Ce fichier vous aide à affronter les erreurs à la compilation en lien avec les analyses lexicale et syntaxique.

Erreurs dans lib/lexer.mll

- trompez-vous dans le nom d'un lexème et recompilez pour voir ce que ça fait
- effacez la ligne définissant le lexème PLUS (par exemple), recompilez, et lancez fouine, saisissez 3+3 : c'est à ce moment qu'on récupère une erreur un peu surprenante, exception Failure("lexing: empty token")

Erreurs dans lib/parser.mly

Si un message indiquant la présence de"shift/reduce conflict" ou "reduce/reduce conflict" s'affiche lors de la compilation, il faut réparer (même si vous avez une fouine qui veut bien s'exécuter). Il s'agit de comprendre ce qu'est un conflit dans la création de l'automate.

Pour cela, on commente ci-dessous un exemple. On vous encourage également à lire

https://cambium.inria.fr/~fpottier/menhir/manual.html#sec39 (au moins le début de la partie 6). Le paragraphe "%prec annotations" dans la partie 4.2.1 du manuel- de menhir est aussi utile.

technique numéro 1 : regarder les conflits

- effacez la ligne "%left TIMES", et recompilez.
- il y a des conflits : il faut les éliminer, que ce soient des conflits shift/reduce ou reduce/reduce
- cela signifie que l'outil menhir a trouvé des erreurs dans votre grammaire.

Pour analyser la situation, il faut regarder le fichier `parser.conflicts`. Ce fichier se trouve dans `_build/default/lib/`

Dans ce ficher, on trouve en particuler quelque chose qui ressemble à :

```
** Conflict (shift/reduce) in state 8.
** Token involved: TIMES
** This state is reached from main after reading:

expression PLUS expression

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the derivations begin

main
expression EOL
(?)

** In state 8, looking ahead at TIMES, reducing production
** expression -> expression PLUS expression
** is permitted because of the following sub-derivation:

expression TIMES expression // lookahead token appears
expression PLUS expression .
```

```
** In state 8, looking ahead at TIMES, shifting is permitted
** because of the following sub-derivation:

expression PLUS expression
```

Avoir omis de préciser l'associativité de TIMES fait que dans certaines situations, l'automate ne sait pas s'il doit empiler TIMES (shift) ou reconnaître l'expression qui précède TIMES. C'est le cas ci-dessus. Dans l'état 8, l'automate a vu passer

```
expression PLUS expression
```

et il rencontre TIMES. Est-ce que le TIMES va avec le second argument du PLUS (shift) ? Ou bien expression PLUS expression est le premier argument de TIMES (reduce) ?

Dans le parser originel, on a donné à TIMES une priorité plus grande que PLUS, ce qui a pour effet de faire gagner le reduce : c'est ce à quoi on s'attend.

Notons que si on écrit 4-2+5, on reconnaît (4-2)+5 : en effet, MINUS et PLUS sont associatifs à gauche, et ont la même priorité.

Les autres conflits que l'on trouve dans parser.conflicts sont des variantes du conflit pour l'état 8.

Vous pouvez aussi mettre les mains dans le cambouis (mais la plupart du temps ça ne devrait pas être nécessaire). Il y a deux techniques pour cela.

technique numéro 2 lancer

```
menhir --dump lib/parser.mly
```

[NB : pour recompiler avec dune, il faudra d'abord supprimer les fichiers lib/parser.ml et lib/parser.mli (mais pas parser.mly!!) qui ont été engendrés par cet appel à menhir]

Vous verrez qu'un fichier `lib/parser.automaton` a été engendré. C'est la description de l'automate. On y trouve notre conflit pour l'état 8 :

```
State 8:
## Known stack suffix:
## expression PLUS expression
## LR(1) items:
expression -> expression . PLUS expression [ TIMES RPAREN PLUS MINUS EOL ]
expression -> expression PLUS expression . [ TIMES RPAREN PLUS MINUS EOL ]
expression -> expression . TIMES expression [ TIMES RPAREN PLUS MINUS EOL ]
expression -> expression . MINUS expression [ TIMES RPAREN PLUS MINUS EOL ]
## Transitions:
-- On TIMES shift to state 5
## Reductions:
-- On TIMES RPAREN PLUS MINUS EOL
--   reduce production expression -> expression PLUS expression
** Conflict on TIMES
```

L'état 8 de l'automate signale une ambiguité: il a le choix entre d'une part faire "shift" lorsqu'il lit TIMES, puis passer à l'état 5, et d'autre part faire "reduce" pour reconnaitre qu'il peut appliquer la règle `expression -> expression PLUS expression`. Par defaut, on lit qu'il fait shift, ce qui le conduit à l'état 5, ou l'on peut voir que le TIMES à été empilé (le point s'est "déplacé sur la droite") :

```
State 5
    expression : expression TIMES . expression [ TIMES RPAREN PLUS MINUS EOL ]
```

Essayez de vous convaincre du fait que ce choix par defaut est le bon, meme s'il ne faut pas se contenter de cela : il ne faut pas qu'il reste de conflits (que ce soient des conflits shift/reduce, ou reduce/reduce) lors de la compilation.

Il est **très très preferable** de developper le parser progressivement, en compilant etape par etape, plutot que de mettre toutes les regles et se retrouver avec 19 conflits shift/reduce et 27 conflits reduce/reduce. .

Les priorites des operateurs (et des "operateurs factices", que l'on peut introduire juste dans `parser.mly`, pour spécifier la priorité d'une règle de grammaire particulière) peuvent etre utiles pour eliminer les conflits. Parfois, il est judicieux de modifier la grammaire, pour la rendre moins ambigue.

Technique numéro 3: regarder l'automate qui s'exécute

Pour afficher des transitions de l'automate, voici comment proceder :

- dans le fichier `lib/dune`, on remplace la ligne `(flags --dump)` par `(flags --dump --trace)`
- on recompile `dune build`
- on execute le programme et on saisit l'expression `2+3*5` : on voit alors s'afficher toutes les transitions de l'automate

```
State 0:
2+3*5
Lookahead token is now INT (0-1)
Shifting (INT) to state 3
State 3:
Lookahead token is now PLUS (1-2)
Reducing production expression -> INT
State 14:
Shifting (PLUS) to state 8
State 8:
Lookahead token is now INT (2-3)
Shifting (INT) to state 3
State 3:
Lookahead token is now TIMES (3-4)
Reducing production expression -> INT
State 9:
Shifting (TIMES) to state 5
State 5:
Lookahead token is now INT (4-5)
```

```
Shifting (INT) to state 3
State 3:
Lookahead token is now EOL (5-6)
Reducing production expression -> INT
State 6:
Reducing production expression -> expression TIMES expression
State 9:
Reducing production expression -> expression PLUS expression
State 14:
Shifting (EOL) to state 15
State 15:
Reducing production main -> expression EOL
State 13:
Accepting
Add(2, Mul(3, 5))
17
```

et voici ci-dessous la meme execution, commentée.

- on note 1>3>6>4 si la pile des etats contient 1 au fond, et 4 en haut.
- l'action shift a pour effet d'empiler un etat (et d'empiler le lexeme qui est lu sur la pile des lexemes)
- l'action reduce correspond a l'application d'une regle de la forme A : B1 B2 .. Bk (on reconnait un A, apres avoir reconnu les B1.. Bk)
    - on depile k elements de la pile des etats
    - on retombe dans un etat qui nous dit "lorsqu'on reconnait un A, goto n"
    - on empile l'etat n en haut de la pile.

```
State 0:                                  \\ ça démarre dans l'état 0
2+3*5                                     \\ la saisie au clavier
Lookahead token is now INT (0-1)          \\ on récupère le lexeme `INT(i)`
Shifting (INT) to state 3                 \\ on empile `INT(i)` et on passe à l'éta
                                             c'est la ligne `On INT shift to state
                                             l'état 0 reste dans la pile des états

State 3:
  0>3
Lookahead token is now PLUS (1-2)
Reducing production expression -> INT  \\ on réduit comme écrit dans l'état 3
                                          on passe ensuite à l'état 14; pourquoi
                                            - car on a reconnun une `expression`
                                            - ce faisant on dépile l'état 3, et d
                                            - l'état 0 nous dit que lorsqu'on rev
                                              une expression, on "shift" à l'état
                                              (c'est la ligne `On expression shif

State 14:
  0>14
Shifting (PLUS) to state 8              \\ on empile et on pass à l'état 8
State 8:
  0>14>8
Lookahead token is now INT (2-3)
```

```
Shifting (INT) to state 3              \\ on empile et on passe à l'état 3
State 3:
  0>14>8>3
Lookahead token is now TIMES (3-4)
Reducing production expression -> INT  \\ on réduit et on passe à l'état 9; pourq
                                          car on a reconnu une expression, on ret
                                          pile d'état et on lit `On expression sh
                                          dans l'état 8 dans lequel on retombe ap
                                          l'état 3
State 9:
  0>14>8>9
Shifting (TIMES) to state 5            \\ on empile et on passe à l'état 5
State 5:
  0>14>8>9>5
Lookahead token is now INT (4-5)       \\ on empile et on passe à l'état 3
Shifting (INT) to state 3
State 3:
  0>14>8>9>5>3
Lookahead token is now EOL (5-6)
Reducing production expression -> INT  \\ on réduit et on va dans l'état 6
                                          (car l'état 5 dit `On expression shift
State 6:
  0>14>8>9>5>6
Reducing production expression -> expression TIMES expression \\ on réduit par la
                                                                  on passe en 9 ca
                                                                  trois fois (le n
                                                                  de la règle à 3
                                                                  on tombe alors s
                                                                  dit `On expressi
State 9:
  0>14>8>9
Reducing production expression -> expression PLUS expression \\ on réduit par la
                                                                passe en 14 car c
                                                                on tombe alors su
                                                                nous envoie sur l
State 14:
  0>14
Shifting (EOL) to state 15             \\ on empile et on passe à l'état 15
State 15:
  0>14>15
Reducing production main -> expression EOL  \\ on réduit par la règle, on dépile
                                               tombe sur l'état 0, et l'état 0 di
                                               `On main shift to state 13` lorsqu
                                               main
State 13:
  0>13
Accepting                              \\ on accepte `main` dans l'état 13
Add(2, Mul(3, 5))                      \\ la sortie du programme
17
```