Menhir 的原理与使用

Ocaml 下的 Parser 生成器

李子鸣 @ PLCT Lab

August 27, 2025

写在前面

听这个的动机？

- **MonnBit 的编译器采用了 Menhir 作为 Parser 生成器。**
- PL 学术界也经常用到 Menhir，写 DSL，Lisp 方言等。

写在前面

听这个的动机？

- **MonnBit 的编译器采用了 Menhir 作为 Parser 生成器。**
- PL 学术界也经常用到 Menhir，写 DSL，Lisp 方言等。

讲这个的动机？

- 让人感受到 Menhir 的好使，拉不用 OCaml 的入坑
- **初学者读 Menhir 手册很搞**：很多设计缘于 yacc 历史流变，各种特性不同章节互相穿插。希望能有一个循序渐进的视角，讲清楚 Menhir 各种 feature 的引入动机。

写在前面

听这个的动机?

- **MonnBit 的编译器采用了 Menhir 作为 Parser 生成器。**
- PL 学术界也经常用到 Menhir，写 DSL，Lisp 方言等。

讲这个的动机?

- 让人感受到 Menhir 的好使，拉不用 OCaml 的入坑
- **初学者读 Menhir 手册很搞**：很多设计缘于 yacc 历史流变，各种特性不同章节互相穿插。希望能有一个循序渐进的视角，讲清楚 Menhir 各种 feature 的引入动机。

需要什么前置知识?

- 学过编译原理（如果学校教的不全就看龙书 Syntax Analysis 章节补课，我可以确保它覆盖所有需要的前置知识）
- 不需要你会 OCaml（这也是为什么不讲错误处理）

如何在项目里引入 Menhir

dune 会默认使用 ocamlyacc 作为构建系统，所以你需要在 dune-project 里面添加

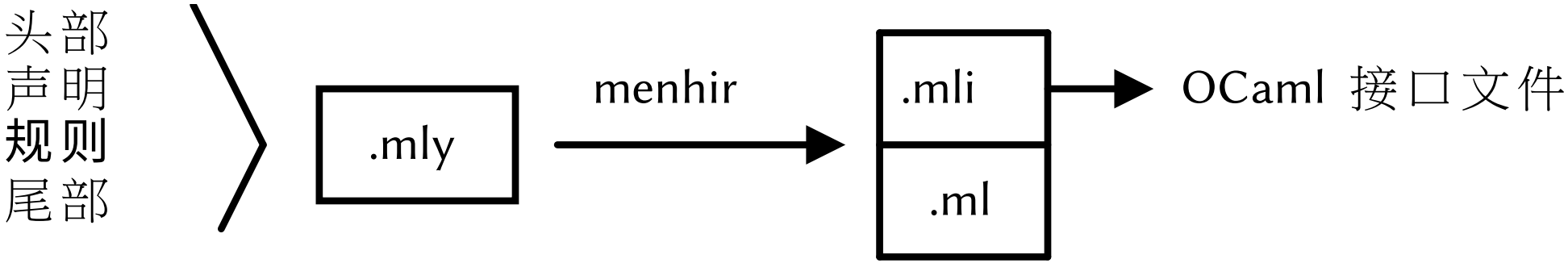
```
(using menhir 2.1)
```

或者

```
(menhir
 (modules parser))
```

前者指定了你的 menhir 高于等于这个版本号。

What is Menhir



`.mly` 中的后缀 y 意味着 yacc，menhir 兼容 yacc，所以如果你用过 Bison 之类的可能会很熟悉。

头部

头部位于 %{ 和 %} 之间。这里的代码会原样复制到生成的.ml 文件中。 通常我们会在头部写一些 open，比如：

```
%{
open Ast
%}
```

稍后在文法定义中，我们可以写 Int i 这样的表达式，而不是 Ast.Int i。

如果需要，我们也可以在头部定义一些 OCaml 函数。

声明

声明部分干三件事：1.定义 tokens、2.运算符优先级和结合性，3. 解析起始点。

声明部分和规则部分必须用 %% 隔开，这说明声明语句和规则语句是不能混着来的。

从 CFG 的角度来讲，$\begin{cases} \%\text{token} \rightarrow \textbf{终结符} \\ \%\text{start} \rightarrow \text{开始}\textbf{变}\text{量} \end{cases}$

```
%token   <int>    INT
%token   TIMES
%token   PLUS
%token   LPAREN
%token   RPAREN
%token   EOF
```

→ 这些都是lexer生 成的token

```
%nonassoc    IN
%nonassoc    ELSE
%left    LEQ
%left    PLUS
%left    times
```

需要声明语言解析的起点。以下声明表示从名为 prog 的规则（在下面定义）开始。该声明还表示解析一个 prog 将返回一个类型为 Ast.expr 的 OCaml 值。

```
%start <Ast.expr> prog
```

声明：类型

```
expr:
  | INT { $1 }
  | STRING { $1 }
  | expr; PLUS; expr { $1 + $3 }
```

expr 可能是个 int 也可能是个 string

声明：类型

```
expr:
  | INT { $1 }
  | STRING { $1 }
  | expr; PLUS; expr { $1 + $3 }
```

expr 可能是个 int 也可能是个 string

这里存在一个 string+int 的类型错误风险

声明：类型

```
expr:
  | INT { $1 }
  | STRING { $1 }
  | expr; PLUS; expr { $1 + $3 }
```

expr 可能是个 int 也可能是个 string

这里存在一个 string+int 的类型错误风险

Menhir 自身无法察觉这个错误。你会在 OCaml 编译阶段通过类型推断得到一个来自生成文件 parser.ml 的错误：

```
File "parser.ml", line 123, char 45-50:
Error: This expression has type string but an expression was expected of type int
```

声明：类型

```
expr:
  | INT { $1 }
  | STRING { $1 }
  | expr; PLUS; expr { $1 + $3 }
```

expr 可能是个 int 也可能是个 string

这里存在一个 string+int 的类型错误风险

Menhir 自身无法察觉这个错误。你会在 OCaml 编译阶段通过类型推断得到一个来自生成文件 parser.ml 的错误：

```
File "parser.ml", line 123, char 45-50:
Error: This expression has type string but an expression was expected of type int
```

自动生成的.ml 可读性差，在上面排错很烦。

声明：类型

```
expr:
  | INT { $1 }
  | STRING { $1 }
  | expr; PLUS; expr { $1 + $3 }
```

expr 可能是个 int 也可能是个 string

这里存在一个 string+int 的类型错误风险

Menhir 自身无法察觉这个错误。你会在 OCaml 编译阶段通过类型推断得到一个来自生成文件 parser.ml 的错误：

```
File "parser.ml", line 123, char 45-50:
Error: This expression has type string but an expression was expected of type int
```

自动生成的.ml 可读性差，在上面排错很烦。 要是类型错误来自.mly 该多好，直接在上面排错！

不难想到：**让.mly 类型化！从.mly 转化成.ml 时就进行类型检查。**

声明：类型

```
expr:
  | INT { $1 }
  | STRING { $1 }
  | expr; PLUS; expr { $1 + $3 }
```

expr 可能是个 int 也可能是个 string

这里存在一个 string+int 的类型错误风险

Menhir 自身无法察觉这个错误。你会在 OCaml 编译阶段通过类型推断得到一个来自生成文件 parser.ml 的错误：

```
File "parser.ml", line 123, char 45-50:
Error: This expression has type string but an expression was expected of type int
```

自动生成的.ml 可读性差，在上面排错很烦。 要是类型错误来自.mly 该多好，直接在上面排错！

不难想到：**让.mly 类型化！从.mly 转化成.ml 时就进行类型检查。**

声明：类型

- **对于终结符**，我们通过在 token 前面用尖括号包住 Ocaml 类型加上类型声明。

```
%token <Ocaml type> lid
```

- **对于非终结符**，我们额外开行写 %type 声明：

```
%type <OCaml type> lid1 ... lidn
```

为 `lid1,..., lidn` 中的每一个都指定了一个 OCaml 类型。

规则：旧语法

声明帮我们解决了 CFG 的终结符和开始变量，那么规则肯定处理的则是非终结符。

```
rulename:
  | expr SEMICOLON  { $1 }
  | production2 { action2 }
  | ...

rulename2:
  |
...
%%
```

规则：旧语法

声明帮我们解决了 CFG 的终结符和开始变量，那么规则肯定处理的则是非终结符。

```
rulename:
  | expr SEMICOLON  { $1 }
  | production2 { action2 }
  | ...

rulename2:
  |
...
%%
```

被|分割开的是一个 production group，由一个 production 和包在花括号里的 action 构成。production 是我们语法所匹配的模式。

规则：旧语法

声明帮我们解决了 CFG 的终结符和开始变量，那么规则肯定处理的则是非终结符。

```
rulename:
  | expr SEMICOLON  { $1 }
  | production2 { action2 }
  | ...

rulename2:
  |
...
%%
```

action 是如果匹配成功，返回的 OCaml 值。 action 里面的 $1 是第一个 token 的值的意思，同理，我们还可以使用 $2, $3。

被|分割开的是一个 production group，由一个 production 和包在花括号里的 action 构成。production 是我们语法所匹配的模式。

把 production group 的 action 扔掉，留下的就是 production，由一系列 producer 构成，producer 在这里你就简单的理解为终结符和非终结符，下一张幻灯片会修正你的理解。

规则：值的绑定

用 $1 和 $2 是 yacc 的传统方式，**可读性差**:

```
expr: INT PLUS INT { $1 + $3 }
```

规则：值的绑定

用 $1 和 $2 是 yacc 的传统方式，**可读性差**:

```
expr: INT PLUS INT { $1 + $3 }
```

Menhir 提供了类似 Bison 中命名引用的手段，就是值绑定。我们在 INT 前面加一个 i1，这样就把此处 INT 接收到的值绑定到了 i1 上，在 action 里你直接调用 i1 就可以了。

```
expr: i1=INT PLUS i2=INT { i1 + i2 }
```

Menhir 的手册里把终结符和非终结符都称作 actual。现在我们知道，producer 并不一定都是 actual，也可以是 actual 前跟一个值绑定。

规则：值的绑定

用 $1 和 $2 是 yacc 的传统方式，**可读性差**:

```
expr: INT PLUS INT { $1 + $3 }
```

Menhir 提供了类似 Bison 中命名引用的手段，就是值绑定。我们在 INT 前面加一个 i1，这样就把此处 INT 接收到的值绑定到了 i1 上，在 action 里你直接调用 i1 就可以了。

```
expr: i1=INT PLUS i2=INT { i1 + i2 }
```

Menhir 的手册里把终结符和非终结符都称作 actual。现在我们知道，producer 并不一定都是 actual，也可以是 actual 前跟一个值绑定。

在 Menhir 的旧语法里，每一个 producer 后面可跟分号也可不跟。也就是说，这样的语法和上面等效：

```
expr: i1=INT; PLUS; i2=INT { i1 + i2 }
```

规则：参数化语法

```
int_list:
    int
  | int_list ',' int

string_list:
    string
  | string_list ',' string
```

规则：参数化语法

```
int_list:
    int
  | int_list ',' int

string_list:
    string
  | string_list ',' string
```

代码重用，维护负担大

规则：参数化语法

```
int_list:
    int
  | int_list ',' int

string_list:
    string
  | string_list ',' string
```

代码重用，维护负担大

我用过 Bison，我晓得，用 #define 解决！

规则：参数化语法

```
int_list:
    int
  | int_list ',' int

string_list:
    string
  | string_list ',' string
```

代码重用，维护负担大

我用过 Bison，我晓得，用 #define 解决！

Wait!

- 类型不安全
- 大量隐式逻辑，增加调试难度，错误信息不好理解
- 宏的语法怪，可读性差，也不是特别好写

规则：参数化语法

```
int_list:
    int
  | int_list ',' int

string_list:
    string
  | string_list ',' string
```

代码重用，维护负担大

我用过 Bison，我晓得，用 #define 解决！

Wait!

- 类型不安全
- 大量隐式逻辑，增加调试难度，错误信息不好理解
- 宏的语法怪，可读性差，也不是特别好写

解决方案：**融入函数/泛型的思路**。

```
list(item): item
          | list(item) ',' item

int_list: list(INT)
string_list: list(STRING)
```

规则：参数化语法

```
int_list:
    int
  | int_list ',' int

string_list:
    string
  | string_list ',' string
```

代码重用，维护负担大

我用过 Bison，我晓得，用 #define 解决！

Wait!

- 类型不安全
- 大量隐式逻辑，增加调试难度，错误信息不好理解
- 宏的语法怪，可读性差，也不是特别好写

解决方案：**融入函数/泛型的思路**。

```
list(item): item
          | list(item) ',' item

int_list: list(INT)
string_list: list(STRING)
```

代码一下子更简洁了。

规则：参数化语法

```
int_list:
    int
  | int_list ',' int

string_list:
    string
  | string_list ',' string
```

代码重用，维护负担大

我用过 Bison，我晓得，用 #define 解决！

Wait!

- 类型不安全
- 大量隐式逻辑，增加调试难度，错误信息不好理解
- 宏的语法怪，可读性差，也不是特别好写

解决方案：**融入函数/泛型的思路**。

```
list(item): item
          | list(item) ',' item

int_list: list(INT)
string_list: list(STRING)
```

代码一下子更简洁了。

而且这是类型安全的! 在 Menhir 中，一个 %type 声明不仅可以针对单个非终结符，也可以针对一个完全应用的参数化非终结符，比如 `list(expression)` 。

规则：参数化语法

```
int_list:
    int
  | int_list ',' int

string_list:
    string
  | string_list ',' string
```

代码重用，维护负担大

我用过 Bison，我晓得，用 #define 解决！

Wait!

- 类型不安全
- 大量隐式逻辑，增加调试难度，错误信息不好理解
- 宏的语法怪，可读性差，也不是特别好写

解决方案：**融入函数/泛型的思路**。

```
list(item): item
          | list(item) ',' item

int_list: list(INT)
string_list: list(STRING)
```

代码一下子更简洁了。

而且这是类型安全的! 在 Menhir 中，一个 %type 声明不仅可以针对单个非终结符，也可以针对一个完全应用的参数化非终结符，比如 `list(expression)` 。

- 参数化语法是 Menhir 区别于普通 Parser 生成器的首要核心优势。😎

规则：inline

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

这种情况下，当一个类似 1+2*3 的移入/归约冲突发生后，parser 会想着通过优先级来解决。

规则：inline

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

这种情况下，当一个类似 1+2*3 的移入/归约冲突发生后，parser 会想着通过优先级来解决。

但是，parser 只会去找 op 的优先级，它不会去找 op 具体是什么的优先级。**op 相当于添加了一个抽象层，屏蔽掉了内部的信息。**

规则：inline

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

这种情况下，当一个类似 1+2*3 的移入/归约冲突发生后，parser 会想着通过优先级来解决。

但是，parser 只会去找 op 的优先级，它不会去找 op 具体是什么的优先级。**op 相当于添加了一个抽象层，屏蔽掉了内部的信息。**

手动解决的办法是放弃 op 这个非终结符，直接把它包含的 PLUS 和 TIMES 规则写到 expr 规则里。

```
expr:
  e = expr; PLUS; f = expr { e + f }
| e = expr; TIMES; f = expr { e * f }
```

规则：inline

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

这种情况下，当一个类似 1+2*3 的移入/归约冲突发生后，parser 会想着通过优先级来解决。

但是，parser 只会去找 op 的优先级，它不会去找 op 具体是什么的优先级。**op 相当于添加了一个抽象层，屏蔽掉了内部的信息。**

手动解决的办法是放弃 op 这个非终结符，直接把它包含的 PLUS 和 TIMES 规则写到 expr 规则里。

```
expr:
  e = expr; PLUS; f = expr { e + f }
| e = expr; TIMES; f = expr { e * f }
```

我们既想要优先级，又想要抽象成 op 带来的可读性，OCaml 给我们了 inline（内联）作为解决方案。

```
%inline op:
| PLUS { ( + ) }
| TIMES { ( * ) }

expr:
| e = expr; o = op; f = expr { o e f }
```

inline 的作用就是自动完成前面提到的的手动转换过程。当你给 op 规则加上 %inline 关键字时，Menhir 会在生成 parser 前，把所有对 op 的引用替换成它自己的定义。最终生成的 parser 代码就等同于手动解决后的文法。

规则：新语法

在其最简单的形式中，一条规则为

```
let lid := expression
```

其左侧 lid 是一个非终结符；其右侧是一个表达式。这种规则定义了一个普通的非终结符，

另一种形式

```
let lid == expression
```

定义了一个 %inline 非终结符。

- 和 OCaml 风格更统一
- 将 inline 和普通规则归类，提高可读性

新语法中，每一个 producer 之间必须有分号。

规则：新语法

在其最简单的形式中，一条规则为

```
let lid := expression
```

其左侧 lid 是一个非终结符；其右侧是一个表达式。这种规则定义了一个普通的非终结符，

另一种形式

```
let lid == expression
```

定义了一个 %inline 非终结符。

- 和 OCaml 风格更统一
- 将 inline 和普通规则归类，提高可读性

新语法中，每一个 producer 之间必须有分号。

新语法还提供了一些很方便的糖：

- 尖括号语法
  - `< id >`：尖括号里面是一个函数或者构造子，产生的动作是把所有前面绑定了的值传入其中。
  - `<>`：等价于`<identity>`，自动返回一个包含所有绑定变量的元组。
- 双关语法糖：`~ = id1` 是一种更简洁的写法，等效于 `id1 = id1`，表示将值绑回一个同名变量。

假设你想把一个解析后的列表反转。你可以写成：

```
my_list := elements = ID+ ; < List.rev >
```

这会自动调用 List.rev 函数，并传入 elements 变量。

规则：新语法

在其最简单的形式中，一条规则为

```
let lid := expression
```

其左侧 lid 是一个非终结符；其右侧是一个表达式。这种规则定义了一个普通的非终结符，

另一种形式

```
let lid == expression
```

定义了一个 %inline 非终结符。

- 和 OCaml 风格更统一
- 将 inline 和普通规则归类，提高可读性

新语法中，每一个 producer 之间必须有分号。

新语法还提供了一些很方便的糖：

- 尖括号语法
  - `< id >`：尖括号里面是一个函数或者构造子，产生的动作是把所有前面绑定了的值传入其中。
  - `<>`：等价于`<identity>`，自动返回一个包含所有绑定变量的元组。
- 双关语法糖：`~ = id1` 是一种更简洁的写法，等效于 `id1 = id1`，表示将值绑回一个同名变量。

假设你想把一个解析后的列表反转。你可以写成：

```
my_list := elements = ID+ ; < List.rev >
```

这会自动调用 List.rev 函数，并传入 elements 变量。

规则：标准库

1. `option(X)`，或 `X?`
   用来识别一个可选的语法元素。
   如果匹配到 X，返回 Some x（x 是 X 的语义值）。
   如果匹配空字符串，返回 None。
   e.g.：`COLON; INT?` 识别一个可选的整数类型注释，如：`int` 或 Nothing。

2. `nonempty_list(X)` 或 `X+`
   用来识别由一个或多个符号 X 组成的非空序列。
   匹配一个或多个连续的 X 符号，返回一个由所有 X 语义值组成的列表。
   e.g.：`ID+` 识别一个或多个连续的标识符，如 `foo bar baz`。

3. `list(X)` 或 `X*`
   识别一个由零个或多个符号 X 组成的序列。
   匹配零个或多个连续的 X 符号。返回一个由所有 X 语义值组成的列表。
   e.g.：ID* 识别一个或多个标识符，或者 Nothing。

4. `separated_nonempty_list(sep, X)`
   识别由一个或多个 X 组成的序列，用 sep 分隔。
   识别模式为 `X sep X sep X ...`，返回一个由所有 X 语义值组成的列表。
   e.g.：separated_nonempty_list(COMMA, expression) 识别如 1, 2, 3 这样的表达式列表。

5. `delimited(opening, X, closing)`
   用于识别被 opening 和 closing 符号包围的 X。
   识别模式为 `opening X closing`，返回 X 语义值。
   e.g.: `delimited(LPAREN, expression, RPAREN)`
   识别一个被括号包围的表达式，如 `(1 + 2)`。

更多：https://gitlab.inria.fr/fpottier/menhir/blob/master/src/standard.mly

规则：多文件

你可能会想要把规则定义在多个不同的文件里，方便管理，组织起来也更层级。

如果你在 `fileA.mly` 中定义了一个非终结符 N，它默认是私有的，不能在 `file_B.mly` 中被引用。如果两个文件意外地定义了同名的私有非终结符，Menhir 会自动重命名它们以避免冲突。

如果你想让一个非终结符 N 能在其他模块中被引用，你必须在它的定义前加上 %public 关键字。

规则：多文件

你可能会想要把规则定义在多个不同的文件里，方便管理，组织起来也更层级。

如果你在 `fileA.mly` 中定义了一个非终结符 N，它默认是私有的，不能在 `file_B.mly` 中被引用。如果两个文件意外地定义了同名的私有非终结符，Menhir 会自动重命名它们以避免冲突。

如果你想让一个非终结符 N 能在其他模块中被引用，你必须在它的定义前加上 %public 关键字。

一个公有非终结符的定义甚至可以跨越多个文件。Menhir 会将所有同名公有非终结符的定义用 | 操作符连接起来，形成一个完整的规则。例如，你可以将 `expression` 规则的不同分支分散到多个模块中。

fileA.mly

```
expression :  expression  PLUS  expression
```

fileB.mly

```
expression :  expression  TIMES  expression
```

每个文件在被 Menhir 合并处理时，都需要保证其局部上下文是完整的。这意味着，一个文件里用到的所有终结符（tokens），都必须在它自己的声明区（即 %% 之前）定义好。

尾部

尾部是可选的，你需要尾部的话就在规则区后面加上 %%，然后后面就是尾部。

尾部的内容和头部一样，都是直接复制进 `.ml` 文件的代码。

本幻灯片由 Typst 制作，通常在您得到此 pdf 的地方也随之包含其源代码，如果没有，电邮至 i@dzming.li