Les opérateurs

`http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/IF/04/`

1 Retour à l'école primaire

L'instituteur explique qu'un calcul est soit une addition, soit une soustraction, etc. En termes de grammaires, il définit la grammaire suivante des expressions arithmétique.

$$S \to E \qquad\qquad E \to \mathbf{int}$$

$$E \to E\,\texttt{+}\,E \qquad E \to E\,\texttt{-}\,E \qquad E \to E * E \qquad E \to E\,\texttt{/}\,E$$

Notez que **int** est au niveau des élèves de l'école primaire qui ne connaissent que les entiers « naturels ».

1. Grâce à votre expérience des quatre opérations et des grammaires, distinguer terminaux et non-terminaux.
2. L'instituteur demande aux élèves d'écrire un calcul.

Oscar	Amélie	Samir	Pamela
1 + 2 * 3	**3 - 2 - 1**	**1 + 2 + 3**	**2 + - 1**

   (a) Montrer qu'Oscar a juste. Pour ce faire on exhibera une dérivation de S (symbole de départ) vers le mot d'Oscar. Une dérivation est tout bêtement une suite de réécritures d'un mot de terminaux et de non-terminaux, chaque réécriture consistant à remplacer un non-terminal par le membre droit d'une production de ce non-nonterminal.

   (b) Montrer que Pamela a faux. Quel est l'ensemble des mots de terminaux (le langage) généré par la grammaire ?

   (c) Compter les dérivations possibles produisant le calcul d'Oscar.

   (d) Le soir, après la classe, aidez Pamela à ratrapper son erreur. Pamela avait en fait déjà une vague idée des entiers relatifs, mais elle doit s'exprimer à l'aide de la grammaire de son instituteur.

3. Le lendemain, l'instituteur demande d'effectuer les calculs. Oscar et Samir s'y collent. Voici leurs résultats.

	1 + 2 * 3	**3 - 2 - 1**	**1 + 2 + 3**
Oscar	**9**	**0**	**6**
Samir	**7**	**2**	**6**

   L'instituteur est tenté de s'enerver, mais il se souvient de sa formation pédagogique. Comment va-t-il interpréter les résultats et comment va-t-il leur présenter la « bonne » méthode.

4. Une grammaire peut exprimer un calcul effectif, il suffit de donner un sens comme un entier aux symboles de la grammaire et de décorer les symboles des productions selon les calculs effectués. Par exemple :

$$E\ [i] \to \mathbf{int}\ [i] \qquad\qquad E\ [i_1 + i_2] \to E\ [i_1]\ \texttt{+}\ E\ [i_2]$$

   Un calcul effectif s'apparente alors à une analyse grammaticale (montrer qu'un mot appartient au langage généré par la grammaire). Plus exactement un calcul est l'inverse d'une dérivation.

   Donner deux analyses de **1 + 2 * 3** qui donnent respectivement le résultat d'Oscar et celui de Samir.

   Concevoir deux grammaires qui expriment les méthodes de calcul d'Oscar et de Samir. Ces grammaires seront d'une part équivalentes à la grammaire de départ (elles permettent d'effectuer tous le calculs syntaxiquement valides), et d'autre part non-ambigües (leurs règles n'autorisent qu'un seul résultat possible).

5. Concevoir la grammaire de l'instituteur dans le même esprit.

2 Vérification

Dans tous les programmes qui vont suivre, on considère une production de départ de la forme $S \to E$ **eof** afin de s'assurer que tout le flux d'entrée est bien analysé.

On se donne une définitions des mots (classe `Token`) et un analyseur lexical fonctionnant selon le principe du flux (classe `Lexer`). On veut écrire une classe `Parser` dans le style du lecteur des graphes de la PC précédente.

```
class Parser {
  private Lexer lexer ;
  Parser (Lexer lexer) {
    this.lexer = lexer ;
    step() ;
  }

  private Token tok ;
  private void step() {
    tok = lexer.read() ;
  }

 // Échoue en affichant « msg attendu, tok trouvé ».
  private void error (String msg, Token tok) { ... }

  void ok() { ... }
}
```

1. Écrire la méthode `ok` qui échoue qand le flux de lexème donné en entrée n'est pas une expression arithmétique valide. (On utilisera une boucle **while**).
2. Remplacer la boucle **while** du programme par des appels recursifs. Puis associer le programme transformé à l'une des grammaires de la question précédente.

3 Retour à l'École polytechnique

La définition de l'ambiguïté à l'aide de calculs arithmétiques manque quand même un peu de généralité. Pour une expression arithmétique donnée, il existe un moyen plus direct de lever l'ambiguïté : mettre des parenthèses.

Nous pouvons décrire l'effet les méthodes de calcul en disant qu'il faut d'abord traduire l'expression donnée en expression « complètement parenthésée », selon une méthode propre à chaque calculateur. Ensuite on peut calculer selon la règle universelle de d'abord effectuer les calculs les plus à l'intérieur. Par exemple, en considérant l'expression **1 * 2 + 3 * 4** on obtient :

Oscar	Samir	Instituteur
(((1 * 2) + 3) * 4)	**(1 * (2 + (3 * 4)))**	**((1 * 2) + (3 * 4))**

Le but de l'exercice est de programmer les traducteurs. On réutilise le modèle de la classe `Parser` pour écrire cette fois une classe `Translate`, munie de trois méthodes `samir`, `oscar` et `instit`.

1. Écrire les deux méthodes auxiliaires suivantes :

```
 // Lire un entier, échoue si la tête du flux n'est pas un entier
String readInt() { ...}
 // Renvoie l'expression parenthésée « ( i1 tok i2 ) »
String echo(String i1, Token tok, String i2)
```

2. Écrire d'abord la méthode `String samir()` qui parenthèse selon Samir. La technique est ici très simple, car on peut s'inspirer directement des règles de la grammaire.
3. Écrire la méthode `String oscar()`. La technique précédente ne fonctionne pas, mais on peut s'en tirer en lisant le flux comme Samir.
4. Écrire la méthode `String instit()`.
5. Comment pourrait-on effectuer les calculs,
   - Selon le principe universel,
   - plus directement,
   - selon une version plus informatique du principe universel.

4 Bonus

1. Prouver la non-ambiguïté de la grammaire de l'instituteur, en utilisant la notion d'arbre de dérivation du cours.
2. Modifier la grammaire de l'instituteur pour traiter les moins et plus unaires (productions $E \rightarrow \texttt{-}\ E$ et $E \rightarrow \texttt{+}\ E$, et les parenthèses données par l'utilisateur (production $E \rightarrow \texttt{(}E\texttt{)}$).
3. Les analyseurs que nous avons écrits sont des méthodes mutuellement récursives qui font leur choix au vu d'un seul lexème. Il sont obtenus quasiment mécaniquement à partir des grammaires.

   Trouver une grammaire non-ambigüe pour laquelle cette méthode d'analyse ne fonctionne pas.

   Indication : considérer une grammaire procédant comme Oscar ou comme Samir en fonction d'un symbole se trouvant à la fin du flux.

Solutions des exercices

Exercice 1

1. Les terminaux sont les quatre symboles des opérations plus les entiers. Les non-terminaux sont définis par les productions à savoir S et E.
2. Bon.
   (a) Il faut dériver le calcul d'Oscar à partir du symbole de départ S. Le non-termimal réduit à chaque étape est comme ça et son remplacement est comme ci.

   $$\underline{S} \to \underline{E} \to \underline{E} + E \to \mathbf{1} + \underline{E} \to \mathbf{1} + \underline{E} * E \to \mathbf{1} + \mathbf{2} * \underline{E} \to \mathbf{1} + \mathbf{2} * \mathbf{3}$$

   (b) Il faut montrer que l'on ne peut pas produire $\mathbf{1} + - \mathbf{2}$. Il suffit de remarquer que l'on ne peut pas dériver le mot vide à partir de E, et donc que l'on ne peut pas dériver, toujours à partir de E, de mots de la forme α **op** ou **op** β, où α et β sont des mots de symboles de la grammaire.

   Une autre façon de voir est de dire que les mots du language défini par la grammaire sont de la forme $\mathbf{int}\ \{\mathbf{op}\ \mathbf{int}\}^*$ (suite alternée d'entiers et d'opérateurs commençant et finissant par un entier). L'inclusion est immédiate. Réciproquement, soit α, un mot de la forme $E\{\mathbf{op}\quad E\}^*$, montrons que α dérive de E (c'est à dire que E peut se réécrire en α). On raisonne par induction :
   - Le cas de base, $\alpha = E$ est immédiat
   - Si $\alpha = E\ \mathbf{op}\ \beta$, alors par hypothèse d'induction on a $E \overset{*}{\Rightarrow} \beta$ et donc : $\alpha = E\quad \mathbf{op}\ \beta \overset{*}{\Leftarrow} E\ \mathbf{op}\ E \leftarrow E$.

   C'est terminé car tout mot de la forme $\mathbf{int}\ \{\mathbf{int}\quad \mathbf{op}\}^*$ dérive d'un mot de la forme $E\{E\quad \mathbf{op}\}^*$ (en appliquant la production $E \to \mathbf{int}$), qui lui même, par le lemme précédent dérive de E et donc de S.

   (c) On peut exprimer les choix possibles pour Oscar à l'aide d'un arbre :

   $$\begin{array}{lclcll} S \to & E\ (16) & \to & E + E\ (8) & \to \mathbf{1} + E \to & \mathbf{1} + E * E (2) \to \mathbf{1} + \mathbf{2} * E \to \mathbf{1} + \mathbf{2} * \mathbf{3} \\ & \downarrow & & \downarrow & & \downarrow \\ & E * E\ (8) & & E + E * E\ (6) & & \mathbf{1} + E * \mathbf{3} \quad \to \mathbf{1} + \mathbf{2} * \mathbf{3} \end{array}$$

   On distingue donc seize façons différentes de montrer que Oscar a bien produit un calcul (une expression arithmétique). Il en va de même pour Amélie et Samir.

   (d) Pamela peut s'en tirer grâce au zéro :

   $$\mathbf{2 + 0 - 1}$$

3. Oscar a effectué les opérations de la gauche vers la droite, tandis que Samir a effectué les opérations de la droite vers la gauche. L'instituteur dira qu'il faut effectuer les multiplications (et les divisions) avant les additions (et les soustraction). Si il a le temps, il dira aussi que si deux opérations identiques se suivent, alors il faut effectivement suivre la méthode d'Oscar. Un elève malin lui fera alors remarquer que dans le cas de deux additions par exemple, l'ordre n'a pas d'importance.
4. Deux façons de calculer $\mathbf{1} + \mathbf{2} * \mathbf{3}$. Comme Oscar.

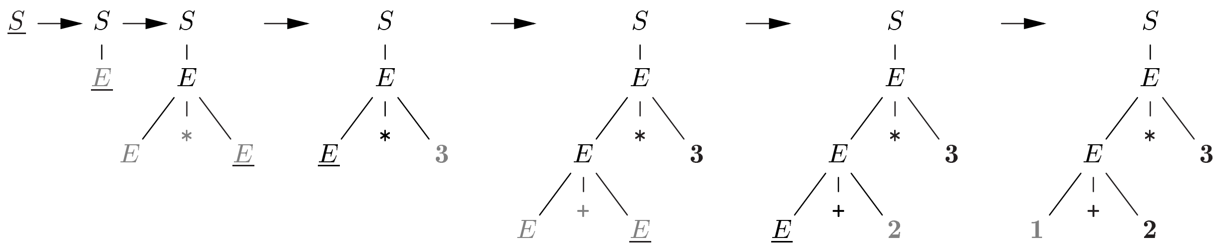
   $$\mathbf{1} + \mathbf{2} * \mathbf{3} \leftarrow \underline{E}[1] + \mathbf{2} * \mathbf{3} \leftarrow E[1] + \underline{E}[2] * \mathbf{3} \leftarrow \underline{E}[3] * \mathbf{3} \leftarrow E[3] * \underline{E}[3] \leftarrow \underline{E}[9]$$

   Comme Samir.

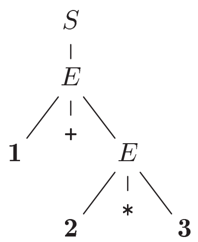
   $$\mathbf{1} + \mathbf{2} * \mathbf{3} \leftarrow \mathbf{1} + \mathbf{2} * \underline{E}[3] \leftarrow \mathbf{1} + \underline{E}[2] * E[3] \leftarrow \mathbf{1} + \underline{E}[6] \leftarrow \underline{E}[1] + E[6] \leftarrow \underline{E}[7]$$

Sur cet exemple on voit assez bien qu'il n'y a au fond que deux façons de comprendre l'expression arithmétique : calculer d'abord l'addition (comme Oscar) ou d'abord la multiplication (comme Samir). Les autres choix ne sont pas significatifs, on peut par exemple inverser les deux premières étapes de l'analyse, sans que celle-ci en soit essentiellement changée.

Les choses sont sans doute un peu plus claires, si on ne réécrit plus des mots lors des dérivations mais des arbres. C'est alors uniquement l'arbre final qui caractérise la dérivation. Ici ça nous donnera, pour la dérivation Oscar.



Tandis que l'arbre final de la dérivation Samir est le suivant.



Faisons deux remarques instructives.

– Sachant que l'on retrouve le mot correspondant à un arbre donné en lisant les feuilles de gauche à droite, l'ambiguïté de la grammaire est ici clairement exposée : il y a deux façons franchement différentes d'analyser le mot **1 + 2 * 3** selon la grammaire des expressions arithmétiques.
– L'arbre de dérivation est presque l'arbre « de syntaxe abstraite ». Il suffit pour le retrouver d'oublier quelques détails, comme les non-terminaux par exemple.

Deux grammaires.

Oscar		Samir	
$S \to E$	$E \to \mathbf{int}$	$S \to E$	$E \to \mathbf{int}$
$E \to E + \mathbf{int}$	$E \to E - \mathbf{int}$	$E \to \mathbf{int} + E$	$E \to \mathbf{int} - E$
$E \to E * \mathbf{int}$	$E \to E \mathbin{/} \mathbf{int}$	$E \to \mathbf{int} * E$	$E \to \mathbf{int} \mathbin{/} E$

Ces grammaires permettent bien d'effectuer tous les calculs possibles et spécifient bien un résultat unique (elle sont équivalentes à la grammaire de départ et non-ambigües). Prenons le cas d'Oscar et d'un mot $\mathbf{int}_1\ \mathbf{op}\ \mathbf{int}_2\ \mathbf{op} \ldots \mathbf{op}\ \mathbf{int}_n$. La seule façon de remonter vers S est la suivante :

$$\mathbf{int}_1\ \mathbf{op}\ \mathbf{int}_2\ \mathbf{op} \ldots \mathbf{op}\ \mathbf{int}_n \leftarrow \underline{E}\ \mathbf{op}\ \mathbf{int}_2\ \mathbf{op} \ldots \mathbf{op}\ \mathbf{int}_n \leftarrow \underline{E}\ \mathbf{op} \ldots \mathbf{op}\ \mathbf{int}_n \leftarrow \underline{E}\ \mathbf{op}\ \mathbf{int}_n \leftarrow \underline{S}$$

5. Voici la grammaire de l'instituteur :

$$S \to E$$

$$E \to P \qquad E \to E \texttt{+} P \qquad E \to E \texttt{-} P$$

$$P \to F \qquad P \to P * F \qquad P \to P \,/\, F$$

$$F \to \mathbf{int}$$

   On peut la paraphraser en disant
   (a) Une expression est une somme de produits de facteurs.
   (b) Les sommes (et les produits) « penchent à gauche ».

   La non-ambiguïté de cette grammaire semble un peu pénible à prouver proprement (l'ordre des opérations est plus libre que dans le cas des deux élèves), mais il semble clair qu'elle s'apparente à celle de Samir, et qu'elle exprime les priorités traditionnelles des opérations.

Exercice 2

1. Le langage reconnu est **int** $\{$**op int**$\}^*$, on peut donc sans trop réfléchir produire cette méthode ok.

```
void ok() {
  if (tok.nature != Token.int) error("entier", tok) ;
  step() ;
  while (tok != Token.EOF) {
    switch (tok.nature) {
      case Token.ADD: case Token.SUB: case Token.MUL: case Token.DIV:
        step() ;
        break ;
      default:
        error("opérateur", tok) ;
    }
    if (tok.nature != Token.int) error("entier", tok) ;
    step() ;
  }
}
```

2. La boucle **while** s'exprime aisément par une récursion terminale :

```
void ok() {
  expression() ;
  if (tok.nature != Token.EOF) error("EOF", tok) ;
}

void expression() {
  if (tok.nature != Token.INT) error("entier", tok) ;
  step() ;
  switch (tok.nature) {
    case Token.ADD: case Token.SUB: case Token.MUL: case Token.DIV:
      step() ;
      return expression() ;
    default:
```

```
      return ;
    }
  }
```

On peut alors associer la production $S \to E$ **eof** à la méthode `ok` et les productions $E \to$ **int op** E | **int** à la méthode `expression`. On a fait le choix de retourner immédiatement de la méthode `expression` quand le deuxième lexème du flux n'est pas un opérateur. Cela s'explique en disant que dans ce cas, c'est nécessairement la production $E \to$ **int** qui s'applique éventuellement, l'autre ne pouvant s'appliquer. En fait, il ne faut effectivement retourner que si le lexème en question est `Token.EOF`. Dans les autres cas, il faudrait signaler une erreur. Mais comme ce lexème sera finalement vérifié (par la méthode `ok`), on peut s'abstenir.

Bref, on a essentiellement suivi la grammaire de Samir. La règle est grossièrement, une méthode par non-terminal, dont le corps est écrit par transformation des membres droits associés, en remplaçant les non-terminaux par des appels de méthode, et les terminaux par des vérification de lexèmes, ou des choix (ici il y un seul choix entre un opérateur et « autre chose »).

Exercice 3

1. Deux méthodes auxiliaires.

```
String readInt() {
  if (tok.nature != Token.INT) error("entier", tok) ;
  String r = Integer.toString(tok.asInt) ;
  step() ;
  return r ;
}

String echo(String i1, Token tok, String i2) {
  switch (tok.nature) {
  case Token.ADD: case Token.SUB:
  case Token.MUL: case Token.DIV:
    return "(" + i1 + tok + i2 + ")" ;
  }
  error("Impossible !!!", tok) ;
  return "" ;
}
```

2. Samir.

```
String samir() {
  String i1 = readInt() ;
  switch (tok.nature) {
  case Token.ADD: case Token.SUB:
  case Token.MUL: case Token.DIV:
    Token op = tok ;
    step() ;
    return echo(i1, op, samir()) ;
  case Token.EOF :
    return i1 ;
  default:
    error("opérateur", tok) ;
    return "" ; // Not Reached
  }
}
```

3. Pour Oscar on ne peut plus s'inspirer des règles, car alors on écrirait un programme ne terminant jamais, la méthode `oscar` se rappelant immédiatement elle-même.

   Le truc est de procéder à l'analyse comme pour Samir, mais de produire un résultat différent. Dans une production $E \rightarrow E$ **op int** le E du membre droit représente le préfixe déjà vu du mot en cours de reconnaissance.

```
String doOscar(String dejavu) {
  switch (tok.nature) {
  case Token.ADD: case Token.SUB:
  case Token.MUL: case Token.DIV:
    Token op = tok ;
    step() ;
    int i2 = readInt() ;
    return doOscar(echo(dejavu, op, i2))
  case Token.EOF :
    return r ;
  default:
    error("opérateur", tok) ;
    return "" ; // Not Reached
  }
}

String oscar() {
  return doOscar(readInt()) ;
}
```

   On notera que la nécessité de disposer d'un `dejavu` non-vide conduit à modifier un peu la grammaire « à la Samir ».

$$S \rightarrow \textbf{int}\ E\ \textbf{eof} \qquad\qquad E \rightarrow \textbf{op int}\ E \qquad\qquad E \rightarrow \epsilon$$

   En pratique, on peut remplacer la récursion (terminale) par une boucle **while**.

```
String oscar() {
  String r = readInt() ;
  while (tok.nature != Token.EOF) {
    switch (tok.nature) {
    case Token.ADD: case Token.SUB:
    case Token.MUL: case Token.DIV:
      Token op = tok ;
      step() ;
      String i = readInt() ;
      r = echo(r, op, i) ;
      break;
    default:
      error("opérateur", tok) ;
    }
  }
  return r ;
}
```

4. Pour l'instituteur, on suit les règles de la grammaire, en se débrouillant pour faire pencher les expressions à gauche comme pour Oscar.

```
private String parseF() {
  return readInt() ;
```

```
}

private String parseP() {
  String r = parseF() ;
  while (tok.nature == Token.MUL || tok.nature == Token.DIV) {
    Token op = tok ;
    step() ;
    String f = parseF() ;
    r = echo(r,op,f) ;
  }
  return r ;
}

private String parseE() {
  String r = parseP() ;
  while (tok.nature == Token.ADD || tok.nature == Token.SUB) {
    Token op = tok ;
    step() ;
    String p = parseP() ;
    r = echo(r,op,p) ;
  }
  return r ;
}

String instit() {
  String r = parseE() ;
  if (tok.nature != Token.EOF) error("EOF", tok) ;
  return r ;
}
```

5. Pour réaliser les calculs selon la méthode universelle il faut procéder à l'analyse syntaxique des expressions bien parenthésées revoyées par les trois methodes `samir`, etc.

   Mais la méthode `universel` utilisera une méthode `calc` qui effectue les calculs, là où les autres analyseurs employaient `echo`.

```
int calc(int i1, Token tok, int i2) {
  switch (tok.nature) {
  case Token.ADD: return i1+i2 ;
  case Token.SUB: return i1-i2 ;
  case Token.MUL: return i1*i2 ;
  case Token.DIV: return i1/i2 ;
  }
  error("Impossible !!!",  tok) ;
  return 0 ;
}
```

   Ensuite on pourra par exemple écrire :

```
String pretty = new Translate(lexer).instit() ;
int r = new Translate(new Lexer (new StringReader (pretty))).universel() ;
```

   L'écriture de la méthode `universel` est facile, les parenthèses levant absolument toutes les ambiguïtés.

```
int doUniversel() {
  if (tok.nature == Token.OPAR) {
```

```
      step() ;
      int i1 = doUniversel() ;
      Token op = null ;
      switch (tok.nature) {
      case Token.ADD: case Token.SUB:
      case Token.MUL: case Token.DIV:
        op = tok ;
        break ;
      default:
        error("opérateur", tok) ;
      }
      step() ;
      int i2 = doUniversel() ;
      if (tok.nature != Token.CPAR) error("parenthèse␣fermante", tok) ;
      step() ;
      return calc(i1, op, i2) ;
    } else if (tok.nature == Token.INT) {
      int i = tok.asInt ;
      step() ;
      return i ;
    } else {
      error("entier␣ou␣parenthèse␣ouvrante", tok) ;
      return 0 ; // Not Reached
    }
  }

  int universel () {
    int r = doUniversel() ;
    if (tok.nature != Token.EOF) error("EOF", tok) ;
    return r ;
  }
```

Note : la grammaire des expressions complètement parenthésées est très semblable à celle des arbres de l'amphi.

$$S \to E\mathbf{eof} \qquad\qquad E \to \mathbf{int} \qquad\qquad E \to \mathbf{(}E\ \mathbf{op}\ E\mathbf{)}$$

L'écriture de l'analyseur est ici particulièrement évidente, le premier non-terminal d'un mot déterminant la production à appliquer.

6. Il serait évidemment plus direct de procéder aux calculs dès la première analyse en remplaçant les appels à `echo` par des appels à `calc` dans les trois méthodes `oscar`, etc.

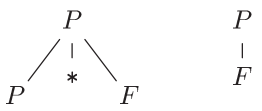
   Il serait plus informatique d'éviter une nouvelle analyse en remplaçant `echo` par une méthode construisant des abres binaires.

Exercice 4

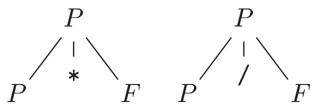
1. Il faut donc prendre un mot quelconque du langage des expressions arithmétique et montrer qu'il ne peut pas y avoir deux arbres de dérivation différents menant à α. Plus précisément on considère un mot α, tel que $\Sigma \overset{*}{\Rightarrow} \alpha$ où Σ est un non-terminal et on montre qu'il n'existe pas d'autre arbre de racine Σ et dont les feuilles lues de gauche à droite forment α.

   Le resultat est immédiat pour $\Sigma = F$, car il n'y qu'une seule production.

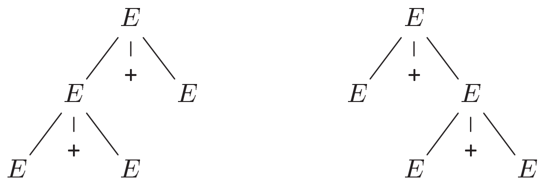
   Considérons P. Une ambiguïté éventuelle porte d'abord sur le choix d'une des trois production. Soit :



Ou encore :



Mais ces deux cas sont impossibles, car un mot dérivé de F ne saurait contenir le symbole $*$, et l'avant dernier terminal de α impose la production.
En toute complétude il faut aussi considérer le cas ou l'application d'une production peut se faire de deux manières différentes. Ici c'est impossible car F ne peut être qu'un **int** et donc l'avant dernier symbole de α determine non seulement une production unique, mais aussi une façon unique de l'utiliser.
Pour bien comprendre le problème, on peut caractériser l'ambiguïté dite d'associativité de la première gramaire ainsi :



On montre ensuite que les arbres de racine E sont pareillement déterminés ce qui entraîne la non-ambiguïté.

2. Il suffit d'introduire ces nouvelles constructions comme de nouvelles productions de F.

$$S \rightarrow E$$

$$E \rightarrow P \qquad E \rightarrow E \texttt{+} P \qquad E \rightarrow E \texttt{-} P$$

$$P \rightarrow F \qquad P \rightarrow P * F \qquad P \rightarrow P \texttt{/} F$$

$$F \rightarrow \textbf{int} \qquad F \rightarrow \texttt{+} F \qquad F \rightarrow \texttt{-} F \qquad F \rightarrow (E)$$

Les productions $F \rightarrow \texttt{+} F$ et $F \rightarrow \texttt{-} F$ expriment que l'application des opérateurs unaires est plus prioritaire que celle des opérateurs binaires. La production $F \rightarrow (E)$ s'explique par la règle « on calcule d'abord à l'intérieur des parenthèses », par ailleurs, on doit pouvoir mettre une expression arbiraires entre parenthèses. On peut encore écrire l'analyse en s'inspirant des règles et de fait il suffit de mofifier `parseF`.

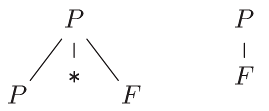
```
private String parseF() {
  switch (tok.nature) {
  case Token.SUB:
    Token op = tok ;
    step() ;
    return echo ("0", op,  parseF()) ;
  case Token.ADD:
    step() ;
```

```
      return parseF() ;
    case Token.OPAR:
      step() ;
      String r = parseE() ;
      if (tok.nature != Token.CPAR) error(")", tok) ;
      step() ;
      return r ;
    default:
      return readInt() ;
    }
  }
```

Prouvons maintenant la non-ambiguïté, qui semble bien assurée car nous utilisons quotidiennement cette grammaire sans quiproquo aucun.

Soit donc un mot α dérivé d'un non-terminal Σ. Le cas $\Sigma = F$ est facile, le premier symbole du flux déterminant le choix d'une production.

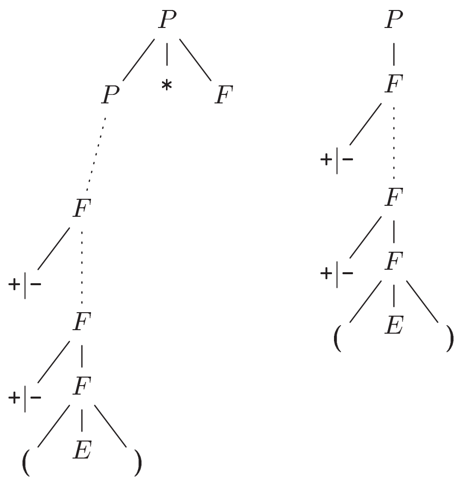
On considère ensuite seulement un example.



On ne peut pas s'en tirer comme précédemment, car cette fois un mot dérivé de F peut contenir le terminal `*`.

On se sert d'un lemme de bon-parenthésage. On peut montrer facilement que tout mot dérivé α d'un non-terminal comprend autant de parenthèses ouvrantes que fermantes. Mais plus précisément, tout préfixe de α contient plus (au sens large) d'ouvrantes que de fermantes, tandis que tout suffixe de α contient moins (toujours au sens large) d'ouvrantes que de fermantes.

En cas d'ambiguïté, donc, on est en raison de la nécessaire parenthèse fermante qui termine le mot α, et de la parenthèse ouvrante qui le commence presque, nécessairement dans le cas suivant.



Autrement dit, α se décompose nécessairement d'une part en $\beta(\eta)\gamma$ et d'autre part en $\beta(\eta')$ où β est une suite peut-être vide de `+` et de `-`, tandis que η et η' dérivent de E. Soit $\eta)$ est un préfixe de η'. Or $\eta)$ contient une fermante de plus que d'ouvrantes, contradiction.

3. Donc deux premières séries de productions.

$$O \to O \, \texttt{+} \, \textbf{int} \qquad\qquad O \to \textbf{int}$$

$$S \to \textbf{int} \, \texttt{+} \, S \qquad\qquad S \to \textbf{int}$$

Puis une dernière série qui se détermine en fonction d'un symbole situé à la fin.

$$E \to O \, \ast \qquad\qquad E \to S \, /$$

Il est clair que cette grammaire n'est pas ambigüe (la preuve est essentiellement la même que pour les grammaires d'Oscar et de Samir). Mais en écrivant la méthode correspondant à E, on ne peut connaître quelle méthode appeler avant que cette dernière aie retourné.
Ici, on peut s'en tirer en lisant de droite à gauche, mais c'est s'éloigner de principe du flux. Par ailleurs, ça ne fonctionnera pas toujours. Considérer par exemple :

$$E \to O \ast S \qquad\qquad E \to S \, / \, O$$