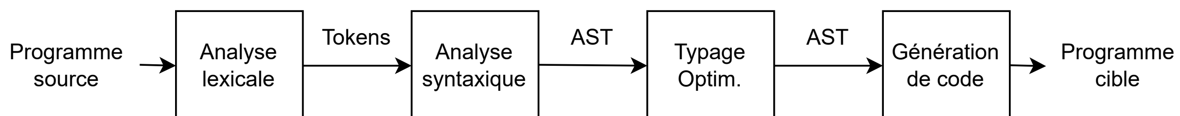
Notes de cours
Semaine 26

Cours Turing

1 L'analyse syntaxique

Cette semaine, nous allons aborder le sujet de l'analyse syntaxique. L'analyse syntaxique est le processus de décomposition d'une séquence d'entrée (exprimée en termes de *tokens*) en une structure arborescente (l'AST, pour *Abstract Syntax Tree*). C'est lors de cette étape que l'on vérifie la validité syntaxique du code source, c'est-à-dire que l'on vérifie que le code source respecte la syntaxe du langage, et que l'on construit une représentation du code source (l'AST) qui sera utilisée par les phases suivantes du compilateur.



L'analyse syntaxique est souvent appelée par sa dénomination anglaise, le *parsing*. On parle généralement de *parser* pour désigner un programme qui effectue l'analyse syntaxique d'un code source.

Le parser prend en entrée une séquence de *tokens* produite par le *lexer* (l'analyseur lexical, le sujet du dernier cours) et produit en sortie un arbre de syntaxe abstraite (AST). Cet arbre sera ensuite inspecté et manipulé par les phases suivantes du compilateur afin de détecter des erreurs (de *types*, par exemple) et de générer du code.

Généralement, l'analyse syntaxique d'un langage est basée sur une *grammaire formelle* qui définit la structure syntaxique du langage. L'outil le plus couramment utilisé pour définir cette structure syntaxique est celui des grammaires non contextuelles. Nous allons aborder ce sujet dans la suite de ce document.

2 Grammaires non contextuelles

Les grammaires non contextuelles sont un formalisme qui permet de décrire la structure syntaxique d'un langage. Le langage en question peut être un langage de programmation ou même un langage naturel (comme le français ou l'anglais), bien que les grammaires non contextuelles soient plus couramment utilisées avec les langages de programmation.

2.1 Symboles terminaux et non-terminaux

Avant de se plonger dans les détails des grammaires non contextuelles, il est important de s'intéresser à la notion de *symboles*. On distingue deux types de symboles dans une grammaire non contextuelle : les *symboles terminaux* et les *symboles non terminaux*.

Symboles terminaux

Les *symboles terminaux* sont des symboles qui ne peuvent pas être remplacés par d'autres symboles. Ils correspondent généralement aux *tokens* produits par l'analyseur lexical. Ce sont les symboles concrets du langage, comme les mots-clés, les opérateurs, les identificateurs, les nombres, etc. On note les symboles terminaux en police de type fixe, comme par exemple `+`, ou en italique, comme *number*.

Par exemple, dans un langage d'expressions arithmétiques, les symboles terminaux peuvent être les symboles suivants : `+`, `*`, `(`, `)`, et *number*.

Symboles non terminaux

Les *symboles non terminaux* sont des symboles qui peuvent être remplacés par d'autres symboles, terminaux ou non terminaux. On note les symboles non terminaux entre chevrons, comme dans $\langle \textit{expression} \rangle$.

Les symboles non terminaux représentent généralement des concepts abstraits du langage, comme les expressions, les déclarations, les instructions, etc. Ils correspondent généralement à des éléments de plus haut niveau du langage, qui sont construits à partir de combinaisons de symboles.

Par exemple, dans un langage d'expression arithmétique, un symbole non terminal pourrait être $\langle \textit{expression} \rangle$, qui représente une expression arithmétique.

Pour définir le sens des symboles non terminaux, on utilise des *règles de production* qui décrivent comment les symboles non terminaux peuvent être remplacés par d'autres symboles.

2.2 Règle de production

Les grammaires non contextuelles sont composées de *règles de production* qui définissent comment les symboles du langage peuvent être combinés pour former des phrases valides. Chaque règle de production associe un *symbole non terminal* à une séquence de symboles terminaux et/ou non terminaux. La règle indique que le symbole non terminal peut être remplacé par la séquence de symboles donnée.

Exemple

Ci-dessous est un exemple de règle de production :

$\langle \textit{expression} \rangle$ ::= $\langle \textit{expression} \rangle$ + $\langle \textit{expression} \rangle$

Cette règle de production indique que le symbole non terminal $\langle \textit{expression} \rangle$ peut être remplacé par la séquence de symboles $\langle \textit{expression} \rangle$ + $\langle \textit{expression} \rangle$.

À gauche d'une règle de production se trouve le symbole à remplacer. Il s'agit forcément d'un symbole non terminal. Les règles de production peuvent contenir, dans leur partie droite, des symboles terminaux et non terminaux. En l'occurrence, $\langle \textit{expression} \rangle$ est un symbole non terminal, tandis que + est un symbole terminal.

Epsilon

Certaines règles ont une partie droite vide. Cela signifie que le symbole non terminal peut être remplacé par une chaîne vide, c'est-à-dire qu'il peut simplement être supprimé. On note cette chaîne vide par ε, comme dans l'exemple suivant :

$\langle \textit{non-terminal} \rangle$ ::= ε

2.3 Grammaire

Une grammaire non contextuelle est un ensemble de règles de production qui définissent la syntaxe d'un langage.

Exemple

Ci-dessous est un exemple de grammaire non contextuelle qui décrit la syntaxe d'un langage d'expressions arithmétiques simples. La grammaire est composées de 4 règles de production, chacune décrivant une manière de former une expression arithmétique valide.

$\langle \textit{expression} \rangle$ ::= $\langle \textit{expression} \rangle$ + $\langle \textit{expression} \rangle$

$\langle \textit{expression} \rangle$ ::= $\langle \textit{expression} \rangle$ * $\langle \textit{expression} \rangle$

$\langle \textit{expression} \rangle$ ::= ($\langle \textit{expression} \rangle$)

$\langle \textit{expression} \rangle$ ::= *number*

Symbole de départ

Les grammaires définissent aussi un symbole de départ. Ce symbole est le symbole non terminal à partir duquel on commence les remplacements pour obtenir une séquence de symboles. Dans l'exemple ci-dessus, le symbole de départ est $\langle \textit{expression} \rangle$.

Généralement, le but de l'analyse syntaxique est de vérifier que la séquence de *tokens* produite par l'analyseur lexical respecte la grammaire donnée en déterminant quelles règles de production appliquer pour obtenir la séquence de *tokens* à partir du symbole de départ de la grammaire.

Remarque

Ces grammaires sont appelées *non contextuelles* car il n'est pas possible de définir des règles de production qui dépendent du contexte dans lequel le symbole à remplacer se trouve. La partie gauche d'une règle de production ne spécifie que le symbole à remplacer, sans tenir compte du contexte dans lequel il se trouve.

2.4 Dérivation

On dit qu'une grammaire dérive une séquence de symboles lorsque l'on arrive obtenir la séquence de symboles en appliquant successivement les règles de production de la grammaire à partir du symbole de départ de la grammaire.

Par exemple, la séquence de symboles `(1 + 2)` peut être dérivée de la grammaire donnée en exemple de la manière suivante :

```
<expression>
( <expression> )
( <expression> + <expression> )
( 1 + <expression> )
( 1 + 2 )
```

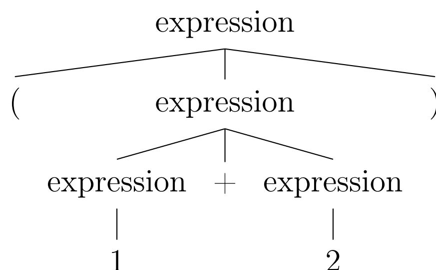
Lorsque l'on remplace toujours le symbole non-terminal le plus à gauche de la séquence, comme dans la dérivation ci-dessus, on parle de *dérivation à gauche*. Lorsqu'il existe une dérivation quelconque d'une séquence de symboles, il existe également une dérivation à gauche de cette séquence qui utilise exactement les mêmes règles de production.

2.5 Arbre de dérivation

Une dérivation peut être représentée sous forme d'arbre de dérivation, qui montre comment les symboles sont remplacés par d'autres symboles.

Exemple

L'arbre de dérivation de la séquence de symboles `(1 + 2)` à partir de la grammaire donnée en exemple plus haut est le suivant :



La structure de l'arbre de dérivation reflète la manière dont les symboles sont remplacés par d'autres symboles dans la dérivation. Elle est souvent proche de la structure de l'arbre de syntaxe abstraite qui sera construit par la suite.

À la différence de l'arbre de syntaxe abstraite, l'arbre de dérivation contient des informations sur les règles de production utilisées pour chaque nœud. L'AST, lui, est une structure plus abstraite qui ne contient que les informations nécessaires pour représenter le sens du code source.

2.6 Ambiguïté

Certaines grammaires peuvent être *ambiguës*. Dans une grammaire ambiguë, une séquence donnée de symboles peut être dérivée de plusieurs manières différentes. Généralement, cela signifie que la séquence de symboles peut être interprétée de plusieurs manières différentes, ce qui peut être problématique pour les utilisateurs du langage.

Exemple

La grammaire donnée en exemple plus haut est ambiguë. Par exemple, la séquence de symboles `1 + 2 * 3` peut être interprétée de deux manières différentes, représentées par les deux dérivations suivantes :

1. Première dérivation :

   ```
   <expression>
   <expression> + <expression>
   1 + <expression>
   1 + <expression> * <expression>
   1 + 2 * <expression>
   1 + 2 * 3
   ```

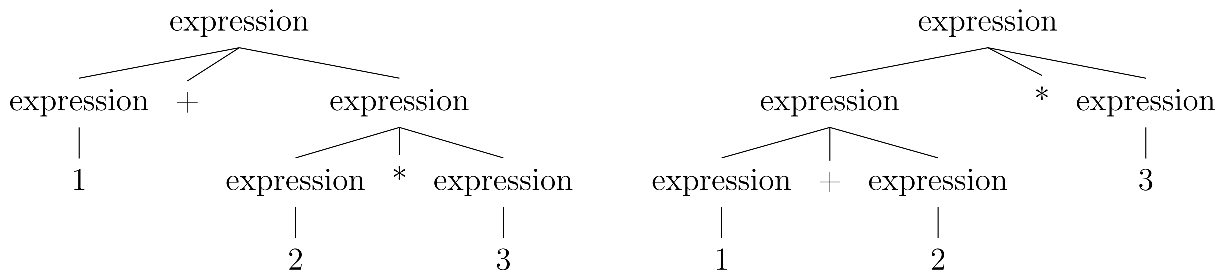
2. Seconde dérivation :

   ```
   <expression>
   <expression> * <expression>
   <expression> + <expression> * <expression>
   1 + <expression> * <expression>
   1 + 2 * <expression>
   1 + 2 * 3
   ```

Notez que les deux dérivations sont des dérivations à gauche, mais que les dérivations sont différentes. La première dérivation interprète l'expression comme une addition de deux termes, dont le second terme est une multiplication. La seconde dérivation interprète l'expression comme la multiplication de deux termes, dont le premier est une addition.

Idéalement, on souhaiterait que l'expression soit interprétée de manière unique. Pour cela, il est nécessaire de modifier la grammaire pour la rendre non ambiguë. Généralement, pour ce genre de langages, cela implique de définir des règles de priorité entre les opérateurs.

L'interprétation donnée par la grammaire dépend des règles de production appliquées. Il est possible de visualiser les dérivations possibles sous forme d'arbre de dérivation, ce qui montre la différence entre les deux interprétations.



2.7 Notation de Backus-Naur étendue

La notation de Backus-Naur étendue (EBNF, pour *Extended Backus-Naur Form*) est une extension de la notation de Backus-Naur (BNF) qui permet de décrire des grammaires de manière plus concise que la notation standard. La notation EBNF ajoute des opérateurs de répétition et de choix qui permettent de décrire de manière plus succincte des grammaires, à la manière des expressions régulières.

Ainsi, la grammaire donnée en exemple plus haut peut être réécrite en EBNF de la manière suivante :

$$
\begin{array}{lcl}
\langle \textit{expression} \rangle & ::= & \langle \textit{expression} \rangle \ ((\texttt{+} \mid \texttt{*}) \ \langle \textit{expression} \rangle)^* \\
& \mid & \texttt{(} \ \langle \textit{expression} \rangle \ \texttt{)} \\
& \mid & \textit{number}
\end{array}
$$

Dans cette version, l'opérateur `*` signifie que l'élément qui le précède peut être répété 0, 1 ou plusieurs fois. L'opérateur `|` permet de définir un choix entre deux éléments.

Notez qu'il y a deux usages différents des parenthèses dans cette grammaire : les parenthèses pour grouper des éléments (ici notées (et)), et les symboles terminaux parenthèses (ici notées `(` et `)`), qui servent à délimiter une expression entre parenthèses dans notre langage.

De manière intéressante, toute grammaire exprimée en EBNF peut être transformée en une grammaire équivalente dans la notation standard vue précédemment.

2.8 Algorithmes d'analyse syntaxique

Les grammaires sont un outil puissant pour décrire la syntaxe d'un langage, mais elles ne sont pas directement exécutables par un ordinateur. Pour cela, il est nécessaire de mettre en œuvre un algorithme d'analyse syntaxique qui va parcourir la séquence de *tokens* produite par l'analyseur lexical et vérifier que cette séquence respecte la grammaire donnée.

Certains algorithmes d'analyse syntaxique sont construits à partir des grammaires et nécessitent que celles-ci soient écrites de manière spécifique. Parmi ces algorithmes, on trouve les algorithmes CYK (Cocke-Younger-Kasami) et Earley. Ces méthodes ont une complexité temporelle au pire des cas cubique en fonction de la taille de l'entrée, ce qui peut les rendre peu efficaces pour l'analyse de code source de grande taille.

D'autres méthodes ne nécessitent pas de grammaire explicite. Dans ce genre de cas, la grammaire est tout de même utile comme outil de documentation et de spécification de la syntaxe. Parmi ces méthodes, on trouve l'analyse syntaxique par descente récursive, comme nous allons le voir dans la suite de ce document.

2.9 Grammaires LL(1)

Les grammaires LL(1) (pour *left-to-right-parse, leftmost derivation, 1 symbol lookahead*) sont un sous-ensemble des grammaires non contextuelles qui peuvent être analysées de manière simple et efficace. Dans une grammaire LL(1), le choix de la règle de production à appliquer au prochain symbole non terminal dans une dérivation à gauche peut être fait en regardant uniquement le prochain symbole dans la séquence d'entrée.

Bien entendu, toutes les grammaires ne sont pas LL(1). Cependant, la plupart des langages de programmation courants peuvent être décrits par des grammaires LL(1). Par exemple, la syntaxe formelle du langage Python était, jusqu'à récemment, décrite par une grammaire LL(1).

Les grammaires LL(1) ont l'avantage d'être non-ambiguës et d'avoir des algorithmes d'analyse syntaxique efficaces (linéaires en temps). De plus, elles se prêtent bien à une analyse par descente récursive, un algorithme d'analyse syntaxique qui est simple à mettre en œuvre et à comprendre que nous allons aborder dans quelques instants.

La grammaire donnée en exemple plus haut n'est pas LL(1) car elle est ambiguë. Il est possible de la rendre LL(1) en la modifiant légèrement, par exemple en ajoutant des règles qui tiennent compte de la priorité entre les opérateurs, comme suit en notation EBNF :

$\langle \textit{expression} \rangle$ ::= $\langle \textit{term} \rangle$ (`+` $\langle \textit{term} \rangle$)*

$\langle \textit{term} \rangle$ ::= $\langle \textit{factor} \rangle$ (`*` $\langle \textit{factor} \rangle$)*

$\langle \textit{factor} \rangle$ ::= `(` $\langle \textit{expression} \rangle$ `)`
| *number*

En notation basique, sans les opérateurs de répétition ni les alternatives, cette grammaire peut être écrite de la manière suivante :

$\langle \textit{expression} \rangle$ ::= $\langle \textit{term} \rangle$ $\langle \textit{expression_extra} \rangle$

$\langle \textit{expression_extra} \rangle$ ::= `+` $\langle \textit{term} \rangle$ $\langle \textit{expression_extra} \rangle$

$\langle \textit{expression_extra} \rangle$ ::= ε

$\langle \textit{term} \rangle$ ::= $\langle \textit{factor} \rangle$ $\langle \textit{term_extra} \rangle$

$\langle \textit{term_extra} \rangle$ ::= `*` $\langle \textit{factor} \rangle$ $\langle \textit{term_extra} \rangle$

$\langle \textit{term_extra} \rangle$ ::= ε

$\langle \textit{factor} \rangle$ ::= `(` $\langle \textit{expression} \rangle$ `)`

$\langle \textit{factor} \rangle$ ::= *number*

Notez l'usage de ε pour représenter la chaîne vide dans la partie droite de certaines règles de production. Cela signifie que le non-terminal peut être remplacé par la chaîne vide, c'est-à-dire qu'il peut simplement disparaître.

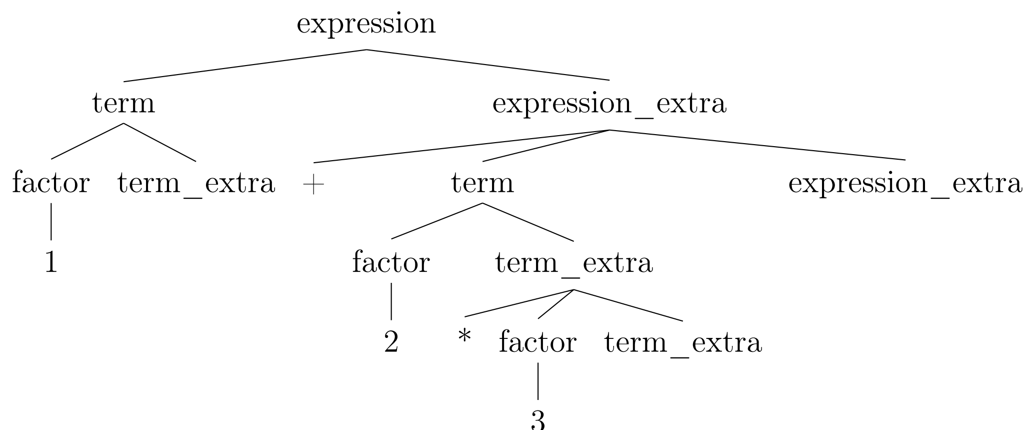
Dans la grammaire ci-dessus, plus aucune ambiguïté n'est possible. Étant donné une séquence de *tokens*, il est toujours possible de déterminer la prochaine règle de production à appliquer en regardant uniquement le symbole suivant dans la séquence de *tokens*.

Exemple

Prenons pour exemple la chaîne de tokens `1 + 2 * 3`, qui correspond à une expression valide dans cette grammaire. Regardons comment cette chaîne peut être dérivée à partir de la grammaire en notation basique donnée plus haut :

```
<expression>
<term> <expression_extra>
<factor> <term_extra> <expression_extra>
1 <term_extra> <expression_extra>
1 <expression_extra>
1 + <term> <expression_extra>
1 + <factor> <term_extra> <expression_extra>
1 + 2 <term_extra> <expression_extra>
1 + 2 * <factor> <term_extra> <expression_extra>
1 + 2 * 3 <term_extra> <expression_extra>
1 + 2 * 3 <expression_extra>
1 + 2 * 3
```

Dans l'exemple de dérivation à gauche ci-dessus, chaque application d'une règle de production est complètement déterminée par le symbole suivant dans la séquence de tokens. La dérivation de cette séquence de tokens peut être représentée sous forme d'arbre de dérivation, comme suit :



La structure en arbre de dérivation reflète la manière dont les symboles sont remplacés par d'autres symboles dans la dérivation. Elle montre que la séquence de tokens est interprétée comme une addition d'un premier terme et d'un second terme qui lui-même est une multiplication.

Remarque

Il est possible de déterminer si une grammaire est LL(1) en analysant la grammaire. Cependant, le faire dépasse le cadre de ce cours.

3 Analyse syntaxique par descente récursive

L'analyse syntaxique par descente récursive est un algorithme d'analyse syntaxique qui est souvent utilisé pour analyser des grammaires LL(1). Cet algorithme est simple à mettre en œuvre et à comprendre, et il est souvent utilisé pour analyser des langages de programmation.

L'approche d'analyse syntaxique par descente récursive repose sur une collection de fonctions mutuellement récursives. Chaque fonction correspond à un symbole non-terminal de la grammaire, et elle est responsable de vérifier que la séquence de tokens en entrée respecte bien une des règles de production associée à ce symbole non-terminal.

Si la grammaire est LL(1), alors il est possible de déterminer la règle de production à appliquer en regardant uniquement le symbole suivant dans la séquence de *tokens*.

Exemple : analyse syntaxique d'expressions arithmétiques

Pour illustrer l'analyse syntaxique par descente récursive, nous allons reprendre l'exemple de la grammaire d'expressions arithmétiques donnée plus haut, dans sa version LL(1), reprise ci-dessous en notation EBNF :

$$
\begin{aligned}
\langle \textit{expression} \rangle &::= \langle \textit{term} \rangle\ (\texttt{+}\ \langle \textit{term} \rangle)^* \\
\langle \textit{term} \rangle &::= \langle \textit{factor} \rangle\ (\texttt{*}\ \langle \textit{factor} \rangle)^* \\
\langle \textit{factor} \rangle &::= \texttt{(}\ \langle \textit{expression} \rangle\ \texttt{)} \\
&\ \ \mid\ \ \textit{number}
\end{aligned}
$$

Tokens

Nous admettrons, dans le cadre de cet exemple, que l'analyseur lexical produit les types de tokens suivants pour les symboles terminaux de la grammaire : `PlusSymbol`, `TimesSymbol`, `OpenParenthesis`, `CloseParenthesis` et `NumberLiteral`.

Définition de l'AST

Notre phase d'analyse syntaxique aura pour but de produire un arbre de syntaxe abstraite. Nous admettrons les constructeurs suivants pour l'AST : `Plus`, `Times`, et `Number`. Les constructeurs `Plus` et `Times` prennent en argument les sous-arbres correspondant aux opérandes de l'opération. Le constructeur `Number` prend en argument la valeur du nombre.

Flux de tokens

Nous admettrons que les tokens sont fournis par un objet `tokens` qui expose la méthode `consume` pour consommer le prochain token dans le flux s'il correspond à un certain type de token. La méthode `consume` retourne le token consommé s'il correspond au type donné, et `None` sinon. Pour simplifier l'implémentation, on fera en sorte que les tokens aient une valeur booléenne qui est toujours `True`, permettant ainsi d'utiliser le résultat de `consume` directement comme condition.

Implémentation

Étant donnés les éléments ci-dessus, nous pouvons écrire une fonction récursive pour chaque symbole non-terminal de la grammaire, comme suit :

```python
# Correspond au non-terminal <expression>
def parse_expression(tokens):
    """Parse an expression."""

    term = parse_term(tokens)
    while tokens.consume(PlusSymbol):
        extra_term = parse_term(tokens)
        term = Plus(term, extra_term)
    return term


# Correspond au non-terminal <term>
def parse_term(tokens):
    """Parse a term."""

    factor = parse_factor(tokens)
    while tokens.consume(TimesSymbol):
        extra_factor = parse_factor(tokens)
        factor = Times(factor, extra_factor)
    return factor


# Correspond au non-terminal <factor>
def parse_factor(tokens):
    """Parse a factor."""

    if tokens.consume(OpenParenthesis):
        expression = parse_expression(tokens)
        if not tokens.consume(CloseParenthesis):
            raise ParseError("Expected closing parenthesis")
        return expression
    elif token := tokens.consume(NumberLiteral):
        return Number(token.value)
    else:
        raise ParseError("Expected number or open parenthesis")
```

Dans cette implémentation, chaque fonction correspond à un symbole non-terminal de la grammaire. Les opérateurs de répétition de la grammaire sont gérés par des boucles `while`. Les alternatives de la grammaire sont gérées par des instructions `if`/`elif`/`else`.

Exemple

Prenons un exemple plus détaillé pour illustrer le fonctionnement de l'analyse syntaxique par descente récursive. Considérons la chaîne de tokens suivante :

`NumberLiteral(1)`, `PlusSymbol()`, `NumberLiteral(2)`, `TimesSymbol()`, `NumberLiteral(3)`

Cette chaîne correspond à l'expression arithmétique `1 + 2 * 3`. Lors d'un appel à `parse_expression` sur la séquence de tokens notée plus haut, voici comment les appels aux différentes fonctions se dérouleraient (dans la suite, l'indentation indique le niveau de profondeur de l'appel) :

- `parse_expression` appelle `parse_term` pour obtenir le premier terme de l'expression.
  - `parse_term` appelle `parse_factor` pour obtenir le premier facteur du terme.
    - `parse_factor` consomme le token `NumberLiteral(1)` et retourne un nœud `Number(1)`.
  - Comme le prochain token n'est pas le token `TimesSymbol()`, l'appel à `parse_term` termine et retourne un nœud `Number(1)`.
- `parse_expression` consomme le token `PlusSymbol()` et appelle `parse_term` pour obtenir le second terme de l'expression.
  - `parse_term` appelle `parse_factor` pour obtenir le premier facteur du terme.
    - `parse_factor` consomme le token `NumberLiteral(2)` et retourne un nœud `Number(2)`.
  - Comme le prochain token est le token `TimesSymbol()`, `parse_term` consomme ce token `TimesSymbol()` et appelle ensuite `parse_factor` pour obtenir le second facteur du terme.
    - `parse_factor` consomme le token `NumberLiteral(3)` et retourne un nœud `Number(3)`.
  - Comme le prochain token n'est pas le token `TimesSymbol()` (il n'y a pas de prochain token), l'appel à `parse_term` termine et retourne un nœud `Times(Number(2), Number(3))`.
- Pour la même raison, `parse_expression` termine et retourne l'expression :

```
Plus(Number(1), Times(Number(2), Number(3)))
```