Lars Gabriel Annell Rydenvald

Université de Genève

Faculté des Sciences

Département d'informatique

Projet de bachelor

Ardoises

Formalismes d'opérateurs et expressions

Sommaire

1 Introduction

Ce travail repose sur Ardoises[3], qui est une plateforme de modélisation formelle. Elle permet aux utilisateurs de créer des modèles formels modulaires, d'effectuer des vérifications de propriétés sur ces modèles et offre une représentation visuelle des formalismes.

Ce travail est constitué de deux parties:

1. La création de modèles formels d'opérateurs et d'expressions. La distinction entre les modèles "opérateurs" et "expressions" est nécessaire pour assurer la modularité des modèles, qui est une fonctionnalité importante du logiciel Ardoises.
2. Pour faciliter la visualisation des modèles formels d'expressions, on propose aussi un analyseur syntaxique dont la grammaire acceptée sera basée sur le modèle d'expression donné par l'utilisateur.

2 Formalismes d'opérateurs et d'expressions

2.1 Layered data

Toute cette partie dépend de la librairie `Layeredata`[4], qui permet l'héritage prototype en Lua. `Layeredata` est aussi responsable de la vérification de propriétés sur les instances des modèles.

Avec cette librairie, on a la possibilité de créer des modèles de base, puis de les spécialiser en définissant plus précisément les propriétés des modèles qui héritent des modèles de base. De plus, grâce au formalismes `data.record` et `data.collection` de la librairie `Ardoises/Formalismes` on peut imposer des propriétés que les modèles plus spécifiques devront satisfaire.

2.2 Distinction entre opérateurs et expressions

Même si cela paraît évident, il est important de noter les différences entre opérateurs et expressions; un opérateur est constitué d'un symbole (sa représentation textuelle), accepte un certain nombre d'opérandes et possède certaines propriétés, tandis qu'une expression est constituée d'un ou plusieurs opérateurs. Sachant qu'une opérande d'un opérateur peut être une sous-expression, il nous faut donc un moyen (de préférence simple) pour exprimer les sous-expressions acceptées.

2.3 Définition des opérateurs

Comme noté plus haut, la modularité des modèles est importante. Supposons deux modèles d'expressions différents:

- Le premier accepte les opérateurs `addition` et `soustraction`.
- Et le deuxième accepte les opérateurs `addition` et `multiplication`.

Ce qu'on veut dans ce cas, c'est pouvoir utiliser le même modèle de l'opérateur `addition`, c'est à dire que l'opérateur doit être conçu pour accepter des sous-expressions en fonction du modèle d'expression qui l'utilise. C'est-à-dire, le modèle d'opérateur ne doit pas définir à l'avance les sous-expressions qu'il accepte. Le modèle de base `operator` imposera donc simplement les propriétés que les opérateurs plus spécifiques devront eux-même définir.

Ensuite, on définira les métadonnées des opérateurs plus spécifiques de la manière suivante:

```
specific_operator [meta] = {
  of       = false, -- Ceci définira plus tard le type des opérandes accepté
                    -- par l'opérateur.
                    -- La propriété est initialisée à "false" car
                    -- on ne connaît pas encore le type, qui peut être une sous-expression
                    -- ou des valeurs litérales.
```

```
  operands = {
    [refines] = { collection },
    [meta   ] = {
      [collection] = {
        minimum    = 2,
        maximum    = 2,
        value_type = re.operator [meta].of, -- Définit le type des opérandes
                                            -- Ce type sera soit une sous-expression,
                                            -- soit un litéral
      },
    },
  },
}
```

2.4 Définition des expressions

La modularité des définitions pour les expressions reste importante; le modèle d'expression le plus abstrait définit donc simplement qu'une expression est composée:

- D'une part d'opérateurs acceptés par l'expression.
- D'autre part de sous-expressions, composées par les opérateurs de l'expression, et des opérandes acceptées par ces opérateurs.

Un modèle plus concret définira ensuite les opérateurs qui composent l'expression. Finalement, on hérite de cette expression pour définir les sous-expressions. Pour donner un exemple; reprenons une expression qui autorise simplement l'addition (à noter: les litéraux (nombres, booléens et chaînes de caractère) sont aussi représentés par des opérateurs). On a donc la définition de l'expression

```
addition_operator = {
  [refines] = { addition },
  [meta   ] = { of = ref }, -- Ce champ est important!
                            -- C'est lui qui définit le type des opérandes acceptées.
                            -- De cette manière, l'opérateur peut accepter des sous-expressions.
                            -- 'ref' fait référence à l'expression qu'on définit.
}

number = {
  [refines] = { literal_number },
  [meta   ] = { of = "number" }, -- Le litéral devrait seulement accepter des opérandes
                                 -- qui sont des nombres
}

-- Finalement on compose pour la définition de notre expression.
addition_expression [meta] = {
  [expression] = {
    addition_operator = addition_operator,
    number            = number,
  }
}
```

Maintenant que l'expression est définie, on peut en faire quelque chose d'encore plus concret:

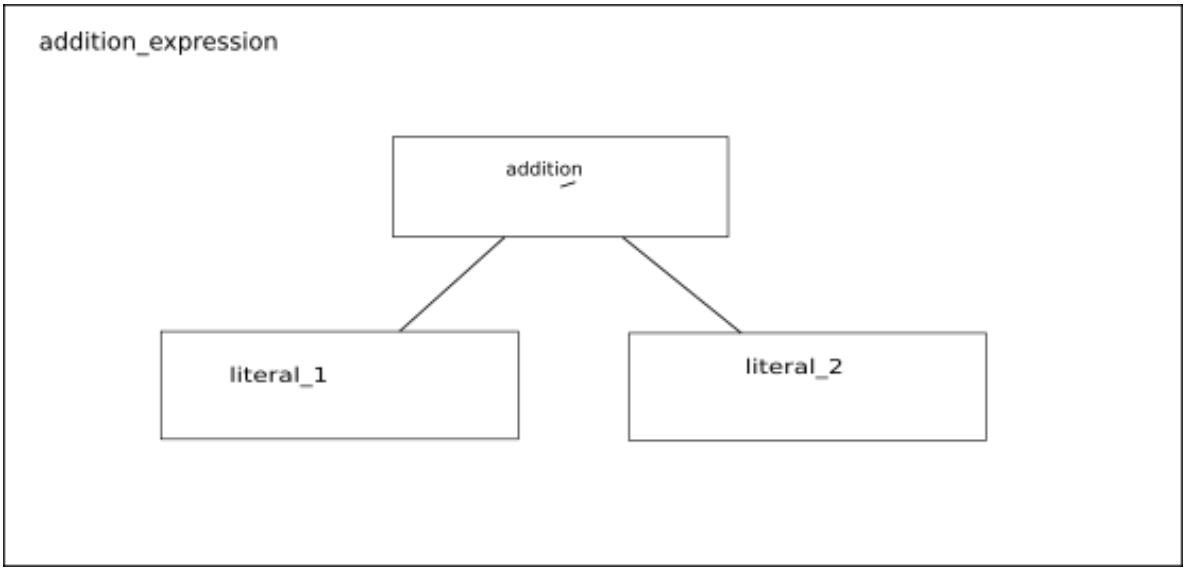
```
literal_1 = {
  [refines] = { addition_expression },
  operator  = addition_expression[meta][expression].number,
  operands  = { 1 },
}

literal_2 = {
  [refines] = { addition_expression },
  operator  = addition_expression[meta][expression].number,
  operands  = { 2 },
```

```
}

addition = {
  [refines] = { addition_expression },
  operator  = addition_expression[meta][expression].addition_operator,
  operands  = {
    literal_1,
    literal_2,
  }
}
```

Cette représentation nous donne l'arbre syntaxique suivant:



2.5 Ajouter des opérateurs à une expression

Supposons qu'on se base sur l'expression de la section précédente, et qu'on veuille rajouter un opérateur "multiplication". Il suffira de créer un modèle pour l'opérateur, créer un modèle pour notre expression, hériter du modèle `addition_expression` puis d'insérer notre opérateur multiplication dans la table `[meta][expression]`. On écrit donc

```
multiplication_addition_expression[refines] = { addition_expression }

multiplication_operator = {
  [refines] = { multiplication },
  [meta   ] = { of = ref },
}

multiplication_addition_expression[meta][expression].multiplication = multiplication_operator
```

et on a notre modèle qui accepte les additions, les multiplications et les nombres.

2.6 Supprimer des opérateurs d'une expression

La librairie `Layeredata` fournit un moyen simple de supprimer des propriétés, la clé `Layer.key.deleted`. Pour retirer un opérateur d'une expression, il suffit alors de créer un nouveau modèle, d'hériter de l'expression qu'on veut modifier, puis d'écrire, par exemple:

```
other_expression[refines] = { addition_expression }
other_expression[meta][expression].addition = Layer.key.deleted
```

De cette manière, notre modèle `other_expression` n'acceptera plus l'opérateur `addition`.

2.7 Malheureusement...

Dû à une limitation de la librairie `Layeredata`, l'approche décrite précédemment ne fonctionne pas en pratique. Le problème identifié est qu'en référençant une expression en tant que type d'opérandes pour un opérateur, on se retrouve dans une boucle infinie lorsque `Layeredata` tente de vérifier que l'instance du modèle respecte les propriétés de celui-ci.

2.7.1 Solution

Nous n'avons pas encore de solution pour ce problème, cependant Alban a décidé d'en prendre la responsabilité.

3 Analyse syntaxique

La modularité de construction des opérateurs et des expressions que nous proposons fait que nous ne connaissons pas en avance la grammaire nécessaire pour effectuer l'analyse syntaxique. Il nous est donc imposé de générer la grammaire à partir du modèle d'expression défini par l'utilisateur. Deux solutions pour effectuer cette tâche et faire la construction de l'arbre syntaxique correspondant se présentaient à nous:

1. Utiliser des expressions rationnelles, qui seraient définies par l'utilisateur pour chaque opérateur.
2. Générer une grammaire à partir de règles prédéfinis pour différents types d'opérateurs.

Le choix a été fait d'utiliser la $2^{ème}$ méthode, car même si les expressions rationnelles auraient pû être un moyen "plus simple", elles posent un problème: les expressions arithmétiques incluant des parenthèses ne sont pas des langages réguliers, puisqu'on peut, en prenant un exemple simple, se retrouver dans un cas comme celui-ci: $\{(`(`^n`)`^n)^*, n \geq 0\}$.

3.1 Choix de l'outil

Il existe plusieurs générateurs d'analyseurs lexicaux, tels que:

1. ANTLR, écrit en Java, générant un analyseur $LL(^*)$
2. Flex/Bison, écrit en C, générant un analyseur $LALR(1)$
3. PEGjs, écrit en JavaScript, générant un parser PEG
4. LPeg, écrit en C, compatible avec Lua, générant un parser PEG
5. LulPeg, implémentation en pur Lua de $LPeg$.

Puisque le projet est écrit en Lua, pour avoir une interopérabilité facile entre les définitions d'opérateurs et expressions, le choix est tombé sur $LPeg$. $LulPeg$ a aussi été considéré, cependant celui-ci est beaucoup plus lent que $LPeg$. En effet, pour produire un arbre syntaxique à partir de l'expression `((((((sum(30-20))))))`, LPeg finit en 2.233 secondes, tandis que $LulPeg$ prend 213 secondes (3 minutes et 33 secondes). Ces nombres sont des moyennes calculées sur 1'000 exécutions pour $LPeg$ et 18 exécutions pour $LulPeg$ (1'000 exécutions de $LulPeg$ aurait pris environ 50 heures), et chronométrées avec la commande `time`, sur Ubuntu 16.04 et un ordinateur équipé d'un processeur Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz.
De plus, par sa modularité (les PEG donnent la possibilité de composer des grammaires de manière procédurale), $LPeg$ permet de facilement générer une grammaire à partir des opérateurs définis par l'utilisateur.

3.2 Génération de grammaire

Supposons les opérateurs simples suivants:

Opérateur	Représentation	Priorité	Type	Associativité
Addition	'+'	11	binaire	droite
Soustraction	'-'	11	binaire	droite
Multiplication	'*'	12	binaire	droite
Division	'/'	12	binaire	droite
Exponentiation	'^'	13	binaire	droite
Négation	'-'	14	unaire préfixe	N/A
nombre	[0-9]+	15	litéral	N/A

Prenons comme base la simple grammaire suivante (qui ne fait rien):

⟨*s*⟩ ::= ⟨*s'*⟩

On va ensuite rajouter un à un les opérateurs dans cette grammaire, de la manière suivante:

1. On trie la table des opérateurs par priorité, dans l'ordre décroissant.
2. Pour chaque type d'opérateur, on crée la règle correspondante. Par exemple, l'opérateur binaire *Addition* dans notre table, aura la règle

   ⟨*Addition*⟩ ::= ⟨*P12*⟩ '+' (⟨*P11*⟩ | ⟨*P12*⟩)

   N.B. Les non-terminaux ⟨*P*11⟩ et ⟨*P*12⟩ sont définis dans la grammaire ci-dessous.
3. On rajoute la nouvelle règle créée dans la grammaire de base

Pour finir, on se retrouve avec une grammaire comme celle-ci:

⟨*s*⟩ ::= ⟨*s'*⟩
⟨*s'*⟩ ::= ⟨*P11*⟩
⟨*P11*⟩ ::= ⟨*Addition*⟩ | ⟨*Soustraction*⟩ | ⟨*P12*⟩
⟨*P12*⟩ ::= ⟨*Division*⟩ | ⟨*Multiplication*⟩ | ⟨*P13*⟩
⟨*P13*⟩ ::= ⟨*Exponentiation*⟩ | ⟨*P14*⟩
⟨*P14*⟩ ::= ⟨*Négation*⟩ | ⟨*P15*⟩
⟨*P15*⟩ ::= '(' ⟨*s*⟩ ')' | ⟨*nombre*⟩

Puisque *LPeg* génère un analyseur lexical de type *PEG*, il faut faire attention à l'ordre de déclaration de la grammaire pour avoir une précédence correcte. En outre, il y a quelques autres limitations à considérer:

1. Si le modèle d'expression autorise les variables, et des mots-clés (ex. `true`, `false`), il est nécessaire de placer les variables à la toute fin de la grammaire, par exemple

   ⟨*P15*⟩ ::= ⟨*nombre*⟩ | ⟨*booléen*⟩ | '(' ⟨*s*⟩ ')' | ⟨*variable*⟩

   Le cas échéant, l'analyseur lexical lira, dans notre exemple, `true` (ou `false`) et considérera le symbole comme étant une variable et non le mot-clé `true`.
2. Les opérateurs qui ne tiennent pas compte de ce qu'il y a "autour" d'eux (litéraux, n-aires) doivent avoir la priorité la plus élevée. Dans un cas comme

   ⟨*P14*⟩ ::= ⟨*nombre*⟩ | ⟨*P15*⟩
   ⟨*P15*⟩ ::= ⟨*Addition*⟩

   la manière décrite plus haut pour construire la grammaire ne fonctionne pas, car ⟨*Addition*⟩ a besoin d'une règle à priorité plus élevée.
3. Il n'est pas possible de donner aux opérateurs une priorité égale ou inférieure à 1. Pour cause: *LPeg* requiert que la première entrée de la table constituant la grammaire soit une chaîne de caractères donnant le nom de la première règle. Exemple simple:

```
grammar = {
  "axiom",
  axiom = lp.R("09")^1
}
```

3.3 Règles utilisées pour les différents types d'opérateurs

Les fonctions pour générer les règles sont stockées dans une table, et retournent une règle *LPeg*. Considérons ici que les espaces blancs (\s, \t, \n) sont ignorés. De plus, notons la "prochaine" règle dans la grammaire par *next_expression*, et la règle courante par *curr_expression*, et la représentation de l'opérateur (e.g. "+") par *op_repr*. Notons aussi l'axiome *s* par *Expression*

3.3.1 Opérateurs binaires

Cette règle est une des plus simples et un des plus évidents.

⟨*binaire*⟩ ::= ⟨*next_expression*⟩ ⟨*op_repr*⟩ (⟨*curr_expression*⟩ | ⟨*next_expression*⟩)

3.3.2 Opérateurs ternaires

Il existe deux variantes d'opérateurs ternaires, la première étant le bloc conditionnel, "if ... then ... else", et l'autre étant le "inline if", par exemple "... ? ... : ...". Pour décrire ces deux opérateurs, on dénombre les représentations des opérateurs par $op\_reprN,\ N \in 1, 3$.

1. Pour la première version, la règle utilisée est

   ⟨*ternaire_v1*⟩ ::= ⟨*op_repr1*⟩ ⟨*Expression*⟩ ⟨*op_repr2*⟩ ⟨*Expression*⟩ ⟨*op_repr3*⟩ ⟨*Expression*⟩

2. Pour la deuxième version, on utilisera

   ⟨*ternaire_v2*⟩ ::= ⟨*next_expression*⟩ ⟨*op_repr1*⟩ ⟨*Expression*⟩ ⟨*op_repr2*⟩ (⟨*curr_expression*⟩ | ⟨*next_expression*⟩)

Il faut cependant faire attention à la priorité qu'on fixe dans les deux cas différents. Le premier cas requiert une priorité élevée (normalement au même niveau que les terminaux), et le deuxième cas nécessite une priorité basse.

3.3.3 Opérateurs unaires préfixes

La règle elle-même n'est pas difficile à définir:

⟨*unaire_préfixe*⟩ ::= ⟨*op_repr*⟩ (⟨*curr_expression*⟩ | ⟨*next_expression*⟩)

Cependant on est dans un cas où il faut aussi donner une priorité élevée à l'opérateur pour qu'il soit reconnu par le reste des opérateurs.

3.3.4 Opérateurs unaires postfixes

Pour ce type d'opérateur, on est obligé de faire une répétition sur la représentation de l'opérateur, à cause du fait qu'on n'a pas la possibilité d'utiliser la récursion à gauche.

⟨*unaire_postfixe*⟩ ::= ⟨*next_expression*⟩ ⟨*op_repr*⟩+

3.3.5 Opérateurs n-aires

Pour représenter les opérateurs n-aires, on utilise la forme représentation_opérateur '(' opérande1, opérande2, ..., opérandeN ')', ce qui nous donne:

⟨*naire*⟩ ::= ⟨*op_repr*⟩ '`(`' ⟨*Expression*⟩ { '`,`' ⟨*Expression*⟩ } '`)`'

3.4 Associativité à gauche

3.4.1 Problème des PEG

On constate que tous les opérateurs binaires dans le tableau précédent sont associatifs à droite. Or, on voudrait pouvoir préciser l'associativité lors de la déclaration des opérateurs, puisque cela modifie l'arbre syntaxique produit par l'analyseur lexical. Dans le cas d'un analyseur *LR*, l'associativité à gauche se traduit simplement par une récursion à gauche, telle que

⟨*Expr*⟩ ::= ⟨*Expr*⟩ '+' ⟨*nombre*⟩ | ⟨*nombre*⟩

Malheureusement, *LPeg* ne gère pas les récursions à gauche, et une élimination de la récursion à gauche du type

⟨*Expr*⟩ ::= ⟨*nombre*⟩ ⟨*Expr'*⟩
⟨*Expr'*⟩ ::= '+' ⟨*Expr'*⟩ | ⟨*nombre*⟩

ne suffit pas. On a encore une associativité à droite.

Il existe des implémentations pour introduire la récursion à gauche dans les grammaires *PEG*, telles que Autumn[1] ou celle proposée par Laurence Tratt[2]. Puisque nous utilisons *LPeg* et Lua, aucune de ces deux méthodes ne sont possibles dans le cadre de ce travail, il nous faut donc notre propre implémentation pour gérer les opérateurs binaires étant déclarés comme étant associatifs à gauche.

3.4.2 Solution pour Ardoises

LPeg nous permet de déclarer des fonctions de capture pour ce qui est lu par l'analyseur généré. Supposons la grammaire suivante:

⟨*s*⟩ ::= ⟨*nombre*⟩ '+' (⟨*s*⟩ | ⟨*nombre*⟩)

Une analyse de l'entrée

```
1 + 2 + 3
```

produira la table

```
{ left = 1, op = +, right = { left = 2, op = +, right = 3 } }
```

Si on a déclaré que notre opérateur *addition* est associatif à gauche, ce résultat est évidemment faux. La table ci-dessus correspond à l'arbre syntaxique de la figure 1. Pour avoir une production correcte,

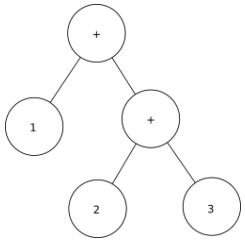


Figure 1

il nous suffit d'opérer de manière récursive sur la table produite. Ainsi on réarrange simplement les nœuds en traversant la table pour obtenir l'arbre représenté sur la figure 2. Il y a cependant certaines

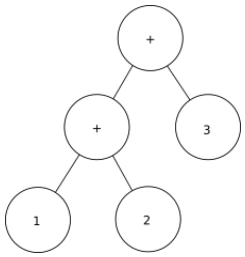


Figure 2

situations où nous sommes obligés de nous arrêter:

1. Le cas où l'opérateur à droite (ou à gauche) n'est pas un opérateur binaire (cela n'a pas de sens de considérer l'associativité gauche / droite pour des opérateurs non binaires)

2. Le cas où l'opérateur à droite (ou à gauche) a une priorité plus élevée que l'opérateur qui essaie d'imposer son associativité à gauche. Effectivement, une expression du type `1 + 2 * 3` doit produire la table `{ left = 1, op = +, right = { left = 2, op = *, right = 3 } }`

Le code décrivant cette transformation se trouve sur mon github (cf. fonction nommée `binary_lassoc`)

3.5 Autres fonctions de capture notables

À part la nécessité de définir la fonction de capture pour l'associativité à gauche, certaines règles nécessitent elles aussi une fonction pour transformer la table produite.

3.5.1 Opérateurs unaires postfixes

L'impossibilité d'utiliser la récursion à gauche avec *LPeg* pose aussi un problème pour les opérateurs postfixes. Si on avait la récursion à gauche, on aurait simplement pu écrire

⟨*unaire_postfixe*⟩ ::= (⟨*curr_expression*⟩ | ⟨*next_expression*⟩) ⟨*op_repr*⟩

Supposons l'opérateur postfixe '%'. Avec la règle précédente on respecte la priorité de déclaration des opérateurs, et on peut aussi capturer une expression du type '3%%%' qui nous donnerait cette production correcte `{ { { 3 % } % } % }`. Sans la récursion à gauche, on ne peut plus capturer l'entièreté de l'expression '3%%%' sans utiliser de répétition. Or la répétition du symbole nous donne une table comme celle-ci: `{ 3 % % % }` ce qui n'est pas ce qu'on veut. On va donc opérer sur la liste qui nous est donnée par *LPeg* pour créer la table juste.

```
local function postfix_capture(p)
  -- function to handle postfix operators when we receive them
  -- The pattern is defined as Exp * op_representation^1,
  -- which means that for an input such as "3~~" we would get
  -- a table such as { 3 ~ ~ }, whereas with this function we get
  -- { { 3 ~ } ~ }
  local function construct(list)
    -- On sait que la table va ressembler à quelque chose
    -- comme { expression ~ ~ ~ ~ }. On commence donc par le dernier élément
    -- de la table, et on crée une nouvelle table de manière récursive
    local function rec(n, t)
      if n == 1 then
        return list[1]
      else
        t = { op = list[n], left = rec(n - 1, { t }) }
      end
      t.op_type = "unary_postfix"
      return t
    end

    return rec(tlen(list), { })
  end

  -- '...' est la liste des arguments
  -- (donne tous les arguments passés à la fonction)
  -- '{ ... }' nous donne simplement une table contenant les arguments
  -- passés à la fonction
  return p / function(...)
    return construct({ ... })
  end
end
```

3.5.2 Opérateurs n-aires

Pour les opérateurs n-aires, la fonction de capture transforme simplement les expressions capturées qui sont en fait un terminal, par exemple "3" ou "variable" en une table.

```
local function nary_capture (p)
  return p / function (op, ...)
    local args = { ... }
    local arglist = { }
    local ctr = 1

    for _, v in ipairs(args) do
      if type(v) ~= "table" then
        arglist[ctr] = { v }
      else
        arglist[ctr] = v
      end
      ctr = ctr + 1
    end

    return { op = op, operands = arglist, op_type = "nary" }
  end
end
```

3.6 Créer une règle pour un nouveau type d'opérateur

Pour permettre aux utilisateurs d'utiliser un type d'opérateur qui n'est pas déjà défini dans le générateur de grammaire, il faut suivre quelques étapes simples. Faisons un exemple, un opérateur quaternaire, étape par étape:

1. Décider du nom du type d'opérateur. Utilisons *quaternary*
2. Créer une entrée dans la table `patterns` avec le même nom que le type d'opérateur, ce qui devrait ressembler à ceci:

   ```
   quaternary = function (operator, curr_expr, next_expr)

   end
   ```

   `operator` est l'objet opérateur passé, ce qui nous permet de récupérer ses propriétés, `curr_expr` est une référence vers la règle elle-même, pour qu'on puisse faire la récursion à droite. Finalement, `next_expr` est une référence vers la règle suivante dans la grammaire (pour la descente récursive).
3. Créer la règle du type d'opérateur. Pour celui-ci, utilisons

   ⟨*quaternary*⟩ ::= ⟨*next_expr*⟩ ⟨*op_repr1*⟩ ⟨*Expression*⟩ ⟨*op_repr2*⟩ ⟨*Expression*⟩ ⟨*op_repr3*⟩ (⟨*quaternary*⟩ | ⟨*next_expr*⟩)

   qui se traduit en Lua par

   ```
   local pattern = (
     next_expr * white * first * white *
     lp.V("axiom") * white * second * white *
     lp.V("axiom") * white * third * white * (curr_expr + next_expr)
   )
   ```

   Quelques explications:

   (a) `lp.V("axiom")` fait référence à l'axiome, c'est-à-dire qu'on autorise toutes les expressions à l'intérieur de l'opérateur.

   (b) `first`, `second` et `third` sont les représentations en caractère (ou chaîne de caractères) de l'opérateur. Ils peuvent être identiques.

   (c) `white` doit obligatoirement être utilisé pour ignorer les caractères blancs (`\s`, `\t`, `\n`).

   (d) Il faut utiliser `next_expr` sur la gauche pour la descente récursive. Sur la droite de la règle, il faut utiliser `curr_expr` pour la récursion à droite (pour pouvoir enchaîner des opérateurs qui ont la même priorité).

4. Définir une fonction de capture. Dans le cas de not re opérateur quaternaire, il est simplement question de retourner une table contenant l'opérateur et les opérandes, puisqu'il n'y a pas de propriétés telles que l'associativité à gauche.
   Les fonctions de capture avec *LPeg* reçoivent en argument ce qui est lu, et les arguments vont dans l'ordre, de gauche à droite. Donc, pour cet opérateur, on définirait la fonction de capture de cette manière:

```
-- p est la règle
function quaternary_capture (p)
  -- On utilise l'opérateur de capture de LPeg
  return p / function (left, op1, middle1, op2, middle2, op3, right)
    -- Et on retourne juste une table.
    -- A savoir: on pourrait utiliser la fonction lpeg.Ct (Capture table),
    -- cependant il est préférable de définir nos clés nous-même, car lpeg.Ct
    -- retourne une table dont les clés sont des nombres (1, 2, ...)
    return {
      left    = left,
      op      = op1,
      op2     = op2,
      op3     = op3,
      middle1 = middle1,
      middle2 = middle2,
      right   = right,
      op_type = "quaternary"
    }
  end
end
```

5. Finaliser le tout. L'entrée `quaternary` dans notre table `patterns` devrait ressembler à ceci:

```
quaternary = function (operator, curr_expr, next_expr)
  -- On commence par récupérer les représentations textuelles de l'opérateur
  local first, second, third = lp.P(operator.operator),
  lp.P(operator.operator1), lp.P(operator.operator2)

  -- On dit à LPeg de capturer ces représentations
  first = lp.C(first)
  second = lp.C(second)
  third = lp.C(third)

  -- On crée la règle
  local pattern = (
    next_expr * white * first * white *
    lp.V("axiom") * white * second * white * lp.V("axiom") * white *
    third * white * (curr_expr + next_expr)
  )

  -- Finalement on retourne la fonction de capture,
  -- et notre règle sera insérée dans notre grammaire globale.
  return quaternary_capture(pattern)
end,
```

6. En définissant notre opérateur de cette façon

```
local quaternary = {
  operator  = "u",
  operator1 = "u",
  operator2 = "u",
  type      = "quaternary",
  priority  = 13
}
```

   on peut maintenant donner

```
1 u 2 u 3 u 4
```

en entrée à l'analyseur ce qui produit la table

```
{
  left    = "1",
  middle1 = "2",
  middle2 = "3",
  right   = "4",
  op      = "u",
  op2     = "u",
  op3     = "u",
  op_type = "quaternary"
}
```

4 Conclusion

La construction des modèles formels d'opérateurs et d'expressions était le point de départ du travail, et malheureusement aussi ce qui a posé le plus de problèmes. Partant d'abord d'un modèle finalement rejeté, puis arrivant à un modèle théoriquement correct mais qui ne fonctionne pas en pratique, cette partie aura crée un contretemps considérable et ne pas avoir trouvé de solution adaptée est plutôt désolant.

Cependant, l'analyse lexicale correspond, à mon avis, bien aux attentes du projet. Utiliser un PEG pour générer une grammaire à partir d'opérateurs fonctionne très bien, malgré les quelques considérations à retenir quant aux priorités qu'on attribue aux opérateurs ainsi que le fait de devoir définir des règles pour les différents types d'opérateur. Il est important de noter toutefois que l'intégration complète de la génération de grammaire n'est pas encore faite, la cause principale étant qu'on n'a pas réussi à constituer et à valider un modèle définitif ni pour les opérateurs, ni pour les expressions.

Finalement, travailler avec *LPeg* pour la génération de grammaire est très intéressant. C'est un analyseur lexical qui répond aux besoins d'Ardoises, par sa forte modularité et sa manipulation facile, malgré le fait qu'il n'accepte pas la récursion à gauche.

Références

[1] Nicolas Laurent, *Parsing Expression Grammars Made Practical*

[2] Laurence Tratt, *Direct Left-Recursive Parsing Expression Grammars*

[3] Alban Linard, Didier Buchs, *Ardoises: Collaborative & Interactive Editing using Layered Data*

[4] Alban Linard, Benoît Barbot, Didier Buchs, Maximilien Colange, Clément Démoulins, Lom Messan Hillah, and Alexis Martin, *Layered Data: a Modular Formal Definition without Formalisms*