CA - TD 1

Exercice 1 :

1) code source
↓ Analyse lexical (lexing)
suite de mots
↓ Analyse syntaxique (parsing)
AST
↓ Analyse sémantique
AST typé
↓ Simplification / optimisation
langage intermédiaire (byte code)
↓ Emission de code
code objet (assembleur)

front-end : Analyse lexical, Analyse syntaxique, Analyse sémantique
middle-end : Simplification / optimisation
back-end : Emission de code
génération de code : langage intermédiaire (byte code)

2) $[[0-9]^+ (+|*)]^* [0-9]^+$

3) Non, ce n'est pas possible. On ne pourrait pas vérifier qu'il y ait autant de parenthèses d'un côté et de l'autre.

Rappel : Expression régulière
N → t
N → tN

Exercice 2 :

1) Même problème que pour le nombre de parenthèses : ce n'est pas possible avec une regex.

2)
```
%%
(*      {c=1; BEGIN(c);}        → on peut écrire n'importe quel code dedans
<c>  *)     {c--; if(c==0) BEGIN(0);}
<c>  (*     {c++;}
<c>  .      {}
```
. → n'importe quoi

Exercice 3 :

1)
- lex.mll = générateur de lexer
- parse.mly = générateur de parser
- ast.ml
- eval.ml

ast.ml
```
type ast =
  | Add of ast * ast
  | Mul of ast * ast
  | Int of int
```

lex.mll
```
rule tokens = parse
  '+'      {PLUS}
| '*'      {MUL}
| [0-9]+ as i  {INT (int_of_string i)}
| eol      {EOL}
| '('      {LPAR}
| ')'      {RPAR}
```

parse.mly
```
%token EOL
%token PLUS
%token MUL
%token <int> INT
%token LPAR
%token RPAR
%type <ast> main
%type <ast> expr
```
les deux règles renvoient un ast

```
%start main     %left PLUS     %left MUL
main : expr EOL {$1};
expr :                          (premier elem ← $1, troisième elem → $3)
    expr  PLUS  expr   {Add($1,$3)}
    ($1)  ($2)  ($3)
  | expr  MUL   expr   {Mul($1,$3)}
  | INT                {Int($1)}
  | LPAR expr RPAR     {$2}
```
plus prioritaire à la fin

2) Fonctionnel : pratique pour l'ast, pattern matching

Orienté-objet : compliqué pour l'ast (hiérarchie complexe), compliqué pour le typage

3) - analyse sémantique - générateur de code (pour le moment, on n'a fait que la partie interprétation)

Exercice 4 :

```
%left PLUS MIN    %left MUL DIV
```

1) expr :

```
expr PLUS expr      { Add($1,$3) }
| expr MUL expr     { Mul($1,$3) }
| INT               { Int($1) }
| LPAR expr RPAR    { $2 }
| expr MIN expr     { ... }
| expr DIV expr     { ... }
| ID                { ... }
| ID LPAR expr RPAR { ... }
```

ID → identifiant

ID LPAR expr RPAR → appel de fonction

↳ il faudrait évidemment modifier les autres fichiers pour tout faire correspondre

2) On veut pas de expr à gauche dans notre grammaire.

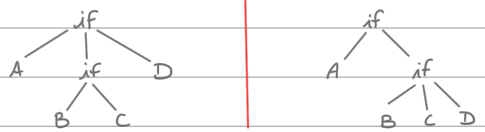
```
expr :
  INT expr1
| ID expr1
| INT
| ID
;

expr1 :
  PLUS expr
  ...
```

Exercice 5 :

1) if ::= if expr then expr [else expr] if A then if B then C else D,

2 ast possibles :



2) ifthen ::= if expr then expr

if expr then ifthenelse else expr

ifthenelse ::= (expr sans if)

e ::= ...