MenhirInterpreter` contains the following definition:

```
type _ terminal =
| T_A : unit terminal
| T_B : int terminal
```

The data constructors are named after the terminal symbols, prefixed with "`T_`".

The type `'a nonterminal` is also a GADT. A value of type `'a nonterminal` represents a nonterminal symbol (without a semantic value). The index `'a` is the type of the semantic values associated with this symbol. For instance, if `main` is the only nonterminal symbol, then the generated module `MenhirInterpreter` contains the following definition:

```
type _ nonterminal =
| N_main : thing nonterminal
```

The data constructors are named after the nonterminal symbols, prefixed with "`N_`".

The type `'a symbol` is the disjoint union of the types `'a terminal` and `'a nonterminal`. In other words, a value of type `'a symbol` represents a terminal or nonterminal symbol (without a semantic value). This type is (always) defined as follows:

```
type 'a symbol =
  | T : 'a terminal -> 'a symbol
  | N : 'a nonterminal -> 'a symbol
```

The type `xsymbol` is an existentially quantified version of the type `'a symbol`. It is useful in situations where the index `'a` is not statically known. It is (always) defined as follows:

```
type xsymbol =
  | X : 'a symbol -> xsymbol
```

The type `item` describes an LR(0) item, that is, a pair of a production `prod` and an index `i` into the right-hand side of this production. If the length of the right-hand side is `n`, then `i` is comprised between 0 and `n`, inclusive.

```
type item =
    production * int
```

The following functions implement total orderings on the types `_ terminal`, `_ nonterminal`, `xsymbol`, `production`, and `item`.

```
val compare_terminals: _ terminal -> _ terminal -> int
val compare_nonterminals: _ nonterminal -> _ nonterminal -> int
val compare_symbols: xsymbol -> xsymbol -> int
val compare_productions: production -> production -> int
val compare_items: item -> item -> int
```

The function `incoming_symbol` maps a (non-initial) LR(1) state `s` to its incoming symbol, that is, the symbol that the parser must recognize before it enters the state `s`.

```
val incoming_symbol: 'a lr1state -> 'a symbol
```

This function can be used to gain access to the semantic value `v` in a stack element `Element (s, v, _, _)`. Indeed, by case analysis on the symbol `incoming_symbol s`, one gains information about the type `'a`, hence one obtains the ability to do something useful with the value `v`.

The function `items` maps a (non-initial) LR(1) state `s` to its LR(0) *core*, that is, to the underlying set of LR(0) items. This set is represented as a list, whose elements appear in an arbitrary order. This set is *not* closed under $\epsilon$-transitions.

```
val items: _ lr1state -> item list
```

The functions `lhs` and `rhs` map a production `prod` to its left-hand side and right-hand side, respectively. The left-hand side is always a nonterminal symbol, hence always of the form `N _`. The right-hand side is a (possibly empty) sequence of (terminal or nonterminal) symbols.

```
val lhs: production -> xsymbol
val rhs: production -> xsymbol list
```

The function `nullable`, applied to a non-terminal symbol, tells whether this symbol is nullable. A nonterminal symbol is nullable if and only if it produces the empty word $\epsilon$.

```
val nullable: _ nonterminal -> bool
```

The function call `first nt t` tells whether the *FIRST* set of the nonterminal symbol `nt` contains the terminal symbol `t`. That is, it returns `true` if and only if `nt` produces a word that begins with `t`. The function `xfirst` is identical to `first`, except it expects a first argument of type `xsymbol` instead of `_ terminal`.

```
val first: _ nonterminal -> _ terminal -> bool
val xfirst: xsymbol -> _ terminal -> bool
```

The function `foreach_terminal` enumerates the terminal symbols, including the special symbol **error**. The function `foreach_terminal_but_error` enumerates the terminal symbols, excluding **error**.

```
val foreach_terminal:           (xsymbol -> 'a -> 'a) -> 'a -> 'a
val foreach_terminal_but_error: (xsymbol -> 'a -> 'a) -> 'a -> 'a
```

## 10. Error handling: the traditional way

Menhir's traditional error handling mechanism is considered deprecated: although it is still supported for the time being, it might be removed in the future. We recommend setting up an error handling mechanism using the new tools offered by Menhir (§11).

***Error handling*** Menhir's error traditional handling mechanism is inspired by that of `yacc` and `ocamlyacc`, but is not identical. A special **error** token is made available for use within productions. The LR automaton is constructed exactly as if **error** was a regular terminal symbol. However, **error** is never produced by the lexical analyzer. Instead, when an error is detected, the current lookahead token is discarded and replaced with the **error** token, which becomes the current lookahead token. At this point, the parser enters *error handling* mode.

In error handling mode, automaton states are popped off the automaton's stack until a state that can *act* on **error** is found. This includes *both* shift *and* reduce actions. (`yacc` and `ocamlyacc` do not trigger reduce actions on **error**. It is somewhat unclear why this is so.)

When a state that can reduce on **error** is found, reduction is performed. Since the lookahead token is still **error**, the automaton remains in error handling mode.

When a state that can shift on **error** is found, the **error** token is shifted. At this point, the parser returns to normal mode.

When no state that can act on **error** is found on the automaton's stack, the parser stops and raises the exception `Error`. This exception carries no information. The position of the error can be obtained by reading the lexical analyzer's environment record.

***Error recovery*** `ocamlyacc` offers an error recovery mode, which is entered immediately after an **error** token was successfully shifted. In this mode, tokens are repeatedly taken off the input stream and discarded until an acceptable token is found. This feature is no longer offered by Menhir.

***Error-related keywords*** The following keyword is made available to semantic actions.

When the `$syntaxerror` keyword is evaluated, evaluation of the semantic action is aborted, so that the current reduction is abandoned; the current lookahead token is discarded and replaced with the **error** token; and error handling mode is entered. Note that there is no mechanism for inserting an **error** token *in front of* the current lookahead token, even though this might also be desirable. It is unclear whether this keyword is useful; it might be suppressed in the future.

## 11. Error handling: the new way

Menhir's incremental API (§9.2) allows taking control when an error is detected. Indeed, as soon as an invalid token is detected, the parser produces a checkpoint of the form `HandlingError _`. At this point, if one decides to let the parser proceed, by just calling `resume`, then Menhir enters its traditional error handling mode (§10). Instead, however, one can decide to take control and perform error handling or error recovery in any way one pleases. One can, for instance, build and display a diagnostic message, based on the automaton's current stack and/or state. Or, one could modify the input stream, by inserting or deleting tokens, so as to suppress the error, and resume normal parsing. In principle, the possibilities are endless.

An apparently simple-minded approach to error reporting, proposed by Jeffery [9] and further explored by Pottier [17], consists in selecting a diagnostic message (or a template for a diagnostic message) based purely on the current state of the automaton.

In this approach, one determines, ahead of time, which are the "error states" (that is, the states in which an error can be detected), and one prepares, for each error state, a diagnostic message. Because state numbers are fragile (they change when the grammar evolves), an error state is identified not by its number, but by an input sentence that leads to it: more precisely, by an input sentence which causes an error to be detected in this state. Thus, one maintains a set of pairs of an erroneous input sentence and a diagnostic message.

Menhir defines a file format, the `.messages` file format, for representing this information (§11.1), and offers a set of tools for creating, maintaining, and exploiting `.messages` files (§11.2). Once one understands these tools,

```
grammar: TYPE UID
grammar: TYPE OCAMLTYPE UID PREC

# A (handwritten) comment.

Ill-formed declaration.
Examples of well-formed declarations:
  %type <Syntax.expression> expression
  %type <int> date time
```

**Figure 16.** An entry in a `.messages` file

there remains to write a collection of diagnostic messages, a more subtle task than one might think (§11.3), and to glue everything together (§11.4).

In this approach to error handling, as in any other approach, one must understand exactly when (that is, in which states) errors are detected. This in turn requires understanding how the automaton is constructed. Menhir's construction technique is not Knuth's canonical LR(1) technique [12], which is usually too expensive to be practical. Instead, Menhir *merges* states [16] and introduces so-called *default reductions*. These techniques *defer* error detection by allowing extra reductions to take place before an error is detected. The impact of these alterations must be taken into account when writing diagnostic messages (§11.3).

In this approach to error handling, the special **error** token is not used. It should not appear in the grammar. Similarly, the `$syntaxerror` keyword should not be used.

### 11.1 The `.messages` file format

A `.messages` file is a text file. Comment lines, which begin with a # character, are ignored everywhere. As is evident in the following description, blank lines are significant: they are used as separators between entries and within an entry.

A `.messages` file is composed of a list of entries. Two entries are separated by one or more blank lines. Each entry consists of one or more input sentences, followed with one or more blank lines, followed with a message. The syntax of an input sentence is described in §8.1. A message is arbitrary text, but cannot contain a blank line. We stress that there cannot be a blank line between two sentences (if there is one, Menhir becomes confused and may complain about some word not being "a known non-terminal symbol").

As an example, Figure 16 shows a valid entry, taken from Menhir's own `.messages` file. This entry contains two input sentences, which lead to errors in two distinct states. A single message is associated with these two error states.

Several commands, described next (§11.2), produce `.messages` files where each input sentence is followed with an auto-generated comment, marked with ##. This special comment indicates in which state the error is detected, and is supposed to help the reader understand what it means to be in this state: What has been read so far? What is expected next?

As an example, the previous entry, decorated with auto-generated comments, is shown in Figure 17. (We have manually wrapped the lines that did not fit in this document.)

An auto-generated comment begins with the number of the error state that is reached via this input sentence.

Then, the auto-generated comment shows the LR(1) items that compose this state, in the same format as in an `.automaton` file. these items offer a description of the past (that is, what has been read so far) and the future (that is, which terminal symbols are allowed next).

Finally, the auto-generated comment shows what is known about the stack when the automaton is in this state. (This can be deduced from the LR(1) items, but is more readable if shown separately.)

```
grammar: TYPE UID
##
## Ends in an error in state: 1.
##
## declaration -> TYPE . OCAMLTYPE separated_nonempty_list(option(COMMA),
##   strict_actual) [ TYPE TOKEN START RIGHT PUBLIC PERCENTPERCENT PARAMETER
##   ON_ERROR_REDUCE NONASSOC LEFT INLINE HEADER EOF COLON ]
##
## The known suffix of the stack is as follows:
## TYPE
##
grammar: TYPE OCAMLTYPE UID PREC
##
## Ends in an error in state: 5.
##
## strict_actual -> symbol . loption(delimited(LPAREN,separated_nonempty_list
##   (COMMA,strict_actual),RPAREN)) [ UID TYPE TOKEN START STAR RIGHT QUESTION
##   PUBLIC PLUS PERCENTPERCENT PARAMETER ON_ERROR_REDUCE NONASSOC LID LEFT
##   INLINE HEADER EOF COMMA COLON ]
##
## The known suffix of the stack is as follows:
## symbol
##

# A (handwritten) comment.

Ill-formed declaration.
Examples of well-formed declarations:
  %type <Syntax.expression> expression
  %type <int> date time
```

**Figure 17.** An entry in a `.messages` file, decorated with auto-generated comments

In a canonical LR(1) automaton, the LR(1) items offer an exact description of the past and future. However, in a noncanonical automaton, which is by default what Menhir produces, the situation is more subtle. The lookahead sets can be over-approximated, so the automaton can perform one or more "spurious reductions" before an error is detected. As a result, the LR(1) items in the error state offer a description of the future that may be both incorrect (that is, a terminal symbol that appears in a lookahead set is not necessarily a valid continuation) and incomplete (that is, a terminal symbol that does not appear in any lookahead set may nevertheless be a valid continuation). More details appear further on (§11.3).

In order to attract the user's attention to this issue, if an input sentence causes one or more spurious reductions, then the auto-generated comment contains a warning about this fact. This mechanism is not completely foolproof, though, as it may be the case that one particular sentence does not cause any spurious reductions (hence, no warning appears), yet leads to an error state that can be reached via other sentences that do involve spurious reductions.

## 11.2 Maintaining `.messages` files

Ideally, the set of input sentences in a `.messages` file should be correct (that is, every sentence causes an error on its last token), irredundant (that is, no two sentences lead to the same error state), and complete (that is, every error state is reached by some sentence).

Correctness and irredundancy are checked by the command `--compile-errors` *filename*, where *filename* is the name of a `.messages` file. This command fails if a sentence does not cause an error at all, or causes an error too early. It also fails if two sentences lead to the same error state. If the file is correct and irredundant, then (as its name suggests) this command compiles the `.messages` file down to an OCaml function, whose code is printed on the standard output channel. This function, named `message`, has type `int -> string`, and maps a state number to a message. It raises the exception `Not_found` if its argument is not the number of a state for which a message has been defined.

Completeness is checked via the commands `--list-errors` and `--compare-errors`. The former produces, from scratch, a complete set of input sentences, that is, a set of input sentences that reaches all error states. The latter compares two sets of sentences (more precisely, the two underlying sets of error states) for inclusion.

The command `--list-errors` first computes all possible ways of causing an error. From this information, it deduces a list of all error states, that is, all states where an error can be detected. For each of these states, it computes a (minimal) input sentence that causes an error in this state. Finally, it prints these sentences, in the `.messages` file format, on the standard output channel. Each sentence is followed with an auto-generated comment and with a dummy diagnostic message. The user should be warned that this algorithm may require large amounts of time (typically in the tens of seconds, possibly more) and memory (typically in the gigabytes, possibly more). It requires a 64-bit machine. (On a 32-bit machine, it works, but quickly hits a built-in size limit.) At the verbosity level `--log-automaton 2`, it displays some progress information and internal statistics on the standard error channel.

The command `--compare-errors` *filename1* `--compare-errors` *filename2* compares the `.messages` files *filename1* and *filename2*. Each file is read and internally translated to a mapping of states to messages. Menhir then checks that the left-hand mapping is a subset of the right-hand mapping. That is, if a state $s$ is reached by some sentence in *filename1*, then it should also be reached by some sentence in *filename2*. Furthermore, if the message associated with $s$ in *filename1* is not a dummy message, then the same message should be associated with $s$ in *filename2*.

To check that the sentences in *filename2* cover all error states, it suffices to (1) use `--list-errors` to produce a complete set of sentences, which one stores in *filename1*, then (2) use `--compare-errors` to compare *filename1* and *filename2*.

The command `--update-errors` *filename* is used to update the auto-generated comments in the `.messages` file *filename*. It is typically used after a change in the grammar (or in the command line options that affect the construction of the automaton). A new `.messages` file is produced on the standard output channel. It is identical to *filename*, except the auto-generated comments, identified by `##`, have been removed and re-generated.

The command `--echo-errors` *filename* is used to filter out all comments, blank lines, and messages from the `.messages` file *filename*. The input sentences, and nothing else, are echoed on the standard output channel. As an example application, one could then translate the sentences to concrete syntax and create a collection of source files that trigger every possible syntax error.

The command `--interpret-error` is analogous to `--interpret`. It causes Menhir to act as an interpreter. Menhir reads sentences off the standard input channel, parses them, and displays the outcome. This switch can be usefully combined with `--trace`. The main difference between `--interpret` and `--interpret-error` is that, when the latter command is used, Menhir expects the input sentence to cause an error on its last token, and displays information about the state in which the error is detected, in the form of a `.messages` file entry. This can be used to quickly find out exactly what error is caused by one particular input sentence.

```
%token ID ARROW LPAREN RPAREN COLON SEMICOLON
%start<unit> program
%%
typ0: ID | LPAREN typ1 RPAREN {}
typ1: typ0 | typ0 ARROW typ1  {}
declaration: ID COLON typ1    {}
program:
| LPAREN declaration RPAREN
| declaration SEMICOLON       {}
```

**Figure 18.** A grammar where one error state is difficult to explain

```
program: ID COLON ID LPAREN
##
## Ends in an error in state: 8.
##
## typ1 -> typ0 . [ SEMICOLON RPAREN ]
## typ1 -> typ0 . ARROW typ1 [ SEMICOLON RPAREN ]
##
## The known suffix of the stack is as follows:
## typ0
##
```

**Figure 19.** A problematic error state in the grammar of Figure 18, due to over-approximation

### 11.3 Writing accurate diagnostic messages

One might think that writing a diagnostic message for each error state is a straightforward (if lengthy) task. In reality, it is not so simple.

***A state, not a sentence*** The first thing to keep in mind is that a diagnostic message is associated with a *state* $s$, as opposed to a sentence. An entry in a `.messages` file contains a sentence $w$ that leads to an error in state $s$. This sentence is just one way of causing an error in state $s$; there may exist many other sentences that also cause an error in this state. The diagnostic message should not be specific of the sentence $w$: it should make sense regardless of how the state $s$ is reached.

As a rule of thumb, when writing a diagnostic message, one should (as much as possible) ignore the example sentence $w$ altogether, and concentrate on the description of the state $s$, which appears as part of the auto-generated comment.

The LR(1) items that compose the state $s$ offer a description of the past (that is, what has been read so far) and the future (that is, which terminal symbols are allowed next). A diagnostic message should be designed based on this description.

***The problem of over-approximated lookahead sets*** As pointed out earlier (§11.1), in a noncanonical automaton, the lookahead sets in the LR(1) items can be both over- and under-approximated. One must be aware of this phenomenon, otherwise one runs the risk of writing a diagnostic message that proposes too many or too few continuations.

As an example, let us consider the grammar in Figure 18. According to this grammar, a "program" is either a declaration between parentheses or a declaration followed with a semicolon. A "declaration" is an identifier, followed with a colon, followed with a type. A "type" is an identifier, a type between parentheses, or a function type in the style of OCaml.

```
%token ID ARROW LPAREN RPAREN COLON SEMICOLON
%start<unit> program
%%
typ0: ID | LPAREN typ1(RPAREN) RPAREN           {}
typ1(phantom): typ0 | typ0 ARROW typ1(phantom) {}
declaration(phantom): ID COLON typ1(phantom)    {}
program:
| LPAREN declaration(RPAREN) RPAREN
| declaration(SEMICOLON)  SEMICOLON            {}
```

**Figure 20.** Splitting the problematic state of Figure 19 via selective duplication

```
%token ID ARROW LPAREN RPAREN COLON SEMICOLON
%start<unit> program
%on_error_reduce typ1
%%
typ0: ID | LPAREN typ1 RPAREN {}
typ1: typ0 | typ0 ARROW typ1  {}
declaration: ID COLON typ1    {}
program:
| LPAREN declaration RPAREN
| declaration SEMICOLON       {}
```

**Figure 21.** Avoiding the problematic state of Figure 19 via reductions on error

The (noncanonical) automaton produced by Menhir for this grammar has 17 states. Using `--list-errors`, we find that an error can be detected in 10 of these 17 states. By manual inspection of the auto-generated comments, we find that for 9 out of these 10 states, writing an accurate diagnostic message is easy. However, one problematic state remains, namely state 8, shown in Figure 19.

In this state, a (level-0) type has just been read. One valid continuation, which corresponds to the second LR(1) item in Figure 19, is to continue this type: the terminal symbol `ARROW`, followed with a (level-1) type, is a valid continuation. Now, the question is, what other valid continuations are there? By examining the first LR(1) item in Figure 19, it may look as if both `SEMICOLON` and `RPAREN` are valid continuations. However, this cannot be the case. A moment's thought reveals that *either* we have seen an opening parenthesis `LPAREN` at the very beginning of the program, in which case we definitely expect a closing parenthesis `RPAREN`; *or* we have not seen one, in which case we definitely expect a semicolon `SEMICOLON`. It is *never* the case that *both* `SEMICOLON` and `RPAREN` are valid continuations!

In fact, the lookahead set in the first LR(1) item in Figure 19 is over-approximated. State 8 in the noncanonical automaton results from merging two states in the canonical automaton.

In such a situation, one cannot write an accurate diagnostic message. Knowing that the automaton is in state 8 does not give us a precise view of the valid continuations. Some valuable information (that is, whether we have seen an opening parenthesis `LPAREN` at the very beginning of the program) is buried in the automaton's stack.

How can one work around this problem? Let us suggest three options.

***Blind duplication of states*** One option would be to build a canonical automaton by using the `--canonical` switch. In this example, one would obtain a 27-state automaton, where the problem has disappeared. However, this option is rarely viable, as it duplicates many states without good reason.

***Selective duplication of states*** A second option is to manually cause just enough duplication to remove the problematic over-approximation. In our example, we wish to distinguish two kinds of types and declarations,

```
program: ID COLON ID LPAREN
##
## Ends in an error in state: 15.
##
## program -> declaration . SEMICOLON [ # ]
##
## The known suffix of the stack is as follows:
## declaration
##
## WARNING: This example involves spurious reductions.
## This implies that, although the LR(1) items shown above provide an
## accurate view of the past (what has been recognized so far), they
## may provide an INCOMPLETE view of the future (what was expected next).
## In state 8, spurious reduction of production typ1 -> typ0
## In state 11, spurious reduction of production declaration -> ID COLON typ1
##
```

**Figure 22.** A problematic error state in the grammar of Figure 21, due to under-approximation

namely those that must be followed with a closing parenthesis, and those that must be followed with a semicolon. We create such a distinction by parameterizing `typ1` and `declaration` with a phantom parameter. The modified grammar is shown in Figure 20. The phantom parameter does not affect the language that is accepted: for instance, the nonterminal symbols `declaration(SEMICOLON)` and `declaration(RPAREN)` generate the same language as `declaration` in the grammar of Figure 18. Yet, by giving distinct names to these two symbols, we force the construction of an automaton where more states are distinguished. In this example, Menhir produces a 23-state automaton. Using `--list-errors`, we find that an error can be detected in 11 of these 23 states, and by manual inspection of the auto-generated comments, we find that for each of these 11 states, writing an accurate diagnostic message is easy. In summary, we have selectively duplicated just enough states so as to split the problematic error state into two non-problematic error states.

***Reductions on error*** A third and last option is to introduce an **%on_error_reduce** declaration (§4.1.7) so as to prevent the detection of an error in the problematic state 8. We see in Figure 19 that, in state 8, the production `typ1` $\rightarrow$ `typ0` is ready to be reduced. If we could force this reduction to take place, then the automaton would move to some other state where it would be clear which of `SEMICOLON` and `RPAREN` is expected. We achieve this by marking `typ1` as "reducible on error". The modified grammar is shown in Figure 21. For this grammar, Menhir produces a 17-state automaton. (This is the exact same automaton as for the grammar of Figure 18, except 2 of the 17 states have received extra reduction actions.) Using `--list-errors`, we find that an error can be detected in 9 of these 17 states. The problematic state, namely state 8, is no longer an error state! The problem has vanished.

***The problem of under-approximated lookahead sets*** The third option seems by far the simplest of all, and is recommended in many situations. However, it comes with a caveat. There may now exist states whose lookahead sets are under-approximated, in a certain sense. Because of this, there is a danger of writing an incomplete diagnostic message, one that does not list all valid continuations.

To see this, let us look again at the sentence `ID COLON ID LPAREN`. In the grammar and automaton of Figure 18, this sentence takes us to the problematic state 8, shown in Figure 19. In the grammar and automaton of Figure 21, because more reduction actions are carried out before the error is detected, this sentence takes us to state 15, shown in Figure 22.

When writing a diagnostic message for state 15, one might be tempted to write: "Up to this point, a declaration has been recognized. At this point, a semicolon is expected". Indeed, by examining the sole LR(1) item in state 15, it looks as if SEMICOLON is the only permitted continuation. However, this is not the case. Another valid continuation is ARROW: indeed, the sentence ID COLON ID ARROW ID SEMICOLON forms a valid program. In fact, if the first token following ID COLON ID is ARROW, then in state 8 this token is shifted, so the two reductions that take us from state 8 through state 11 to state 15 never take place. This is why, even though ARROW does not appear in state 15 as a valid continuation, it nevertheless is a valid continuation of ID COLON ID. The warning produced by Menhir, shown in Figure 22, is supposed to attract attention to this issue.

Another way to explain this issue is to point out that, by declaring %on_error_reduce typ1, we make a choice. When the parser reads a type and finds an invalid token, it decides that this type is finished, even though, in reality, this type could be continued with ARROW .... This in turn causes the parser to perform another reduction and consider the current declaration finished, even though, in reality, this declaration could be continued with ARROW ....

In summary, when writing a diagnostic message for state 15, one should take into account the fact that this state can be reached via spurious reductions and (therefore) SEMICOLON may not be the only permitted continuation. One way of doing this, without explicitly listing all permitted continuations, is to write: "Up to this point, a declaration has been recognized. If this declaration is complete, then at this point, a semicolon is expected".

### 11.4 A working example

The CompCert verified compiler offers a real-world example of this approach to error handling. The "pre-parser" is where syntax errors are detected: see cparser/pre_parser.mly. A database of erroneous input sentences and (templates for) diagnostic messages is stored in cparser/handcrafted.messages. It is compiled, using --compile-errors, to an OCaml file named cparser/pre_parser_messages.ml. The function Pre_parser_messages.message, which maps a state number to (a template for) a diagnostic message, is called from cparser/ErrorReports.ml, where we construct and display a full-fledged diagnostic message.

In CompCert, we allow a template for a diagnostic message to contain the special form $i, where i is an integer constant, understood as an index into the parser's stack. The code in cparser/ErrorReports.ml automatically replaces this special form with the fragment of the source text that corresponds to this stack entry. This mechanism is not built into Menhir; it is implemented in CompCert using Menhir's incremental API.

## 12. Coq back-end

Menhir is able to generate a parser that whose correctness can be formally verified using the Coq proof assistant [11]. This feature is used to construct the parser of the CompCert verified compiler [14].

Setting the --coq switch on the command line enables the Coq back-end. When this switch is set, Menhir expects an input file whose name ends in .vy and generates a Coq file whose name ends in .v.

Like a .mly file, a .vy file is a grammar specification, with embedded semantic actions. The only difference is that the semantic actions in a .vy file are expressed in Coq instead of OCaml. A .vy file otherwise uses the same syntax as a .mly file. CompCert's cparser/Parser.vy serves as an example.

Several restrictions are imposed when Menhir is used in --coq mode:

- The error handling mechanism (§10) is absent. The $syntaxerror keyword and the **error** token are not supported.
- Location information is not propagated. The $start* and $end* keywords (Figure 14) are not supported.
- **%parameter** (§4.1.2) is not supported.
- **%inline** (§5.3) is not supported.
- The standard library (§5.4) is not supported, of course, because its semantic actions are expressed in OCaml. If desired, the user can define an analogous library, whose semantic actions are expressed in Coq.
- Because Coq's type inference algorithm is rather unpredictable, the Coq type of every nonterminal symbol must be provided via a **%type** or **%start** declaration (§4.1.5, §4.1.6).

- Unless the proof of completeness has been deactivated using `--coq-no-complete`, the grammar must not have a conflict (not even a benign one, in the sense of §6.1). That is, the grammar must be LR(1). Conflict resolution via priority and associativity declarations (§4.1.4) is not supported. The reason is that there is no simple formal specification of how conflict resolution should work.

The generated file contains several modules:

- The module `Gram` defines the terminal and non-terminal symbols, the grammar, and the semantic actions.
- The module `Aut` contains the automaton generated by Menhir, together with a certificate that is checked by Coq while establishing the soundness and completeness of the parser.

The type of the terminal symbols is an inductive type, with one constructor for each terminal symbol. A terminal symbol per se does not carry a the semantic value. We also define the type `token` of tokens, that is, dependent pairs of a terminal symbol and a semantic value of an appropriate type for this symbol. We model the lexer as an object of type `Streams.Stream token`, that is, an infinite stream of tokens.

The proof of termination of an LR(1) parser in the case of invalid input seems far from obvious. We did not find such a proof in the literature. In an application such as CompCert [14], this question is not considered crucial. For this reason, we did not formally establish the termination of the parser. Instead, we use the "fuel" technique. The parser takes an additional parameter of type `nat` that indicates the maximum number of steps the parser is allowed to perform. In practice, after extracting the code to OCaml, one can use the standard trick of passing an infinite amount of fuel, defined in OCaml by `let rec inf = S inf`.

Parsing can have three different outcomes, represented by the type `parse_result`. (This definition is implicitly parameterized over the initial state `init`. We omit the details here.)

```
Inductive parse_result :=
| Fail_pr:    parse_result
| Timeout_pr: parse_result
| Parsed_pr:
    symbol_semantic_type (NT (start_nt init)) ->
    Stream token ->
    parse_result.
```

The outcome `Fail_pr` means that parsing has failed because of a syntax error. (If the completeness of the parser with respect to the grammar has been proved, this implies that the input is invalid). The outcome `Timeout_pr` means that the fuel has been exhausted. Of course, this cannot happen if the parser was given an infinite amount of fuel, as suggested above. The outcome `Parsed_pr` means that the parser has succeeded in parsing a prefix of the input stream. It carries the semantic value that has been constructed for this prefix, as well as the remainder of the input stream.

For each entry point `entry` of the grammar, Menhir generates a parsing function `entry`, whose type is `nat -> Stream token -> parse_result`.

Two theorems are provided, named `entry_point_correct` and `entry_point_complete`. The correctness theorem states that, if a word (a prefix of the input stream) is accepted, then this word is valid (with respect to the grammar) and the semantic value that is constructed by the parser is valid as well (with respect to the grammar). The completeness theorem states that if a word (a prefix of the input stream) is valid (with respect to the grammar), then (given sufficient fuel) it is accepted by the parser.

These results imply that the grammar is unambiguous: for every input, there is at most one valid interpretation. This is proved by another generated theorem, named `Parser.unambiguous`.

The parsers produced by Menhir's Coq back-end must be linked with a Coq library, which can be found in the CompCert tree [14, 13], in the `cparser/validator` subdirectory. CompCert can be used as an example if one wishes to use Menhir to generate a formally verified parser as part of some other project.

## 13. Comparison with `ocamlyacc`

Here is an incomplete list of the differences between `ocamlyacc` and Menhir. The list is roughly sorted by decreasing order of importance.

- Menhir allows the definition of a nonterminal symbol to be parameterized by other (terminal or nonterminal) symbols (§5.2). Furthermore, it offers a library of standard parameterized definitions (§5.4), including options, sequences, and lists. It offers some support for EBNF syntax, via the ?, +, and * modifiers.
- `ocamlyacc` only accepts LALR(1) grammars. Menhir accepts LR(1) grammars, thus avoiding certain artificial conflicts.
- Menhir's **%inline** keyword (§5.3) helps avoid or resolve some LR(1) conflicts without artificial modification of the grammar.
- Menhir explains conflicts (§6) in terms of the grammar, not just in terms of the automaton. Menhir's explanations are believed to be understandable by mere humans.
- Menhir offers an incremental API (in `--table` mode only) (§9.2). This means that the state of the parser can be saved at any point (at no cost) and that parsing can later be resumed from a saved state.
- In `--coq` mode, Menhir produces a parser whose correctness and completeness with respect to the grammar can be checked by Coq (§12).
- Menhir offers an interpreter (§8) that helps debug grammars interactively.
- Menhir allows grammar specifications to be split over multiple files (§5.1). It also allows several grammars to share a single set of tokens.
- Menhir produces reentrant parsers.
- Menhir is able to produce parsers that are parameterized by OCaml modules.
- `ocamlyacc` requires semantic values to be referred to via keywords: `$1`, `$2`, and so on. Menhir allows semantic values to be explicitly named.
- Menhir warns about end-of-stream conflicts (§6.4), whereas `ocamlyacc` does not. Menhir warns about productions that are never reduced, whereas, at least in some cases, `ocamlyacc` does not.
- Menhir offers an option to typecheck semantic actions *before* a parser is generated: see `--infer`.
- `ocamlyacc` produces tables that are interpreted by a piece of C code, requiring semantic actions to be encapsulated as OCaml closures and invoked by C code. Menhir offers a choice between producing tables and producing code. In either case, no C code is involved.
- Menhir makes OCaml's standard library module `Parsing` entirely obsolete. Access to locations is now via keywords (§7). Uses of `raise Parse_error` within semantic actions are deprecated. The function `parse_error` is deprecated. They are replaced with keywords (§10).
- Menhir's error handling mechanism (§10) is inspired by `ocamlyacc`'s, but is not guaranteed to be fully compatible. Error recovery, also known as re-synchronization, is not supported by Menhir.
- The way in which severe conflicts (§6) are resolved is not guaranteed to be fully compatible with `ocamlyacc`.
- Menhir warns about unused **%token**, **%nonassoc**, **%left**, and **%right** declarations. It also warns about **%prec** annotations that do not help resolve a conflict.
- Menhir accepts OCaml-style comments.
- Menhir allows **%start** and **%type** declarations to be condensed.
- Menhir allows two (or more) productions to share a single semantic action.
- Menhir produces better error messages when a semantic action contains ill-balanced parentheses.
- `ocamlyacc` ignores semicolons and commas everywhere. Menhir also ignores semicolons everywhere, but treats commas as significant. Commas are optional within **%token** declarations.

- `ocamlyacc` allows **%type** declarations to refer to terminal or non-terminal symbols, whereas Menhir requires them to refer to non-terminal symbols. Types can be assigned to terminal symbols with a **%token** declaration.

## 14. Questions and Answers

◇ **Is Menhir faster than `ocamlyacc`? What is the speed difference between `menhir` and `menhir --table`?** A (not quite scientific) benchmark suggests that the parsers produced by `ocamlyacc` and `menhir --table` have comparable speed, whereas those produced by `menhir` are between 2 and 5 times faster. This benchmark excludes the time spent in the lexer and in the semantic actions.

◇ **How do I write `Makefile` rules for Menhir?** This can be quite difficult, especially when `--infer` is used. Look at `demos/obsolete/Makefile.shared` or (better) use `ocamlbuild`, which has built-in compilation rules for OCaml and Menhir.

◇ **Menhir reports *more* shift/reduce conflicts than `ocamlyacc`! How come?** `ocamlyacc` sometimes merges two states of the automaton that Menhir considers distinct. This happens when the grammar is not LALR(1). If these two states happen to contain a shift/reduce conflict, then Menhir reports two conflicts, while `ocamlyacc` only reports one. Of course, the two conflicts are very similar, so fixing one will usually fix the other as well.

◇ **I do not use `ocamllex`. Is there an API that does not involve lexing buffers?** Like `ocamlyacc`, Menhir produces parsers whose monolithic API (§9.1) is intended for use with `ocamllex`. However, it is possible to convert them, after the fact, to a simpler, revised API. In the revised API, there are no lexing buffers, and a lexer is just a function from unit to tokens. Converters are provided by the library module `MenhirLib.Convert`. This can be useful, for instance, for users of `ulex`, the Unicode lexer generator. Also, please note that Menhir's incremental API (§9.2) does not mention the type `Lexing.lexbuf`. In this API, the parser expects to be supplied with triples of a token and start/end positions of type `Lexing.position`.

◇ **I need both %inline and non-%inline versions of a non-terminal symbol. Is this possible?** Define an **%inline** version first, then use it to define a non-**%inline** version, like this:

```
%inline ioption(X):  (* nothing *) { None } | x = X { Some x }
        option(X): o = ioption(X) { o }
```

This can work even in the presence of recursion, as illustrated by the following definition of (reversed, left-recursive, possibly empty) lists:

```
%inline irevlist(X):    (* nothing *) { [] } | xs = revlist(X) x = X { x :: xs }
        revlist(X): xs = irevlist(X) { xs }
```

The definition of `irevlist` is expanded into the definition of `revlist`, so in the end, `revlist` receives its normal, recursive definition. One can then view `irevlist` as a variant of `revlist` that is inlined one level deep.

◇ **Can I ship a generated parser while avoiding a dependency on `MenhirLib`?** Yes. One option is to use the code-based back-end (that is, to not use `--table`). In this case, the generated parser is self-contained. Another option is to use the table-based back-end (that is, use `--table`) and include a copy of the files `menhirLib.{ml,mli}` together with the generated parser. The command `menhir --suggest-menhirLib` will tell you where to find these source files.

◇ **Why is `$startpos` off towards the left? It seems to include some leading whitespace.** Indeed, as of 2015/11/04, the computation of positions has changed so as to match `ocamlyacc`'s behavior. As a result,

`$startpos` can now appear to be too far off to the left. This is explained in §7. In short, the solution is to use `$symbolstartpos` instead.

## 15. Technical background

After experimenting with Knuth's canonical LR(1) technique [12], we found that it *really* is not practical, even on today's computers. For this reason, Menhir implements a slightly modified version of Pager's algorithm [16], which merges states on the fly if it can be proved that no reduce/reduce conflicts will arise as a consequence of this decision. This is how Menhir avoids the so-called *mysterious* conflicts created by LALR(1) parser generators [6, section 5.7].

Menhir's algorithm for explaining conflicts is inspired by DeRemer and Pennello's [5] and adapted for use with Pager's construction technique.

By default, Menhir produces code, as opposed to tables. This approach has been explored before [3, 8]. Menhir performs some static analysis of the automaton in order to produce more compact code.

When asked to produce tables, Menhir performs compression via first-fit row displacement, as described by Tarjan and Yao [20]. Double displacement is not used. The action table is made sparse by factoring out an error matrix, as suggested by Dencker, Dürre, and Heuft [4].

The type-theoretic tricks that triggered our interest in LR parsers [18] are not implemented in Menhir. In the beginning, we did not implement them because the OCaml compiler did not at the time offer generalized algebraic data types (GADTs). Today, OCaml has GADTs, but, as the saying goes, "if it ain't broken, don't fix it".

The main ideas behind the Coq back-end are described in a paper by Jourdan, Pottier and Leroy [11].

## 16. Acknowledgements

Menhir's interpreter (`--interpret`) and table-based back-end (`--table`) were implemented by Guillaume Bau, Raja Boujbel, and François Pottier. The project was generously funded by Jane Street Capital, LLC through the "OCaml Summer Project" initiative.

Frédéric Bour provided motivation and an initial implementation for the incremental API and inspection API.

Jacques-Henri Jourdan designed and implemented the Coq back-end and did the Coq proofs for it.

Gabriel Scherer provided motivation for investigating Jeffery's technique.

## References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Andrew Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[3] Achyutram Bhamidipaty and Todd A. Proebsting. Very fast YACC-compatible parsers (for very little effort). *Software: Practice and Experience*, 28(2):181–190, 1998.

[4] Peter Dencker, Karl Dürre, and Johannes Heuft. Optimization of parser tables for portable compilers. *ACM Transactions on Programming Languages and Systems*, 6(4):546–572, 1984.

[5] Frank DeRemer and Thomas Pennello. Efficient computation of $LALR(1)$ look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, 1982.

[6] Charles Donnelly and Richard Stallman. *Bison*, 2015.

[7] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2000.

[8] R. Nigel Horspool and Michael Whitney. Even faster LR parsing. *Software: Practice and Experience*, 20(6):515–535, 1990.

[9] Clinton L. Jeffery. Generating LR syntax error messages from examples. *ACM Transactions on Programming Languages and Systems*, 25(5):631–640, 2003.

[10] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, 1979.

[11] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating $LR(1)$ parsers. In *European Symposium on Programming (ESOP)*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer, 2012.

[12] Donald E. Knuth. On the translation of languages from left to right. *Information & Control*, 8(6):607–639, 1965.

[13] Xavier Leroy. The CompCert C verified compiler. `https://github.com/AbsInt/CompCert`, 2014.

[14] Xavier Leroy. The CompCert C compiler. `http://compcert.inria.fr/`, 2015.

[15] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system*, 2005.

[16] David Pager. A practical general method for constructing $LR(k)$ parsers. *Acta Informatica*, 7:249–268, 1977.

[17] François Pottier. Reachability and error diagnosis in LR(1) automata. In *Journées Francophones des Langages Applicatifs (JFLA)*, 2016.

[18] François Pottier and Yann Régis-Gianas. Towards efficient, typed LR parsers. *Electronic Notes in Theoretical Computer Science*, 148(2):155–180, 2006.

[19] David R. Tarditi and Andrew W. Appel. *ML-Yacc User's Manual*, 2000.

[20] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.