Menhir Reference Manual
(version 20070322)

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 Objective Caml programming language [10], into parsers, again expressed in Objective Caml. It is based on Knuth's LR(1) parser construction technique [9]. It is strongly inspired by its precursors: `yacc` [8], `ML-Yacc` [13], and `ocamlyacc` [10], 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, 6]. They are also invited to have a look at the `demos` directory in Menhir's distribution.

At this stage, potential users should be warned about two facts. First, Menhir's feature set is not 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 and recovery mechanism, are still potentially subject to incompatible changes. Second, the present release is *alpha*-quality. There is much 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 Objective Caml code that is written to the `.ml` file.

`--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[oix]`, *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 Objective Caml modules that the grammar specification depends upon to have been compiled first. The file `demos/Makefile.shared` helps exploit the `--depend` 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`.

`--error-recovery`. This switch causes error recovery code to be generated. Error recovery, also known as re-synchronization, consists in dropping tokens off the input stream, after an error has been detected, until a token that can be shifted in the current state is found. This behavior is made optional because it is seldom exploited and requires extra code in the parser. See also §8.

`--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 Objective Caml 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.

`--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.

`--infer`. This switch causes the semantic actions to be checked for type consistency *before* the parser is generated. This is done by invoking the Objective Caml 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 Objective Caml 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.

`--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 Objective Caml 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`, which performs its own postprocessing.

`--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.

specification ::= *declaration* ... *declaration* **%%** *rule* ... *rule* [**%%** *Objective Caml code*]
declaration ::= **%{** *Objective Caml code* **%}**
%parameter < *uid* **:** *Objective Caml module type* >
%token [< *Objective Caml type* >] *uid* ... *uid*
%nonassoc *uid* ... *uid*
%left *uid* ... *uid*
%right *uid* ... *uid*
%type < *Objective Caml type* > *lid* ... *lid*
%start [< *Objective Caml type* >] *lid* ... *lid*
rule ::= [**%public**] [**%inline**] *lid* [**(** *id*, ..., *id* **)**] **:** [|] *group* | ... | *group*
group ::= *production* | ... | *production* { *Objective Caml code* } [**%prec** *id*]
production ::= *producer* ... *producer* [**%prec** *id*]
producer ::= [*lid* **=**] *actual*
actual ::= *id* [**(** *actual*, ..., *actual* **)**] [**?** | **+** | *****]

Figure 1. Syntax of grammar specifications

`--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.

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

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. They can be omitted otherwise.

Identifiers (*id*) coincide with Objective Caml identifiers, except they are not allowed to contain the quote (**'**) character. Following Objective Caml, 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 Objective Caml-style (surrounded with **(*** and ***)**, can be nested). Of course, inside Objective Caml code, only Objective Caml-style comments are allowed.

Objective Caml 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 Objective Caml code, surrounded with **%{** and **%}**. It is copied verbatim at the beginning of the `.ml` file. It typically contains Objective Caml **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* **:** *Objective Caml module type* $>$

causes the entire parser to become parameterized over the Objective Caml module *uid*, that is, to become an Objective Caml functor. 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 Objective Caml header are observed only when the functor is applied, not when it is defined.

4.1.3 Tokens

A declaration of the form:

%token [$<$ *Objective Caml 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 Objective Caml 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 $<$ *Objective Caml type* $>$ $lid_1 \ldots lid_n$

assigns an Objective Caml type to each of the nonterminal symbols $lid_1, \ldots, lid_n$. For start symbols, providing an Objective Caml 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 Objective Caml's type error messages.

4.1.6 Start symbols

A declaration of the form:

%start [$<$ *Objective Caml 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 Objective Caml 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.

4.2 Rules

Following the mandatory **%%** keyword, a sequence of rules is expected. Each rule defines a nonterminal symbol *id*. In its simplest form, a rule begins with *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 Objective Caml 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 Objective Caml 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 these semantic values via positional keywords of the form **$1**, **$2**, etc. This style is discouraged.

%prec ***annotations*** An annotation of the form **%prec** *uid* indicates that the precedence level of the production group is the level assigned to the symbol *uid* 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 simply consists of a terminal or nonterminal symbol. The optional actual parameters **(** *actual*, ..., *actual* **)** and the optional modifier (**?**, **+**, or *****) are explained further on (see §5.2 and Figure 2).

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):

```
%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).

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

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 }
```

Name	Recognizes	Produces	Comment
option(*X*)	$\epsilon \mid X$	α *option*, if $X : \alpha$	
ioption(*X*)	$\epsilon \mid X$	α *option*, if $X : \alpha$	(inlined)
boption(*X*)	$\epsilon \mid X$	*bool*	
loption(*X*)	$\epsilon \mid X$	α *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*	α, if $X : \alpha$	
terminated(*X*, *closing*)	*X closing*	α, if $X : \alpha$	
delimited(*opening*, *X*, *closing*)	*opening X closing*	α, if $X : \alpha$	
list(*X*)	a possibly empty sequence of *X*'s	α *list*, if $X : \alpha$	
nonempty_list(*X*)	a nonempty sequence of *X*'s	α *list*, if $X : \alpha$	
separated_list(*sep*, *X*)	a possibly empty sequence of *X*'s separated with *sep*'s	α *list*, if $X : \alpha$	
separated_nonempty_list(*sep*, *X*)	a nonempty sequence of *X*'s separated with *sep*'s	α *list*, if $X : \alpha$	

Figure 3. Summary of the standard library

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.

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:

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.

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`

```
%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

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 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*

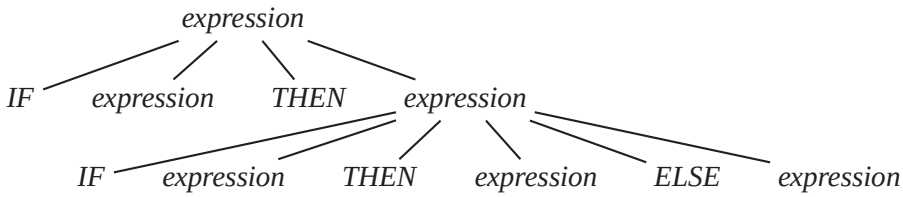

Figure 5. A partial derivation tree that justifies shifting

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

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

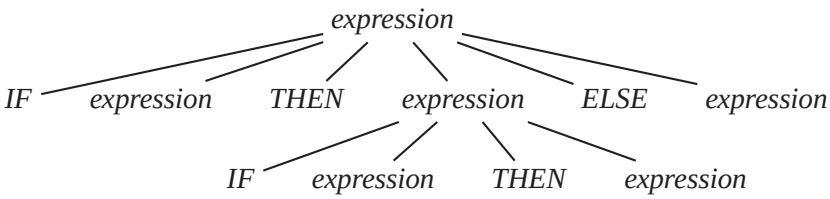

Figure 7. A partial derivation tree that justifies reducing

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

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 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* → *IF expression THEN expression*.

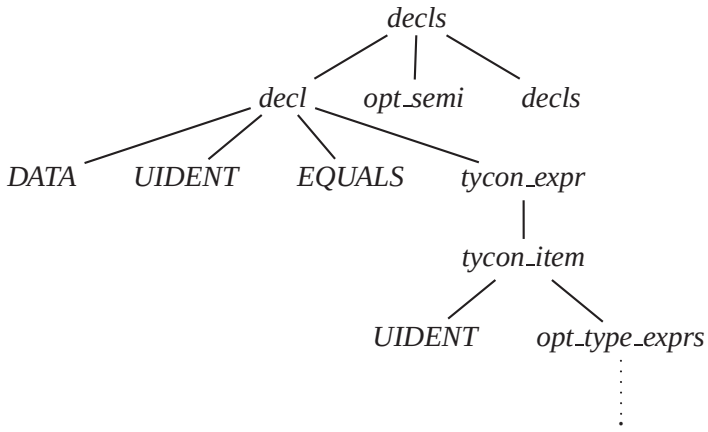

Figure 9. A partial derivation tree that justifies reducing

```
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

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 ϵ 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 ϵ, 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 $S' \rightarrow S$. The corresponding start state of the LR(1) automaton is composed of the LR(1) item $S' \rightarrow . \ S \ [\#]$. 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 `.conflicts` file, is shown in Figure 12. Explanations at the end of the `.conflicts` 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* → *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* → *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.

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

```
%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

`$startpos`	start position of the sentence derived out of the production that is being reduced
`$endpos`	end position of the sentence derived out of the production that is being reduced
`$startpos(` $i | id `)`	start position of the sentence derived out of the symbol whose semantic value is referred to as $i or *id*
`$endpos(` $i | id `)`	end position of the sentence derived out of the symbol whose semantic value is referred to as $i or *id*
`$startofs`	start offset of the sentence derived out of the production that is being reduced
`$endofs`	end offset of the sentence derived out of the production that is being reduced
`$startofs(` $i | id `)`	start offset of the sentence derived out of the symbol whose semantic value is referred to as $i or *id*
`$endofs(` $i | id `)`	end offset of the sentence derived out of the symbol whose semantic value is referred to as $i or *id*

Figure 14. Position-related keywords

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, documented in Figure 14, available to semantic actions. Note that these keywords are *not* available elsewhere—in particular, not within Objective Caml headers. Note also that Objective Caml's standard library module `Parsing` is deprecated. The functions that it offers *can* be called, but will return dummy positions.

8. Error handling and recovery

Error handling Menhir's error handling and recovery 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 either enters *error recovery* mode, if the `--error-recovery` switch was enabled at compile time, or 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 Error recovery mode is entered immediately after an **error** token was successfully shifted, and only if Menhir's `--error-recovery` switch was enabled when the parser was produced. In error recovery mode, tokens are repeatedly taken off the input stream and discarded until an acceptable token is found. A token

is acceptable if the current state has an action on that token. When an acceptable token is found, the parser returns to normal mode and the action takes place. Error recovery is also known as *re-synchronization*.

Error recovery mode is peculiar, in that it can cause non-termination if the token stream is infinite. In practice, token streams often *are* infinite, due to an `ocamllex` peculiarity: every `ocamllex`-generated analyzer that maps the **eof** pattern to an *EOF* token will produce an infinite stream of *EOF* tokens, even if the underlying text that is being scanned is finite. In order to address this issue, Menhir attributes special meaning to the token named *EOF*, if there is one in the grammar specification, when `--error-recovery` is enabled. It checks that every automaton state that can be reached when in error recovery mode accepts this token, and issues a warning otherwise. This ensures that the parser always terminates.

Error-related keywords A couple of error-related keywords are 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.

The `$previouserror` keyword evaluates to an integer value, and indicates how many tokens were successfully shifted since the last **error** token was shifted. This allows heuristics such as *"when a new error is detected, do not display a new error message unless the previous error is ancient enough"* to be implemented if and where desired.

When are errors detected? An error is detected when the current state of the automaton has no action on the current lookahead token. Thus, understanding exactly when errors are detected requires understanding how the automaton is constructed. Menhir's construction technique is *not* Knuth's canonical LR(1) technique [9], which is too expensive to be practical. Instead, Menhir *merges* states [11] and introduces so-called *default reductions*. Both techniques can *defer* error detection by allowing extra reductions to take place before an error is detected. All LALR(1) parser generators exhibit the same problem.

9. A 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 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 Objective Caml 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 Objective Caml closures and invoked by C code. Menhir produces no tables and requires no C stubs: the generated parser is pure Objective Caml code.
- Menhir makes Objective Caml'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 (§8).
- Menhir's error handling and error recovery mechanisms (§8) are inspired by `ocamlyacc`'s, but are not guaranteed to be fully compatible. Error recovery, also known as re-synchronization, is now optional.
- 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 Objective Caml-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.

10. Questions and Answers

◇ **Turning on `--infer` broke my `Makefile`! What should I do?** Look at `demos/Makefile.shared`. It is meant to be re-used without change. If it does not suit your needs, you can copy parts of it into your own `Makefile`, or submit suggestions for improvement.

◇ **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.

11. Technical background

After experimenting with Knuth's canonical LR(1) technique [9], 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 [11], 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 [5, section 5.7].

Menhir's algorithm for explaining conflicts is inspired by DeRemer and Pennello's [4] and adapted for use with Pager's construction technique.

Menhir produces code, as opposed to tables. This approach has been explored before [3, 7]. Menhir performs some static analysis of the automaton in order to produce more compact code.

The type-theoretic tricks that triggered our interest in LR parsers [12] are not implemented in Menhir, because the Objective Caml compiler does not yet offer the required features. This will hopefully be addressed in the future.

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 & Experience*, 28(2):181–190, February 1998.

[4] 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.

[5] Charles Donnelly and Richard Stallman. *Bison*, September 2005.

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

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

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

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

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

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

[12] François Pottier and Yann Régis-Gianas. Towards efficient, typed LR parsers. In *ACM Workshop on ML*, Electronic Notes in Theoretical Computer Science, pages 149–173, September 2005.

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