

# Syntactic Analysis with Menhir

Letterio Galletta

# Agenda

- Examples of hand-coded parsers
- Brief tutorial on menhir

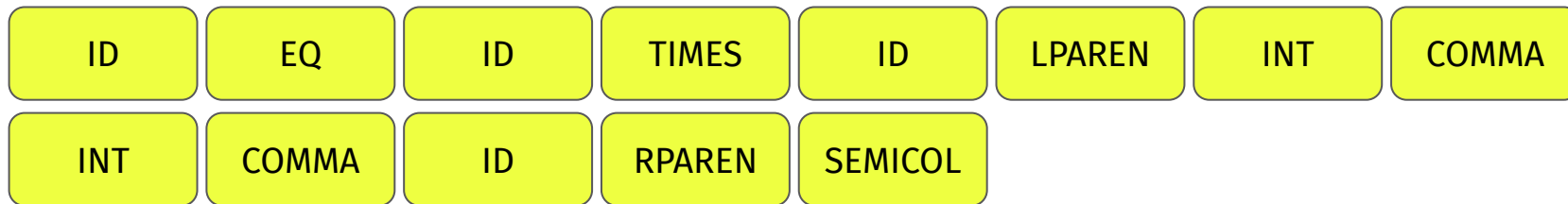
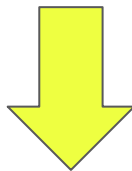
## References

- [Menhir homepage](#)
- [Menhir reference manual](#)
- [Parsing with OCamllex and Menhir](#)

# Syntactic analysis 1 (parsing)

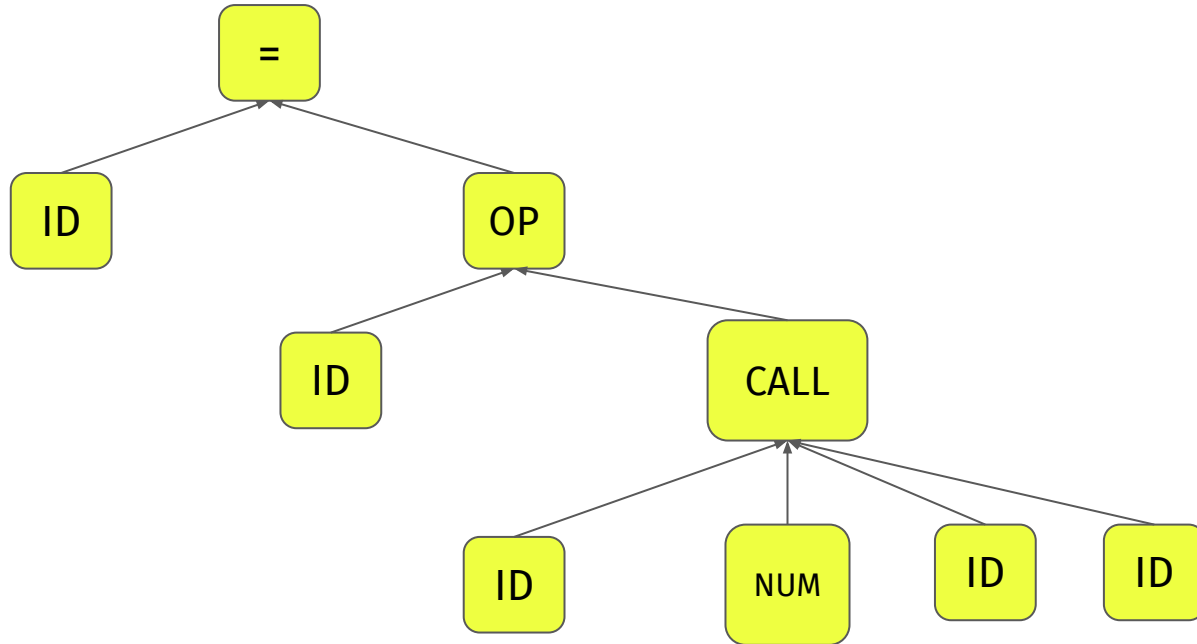
Translate a stream of tokens to a intermediate representation

```
result _ = _i_*_bar (0, _42, _q);
```



# Syntactic analysis 2 (parsing)

Translate a stream of tokens to a intermediate representation



Symbol	Info
result	local, INT
i	local, INT
bar	global, FUNC
q	local, INT

# Syntactic analysis (parsing)

## Goals:

- Verify that program are well written
- Build a structured representation of the code (AST) that can easily manipulated from the rest of the compiler

## Describing syntax:

- Syntax rules are usually specified by context-free grammars

# Implementing parsers

Two approaches:

- Use parser generators, e.g., bison, flex, etc.

They take as input a description of grammar (LR(1), LALR(1)) and code to run when a sentence is recognised

- Hand-coded parsers

You need to implement all the details for recognize a sentence according to a rule of the grammar

# Hand-coded parsers

Typically, they are top-down recursive recognizer that consist of

- An handle to the lexer to obtain the next token
- A function for each non-terminal symbol implementing the right-handside of the rule and the actions to build the intermediate representation
- A parsing function is usually a big switch on the current token (or some lookahead tokens)

# Hand-coded parsers: examples

Toy compilers:

- Lox language from the book [Crafting Interpreters](#)

See [Parser.java](#)

- Subset of Pascal language from the book [Writing Compilers and Interpreters](#)

See code of [Chapter 5](#) package Frontend



# Hand-coded parsers: examples

Real world compilers:

- Golang, see [parser.go](https://github.com/golang/parser)
- Rust, see [librustc\\_parse/lib.rs](https://github.com/librustc_parse/lib.rs)
- Python (CPython implementation), see [Parser/parser.c](https://github.com/python/cpython/blob/master/Parser/parser.c)
- Clang, see [Parser/Parser.h](https://github.com/llvm/llvm-project/blob/main/clang/Parser/Parser.h) and [Parser/Parser.cpp](https://github.com/llvm/llvm-project/blob/main/clang/Parser/Parser.cpp)
- Many others, e.g., Solidity, V8, Scala, ...

# Implementing Parser Automatically

Syntactic specification  
provided by the user

Rules (grammar  
production)

+

Actions (code)

Tool output

Parsing tables

Parser code

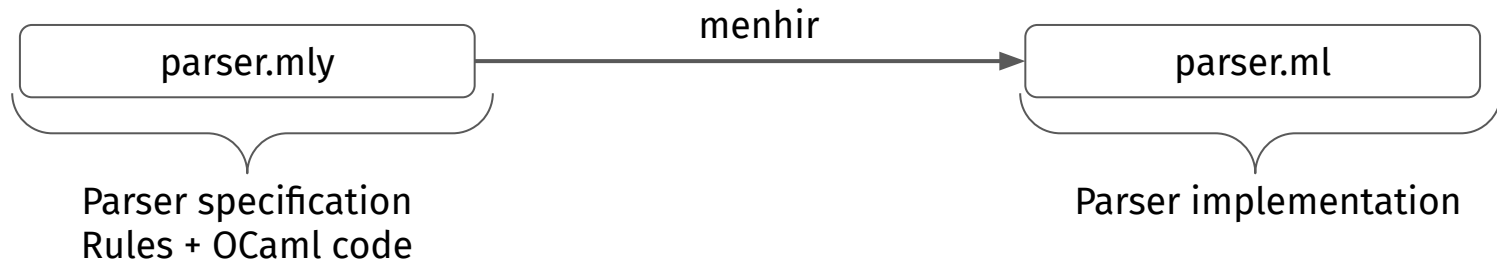


# Generated scanners: examples

Real world compilers:

- OCaml, see [parsing/parser.mly](#)
- Haskell, see [GHC/Parser.y](#)
- FSharp, see [fsharp/pars.fsy](#)

# Constructing Parsers with menhir



- You need to install it via  
`$ opam install menhir`
- The generated file is compiled and linked with the application  
`$ ocamlbuild -use-menhir app.byte`
- The parser is available through a specific function representing the start symbol of the grammar taking in input a function for scanning

# Menhir grammar specification

```
%{  
    Header          (* Ocaml code copied in the generated file: optional; *)  
%}  
  
    Declarations    (* Tokens, start symbol, associativity: mandatory *)  
  
%%  
  
    Grammar rules   (* grammar production + code *)  
  
%%  
  
    Trailer         (* Ocaml code copied in the generated file: optional; *)
```

# Token declaration

Tokens are the terminal symbols and we need to declare them

A token declaration takes the form

```
%token [<OCaml type t>] uid [qid]
```

And defines the identifier uid as terminal symbol in the grammar specification and as data constructors of the token type

If an OCaml type t is present, then uid carries a semantic value of type t

If a quoted identifier qid is present, then it can work as an alias for the terminal symbol uid throughout the grammar specification

# Start symbol declaration

A declaration of the form

```
%start [<OCaml type t>] lid
```

declares the non-terminal symbol `lid` to be the start symbol

The name `lid` becomes the name of a function that can be used to invoke the parser

The type specifies what the parser returns

# Type declaration

A declaration of the form

```
%type <OCaml type t> lid
```

specifies the non-terminal symbol `lid` has type `t`

It is mandatory to specify a type for the start symbol, optional for other non-terminals



# Priority and associativity declaration

A declaration of the form

`%noassoc uid`

`%left uid`

`%right uid`

specifies a priority level and an associativity to the symbols uid

The priority is assigned implicitly following the order of declaration  
(from lower to higher priority)

uid can be a token or a dummy symbol used to associate a priority to a rule (see later)

## Rule syntax (1)

The syntax for rules is

nonterminal :

```
id1=symbol ... idn=symbol [%prec symbol] { semantic-action: ocaml code }  
| ...  
;
```

Where

- symbol is a terminal or non-terminal
- idi are names used to access the semantic value in the ocaml code (it is also possible use the yacc notation with \$i to access the value of the i-symbol)

## Rule syntax (2)

The syntax for rules is

nonterminal :

```
id1=symbol ... idn=symbol [%prec symbol] { semantic-action: ocaml code }  
| ...  
;
```

Where

- %prec directive override the default precedence and associativity of the rule with the precedence and associativity of the given symbol
- The precedence level assigned to each production is the level assigned to the rightmost terminal symbol that appears in it

# A first example

The idea:

a small calculator that reads arithmetic expressions from the standard input, evaluates them and outputs their values on the standard output

Three files:

1. [lexer.mll](#): a ocamllex lexer that tokenizes the input
2. [parser.mly](#): the menhir parser
3. [calc.ml](#): the main program that reads a line and invokes the parser

See the [calc](#) directory in the repository

## A second example

The idea:

A version of the previous example that uses the alias mechanism for tokens

For example:

- %token PLUS "+"
- %token MINUS "-"
- %token TIMES "\*"

See the [calc-alias](#) demo on the menhir repository

## A third example

The idea:

A version of the previous example that build an abstract syntax tree for the arithmetic expression

See the [calc-ast](#)

# Source File Position

ocamllex tracks the position of tokens through associating to each token values of the type `Lexing.position`

menhir extends that mechanism by associating pair of positions both to terminal and non-terminal symbols

The position of a symbol is available inside the semantic action of a rule through a set of keywords

# Position-related keywords

Pattern	Meaning
<code>\$startpos/\$endpos</code>	start/end position of the first/last symbol in the production's right-hand side, if there is one; end position of the most recently parsed symbol, otherwise
<code>\$startpos (\$i   id)</code> <code>/\$endpos(\$i   id)</code>	start/end position of the symbol named <code>\$i</code> or <code>id</code>
<code>\$startofs/\$endofs/</code> <code>\$startofs (\$i   id)</code> <code>/\$endofs(\$i   id)</code>	Similar to above, but return an offset instead that a value of type <code>Lexing.position</code>
<code>\$loc/\$loc(\$i   id)</code>	Stands for the pair <code>(\$startpos, \$endpos)</code> or <code>(\$startpos(\$i   id), \$endpos(\$i/id))</code>



## Example 4

The idea:

An extension of the calc which builds an AST that stores also the position of each construct

See the [calc-ast-pos](#)

# Splitting grammar specifications

Grammar specifications can be split in multiple files: it is sufficient invoke menhir with multiple file names

By default a non-terminal is not visible in other file as long as the non-terminal is declared as `%public` or it is the start symbol

Note that the definition of a `%public` non-terminal can be split across multiple file, they are joined together using the choice operator (`|`)

A typical example of modularity consists in placing all token declaration in one module, so as that they can be shared by different parser working on the same set of tokens

## Example 5

An version of the calc example with one lexer and two parsers:

1. expressions in infix notation
2. expressions in postfix notation

The two parsers share a single set of tokens (see [tokens.mly](#)) and that share some productions (see [common.mly](#))

See the [calc-two](#)

# Rules with parameters (1)

A rule can be parametrized over an arbitrary number of symbols

```
%public option(X):
```

```
|      { None }
```

```
| x = X { Some x }
```

option(X) expands to either

- the empty string, producing the semantic value None, or
- to the string X, producing the semantic value Some x

## Rules with parameters (2)

Consider to have

declarations:  $\{ [] \}$

|  $ds = \text{declarations}$  option(COMMA)  $d = \text{declaration} \{ d :: ds \}$

When option is instantiated, intuitively we have

optional\_comma:

|  $\{ \text{None} \}$

|  $x = \text{COMMA} \{ \text{Some } x \}$

declarations:

|  $\{ [] \}$

|  $ds = \text{declarations}$  optional\_comma  $d = \text{declaration} \{ d :: ds \}$

# The Standard Library

Menhir provides a collection of commonly used definitions

Name	Recognize	Produces
option(X)	$X \mid \varepsilon$	A value of type $t$ option if $X$ produces $t$
pair(X,Y)	$X \ Y$	A value of type $t_1 * t_2$ if $X$ produce $t_1$ and $Y$ produces $t_2$
separated_pair(X, sep, Y)	$X \ \text{sep} \ Y$	-
list(X)	$X^*$	A value of type $t$ list, if $X$ produces $t$
nonempty_list(X)	$X^+$	-
separated_list(sep,X)	$X?(sep \ X)^*$	-
separated_nonempty_list(sep,X)	$X (sep \ X)^*$	-

## Example 6

An version of `calc` which reads multiple expressions on the same line separated by a comma: it returns a list of integers

See [calc-multi](#)

# Mehnir APIs

When Menhir processes a grammar specification `parser.mly`, it produces:

- A module `Parser` module
- A file `parser.ml` and an interface `parser.mli`

The file `parser.mli` defines the API we can use to interact with the parser:

1. Monolithic API
2. Incremental API (when invoked with the `--table` flag)



# Menhir Monolithic API

This API provides:

- The algebraic data type `token` that has a case for each token declared with the directive `%token` in the `*.mly` file
- An exception `Error` with no arguments raised when a syntactic error is detected
- The parsing function named after the start symbol of the grammar, whose type is

`(Lexing.lexbuf -> token) -> Lexing.lexbuf -> t`

where `t` is the type of values returned by the parser

## Menhir Incremental API (--table option required)

In this API (in Incremental module) the user needs to drive the parser

- The parser does not have access to a lexer, the user is responsible to feed it with tokens
- The user has access to the intermediate states of the parser and attach functionalities
- The parsing function named after the start symbol of the grammar, whose type is

position -> t MenhirInterpreter.checkpoint

# Checkpoints

A checkpoint represent an intermediate or final state of the parser

type 'a checkpoint = private

- | InputNeeded of 'a env (\* The parser needs more tokens \*)
- | Shifting of 'a env \* 'a env \* bool (\* The parser is a shift transition \*)
- | AboutToReduce of 'a env \* production (\* The parser is a reduce step \*)
- | HandlingError of 'a env (\* The parser detect an error \*)
- | Accepted of 'a (\* The input is accepted: a value of type 'a is returned \*)
- | Rejected (\* The input is rejected \*)

'a env represents the current state of the stack of the automaton

# Driving the parser (1)

To drive the parser with the incremental API you have to

- provide a token supplier, a function that delivers a new token every time is invoked

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

- Call the functions
  - offer to feed the parser with a new token when in a InputNeeded checkpoint
  - resume to allow the parser to continue when in a AboutToReduce or HandlingError checkpoints

## Driving the parser (2)

To implement the monolithic API on top of the incremental:

- Call the parsing function to obtain the first checkpoint
- Repeatedly call the offer and resume functions until a final checkpoint is obtained
  - If it is `Accepted(v)`, return `v`
  - If it is `Rejected`, throws an exception
- If an error occurs during the parsing throw an exception

See the functions `lexer_lexbuf_to_supplier`, `loop` and `loop_handle` in the documentation.

## Example 7

An version of calc which uses the intermediate API

It shows two functions loop in [calc.ml](#) that drives the parser

1. The first one only uses offer and resume
2. The second one uses the library function loop\_handle

See the [calc-incremental](#)

## Example 8

A parser for Json

- [Json.ml](#): the definition of the Json abstract syntax tree (we distinguish between integers and floats)
- [parser.mly](#) & [lexer.mll](#): the parser and lexer (the parser returns a json values one by one)
- No support for unicode

To compile:

```
$ ocamlbuild -use-menhir -use-ocamlfind -package ppx_deriving.std main.byte
```

## Other features

- An interpreter mode that allows reading a sentence from the standard input, parsing it and printing on the standard output the result: this is useful for debugging a grammar

```
$ menhir --interpret --interpret-show-cst demos/calc/parser.mly
```

- A declarative approach for error reporting that builds and displays a diagnostic message based on the state of the automaton
- Detection of explanation of conflicts in the grammar: using the `--explain` flag it generates a file with the log the conflicts

See documentation for details



# Conclusions

- Examples of handwritten parsers
- Tutorial on main features of menhir
- Some examples of parsers

## References

- [Menhir homepage](#)
- [Menhir reference manual](#)
- [Parsing with OCamllex and Menhir](#)

# Examples of parsers generators in other languages

Name	Parsing Algorithm	Languages
<a href="#">Bison</a>	GLR/LARL	C/C++/Java
<a href="#">JavaCup</a>	LALR	Java
<a href="#">JavaCC</a>	LL(1)	Java
<a href="#">CocoR</a>	LL(k)	Java/C#/C++/F#/Swift/...
<a href="#">FSYacc</a>	LR/LALR	F#
<a href="#">Antlr4</a>	ALL(k)	Java/C#/C++/Python/Go/....

For further examples see

[https://en.wikipedia.org/wiki/Comparison\\_of\\_parser\\_generators](https://en.wikipedia.org/wiki/Comparison_of_parser_generators)

# Parser combinators

An approach typically used in functional languages for defining recursive descent parsers

The underlying idea is that a parser is a function that takes a string as input and returns some structure as output

A parser combinator is a high-order function that accepts some parsers as input and returns a new parser as output

# Some examples of parser combinators

Name	Languages
<a href="#">FParsec</a>	F#
<a href="#">Angstrom</a>	OCaml
<a href="#">Parjs</a>	Javascript
<a href="#">Scala Parser Combinators</a>	Scala
<a href="#">Parsec</a>	Haskell
<a href="#">Combine</a>	Rust
...	

# Parsing Expression Grammars (PEG)

It is a special kind of formal grammar, similar to CFG where the choice operator `|` is ordered: the choice operator selects the first match that succeeds ignoring the others

The packrat parsing algorithm is a sort of recursive descent parser that exploits memoization to ensure that each parsing function is invoked once at a given input position

# Some examples of PEG parser

Name	Languages
<a href="#">pappy</a>	Haskell
<a href="#">Parboiled2</a>	Scala/Java
<a href="#">Peg.js</a>	Javascript
<a href="#">PEG</a>	Go
<a href="#">PEGTL</a>	C++
...	