Conservatoire National des Arts et Métiers
EPN 5 - Informatique

Année 2018–2019

UE NFP 136 (VARI 2)

Sujet du projet 2018-2019 : Compilation pour JAVA

Présentation du projet

La compilation est l'action de traduire un programme écrit dans un langage source en son équivalent écrit dans un langage cible. En général, on considère que le langage source est un langage de plus ou moins haut niveau (du C, du JAVA, du C++, etc.) et que le langage cible est du langage machine (binaire), mais parfois ce dernier est simplement un langage de plus bas niveau (c'est-à-dire plus proche du langage machine que le langage source).

Par exemple, compiler du code JAVA en fournit une version traduite en *bytecode*, à travers un ou plusieurs fichiers `.class`. Dans ce projet, on se propose de réaliser un compilateur pour traduire du code écrit dans un langage de type "pseudo-code" vers du JAVA. Ce langage, dont le nom éminemment original sera NFP136, sera défini par une grammaire formelle de type BNF (pour *Backus-Naur Form*), telle que décrite en cours. Le compilateur à réaliser sera basé sur une analyse lexicale et syntaxique du langage NFP136 effectuée via un arbre de syntaxe, appelé arbre BNF dans la suite.

Ce document aborde, dans le détail, tous les points à prendre en compte pour écrire un tel compilateur en JAVA, et présente la grammaire BNF décrivant le langage NFP136. Il a pour objectif de vous guider pas à pas dans les différentes étapes de la réalisation de ce projet, et est à lire attentivement.

Étape 1 : la méthode `lireFichierSource`

La première étape consiste à lire le fichier contenant le code source, pour pouvoir ensuite construire l'arbre BNF associé. Pour cela, on stockera tout le contenu de ce fichier dans une unique variable de type `String`, que l'on pourra ensuite aisément manipuler et découper. Le suffixe de chacun des fichiers sources sera `.nfp136` (par analogie avec `.java`, `.c`, ou `.class`), pour indiquer que le code contenu dans ce fichier est écrit dans le langage NFP136.

Pour lire le contenu du fichier, il suffit d'utiliser les méthodes décrites dans le cadre du TP 3. Ainsi, la première possibilité est d'utiliser une variable de type `BufferedReader`. Par exemple, si le fichier à lire (dont on ne détaille ici que les deux premières lignes), de nom "code.nfp136", est le suivant :

```
PROGRAMME Test
methode1():ENTIER
```

Alors, pour accéder au contenu de ce fichier, il suffit d'initialiser une variable `lecteur` de type `BufferedReader`, de la façon suivante :

```
BufferedReader lecteur = new BufferedReader(new FileReader("code.nfp136"));
```

En appelant la méthode `readLine` (qui renvoie une valeur de type `String`) sur cette variable, on lira le contenu de la ligne suivante du fichier de nom "code.nfp136" (la lecture d'un fichier étant séquentielle). Ainsi, dans le cas de ce fichier "code.nfp136", après le premier appel à cette méthode (qui lit la première ligne du fichier), la variable `ligne` contiendra la valeur `"PROGRAMME Test"`. Après un deuxième appel à cette méthode (qui lit la deuxième ligne du fichier), la variable `ligne` contiendra la valeur `"methode1():ENTIER"` :

```
String ligne = lecteur.readLine(); //ligne contient "PROGRAMME Test"
ligne = lecteur.readLine(); //ligne contient "methode1():ENTIER"
```

La construction d'une variable de type `BufferedReader` est susceptible de générer une exception (si le fichier de nom "code.nfp136" n'existe pas, par exemple), et il serait donc préférable de gérer toutes ces instructions dans un bloc `try{...} catch(Exception e){...}`. Enfin, après utilisation, cette variable `lecteur` doit être fermée, à l'aide de l'instruction `lecteur.close();`.

Pour lire le fichier "code.nfp136", il est également possible d'utiliser la classe `Scanner`, de la façon suivante :

```
Scanner lecteur = new Scanner(new File("code.nfp136"));
```

On peut alors lire chaque ligne à l'aide de la méthode `nextLine()` :

```
String ligne = lecteur.nextLine(); //premier appel : ligne="PROGRAMME Test"
ligne = lecteur.nextLine(); //deuxième appel : ligne="methode1():ENTIER"
```

(Dans les deux cas, si la ligne lue est vide, c'est-à-dire ne contient rien d'autre qu'un saut de ligne, on a `ligne.isEmpty()` égal à vrai.) Quelle que soit la méthode utilisée, on mettra bout à bout, dans une unique variable de type `String` (dénommée ici `contenu`), toutes les informations lues sur chacune des lignes du fichiers (en séparant le contenu de deux lignes consécutives par un espace `" "`), à l'aide de l'opérateur de concaténation (`+` en JAVA). Par exemple, on pourra écrire : `contenu = contenu + " " + ligne`.

On aura par la suite besoin de "découper" la chaîne `contenu` selon différents séparateurs (notamment selon l'espace `" "`), et on vous rappelle pour cela l'existence de la méthode `split()` de la classe `String`, qui "découpe" la variable de type `String` sur laquelle la méthode est appelée en morceaux séparés par le symbole passé en paramètre (par exemple `" "`), et stocke les morceaux obtenus dans le tableau d'éléments de type `String` renvoyé.

Étape 2 : la classe `ArbreBNF`

La deuxième étape consiste à définir une classe qui sera utilisée pour représenter le code source par un arbre BNF, dont les feuilles seront des objets terminaux (unités lexicales du langage) et les autres nœuds des objets non terminaux de la grammaire BNF considérée. Cette classe aura pour nom `ArbreBNF`, et s'inspirera de la classe présentée en cours pour définir des arbres non nécessairement binaires par chaînage. Ainsi, chaque objet (variable) de cette classe représentera un nœud de l'arbre considéré (l'arbre complet sera donné par la référence vers sa racine, et correspondra à l'intégralité du programme), et contiendra 5 variables d'instance (appelées aussi *attributs*) ; on rappelle que le bon usage en JAVA est de les déclarer comme `private` :

- Trois références : une vers le père du nœud courant, une vers son premier fils (le plus à gauche), et une vers son frère droit,
- Une variable de type `String` représentant le type du nœud courant,
- Une variable de type `String` représentant la valeur contenue dans le nœud courant (c'est-à-dire la donnée qu'il porte).

Cette classe contiendra évidemment un constructeur, qui pourra initialiser la valeur de chacun de ces 5 attributs. La méthode la plus importante de cette classe construira un arbre BNF à partir des informations récupérées dans le fichier source. Plus précisément, elle utilisera les variables de type `String` représentant les valeurs et types des nœuds de l'arbre en construction pour ajouter un ou plusieurs fils au nœud courant, en fonction du type de ce dernier. Pour cela, il faut spécifier la grammaire BNF qui définit le langage NFP136, et qui est comme suit (la racine sera de type `<programme>`) :

```
<programme> ::= PROGRAMME <id> <corps programme>

<corps programme> ::= DEBUT_MAIN <liste instructions> FIN_MAIN |
<liste declarations methodes> DEBUT_MAIN <liste instructions> FIN_MAIN

<liste declarations methodes> ::= <id>(<liste parametres>):{<type>|RIEN}
DEBUT <liste instructions> FIN | <id>(<liste parametres>):{<type>|RIEN}
DEBUT <liste instructions> FIN <liste declarations methodes>

<liste parametres> ::= <id>:<type> | <id>:<type>,<liste parametres>

<liste instructions> ::= {<declaration>|<affectation>|<affectation chaine>|
<appel>|<boucle tant que>|<boucle pour>|<bloc si>} |
{<declaration>|<affectation>|<affectation chaine>|<appel>|<boucle tant que>|
<boucle pour>|<bloc si>} <liste instructions>
```

<declaration> ::= <id>:<type>

<affectation> ::= <id>:=<expression> | <id>[<id>]:=<expression>

<affectation chaine> ::= <id>:="<chaine de caracteres>"

<appel> ::= APPELER(<id>(<liste parametres appel>)) | RETOURNER(<expression>) |
{AFFICHER|AFFICHER_LIGNE}(<expression>) | {SAISIR|SAISIR_ENTIER}(<id>)

<liste parametres appel> ::= <expression> | <expression>,<liste parametres appel>

<expression> ::= <terme> | <terme>{+|-|/|*}<terme>

<boucle tant que> ::= TANT_QUE <condition> FAIRE <liste instructions> FIN_TANT_QUE

<boucle pour> ::= POUR <id> ALLANT DE <expression> A <expression> FAIRE
<liste instructions> FIN_POUR

<bloc si> ::= SI <condition> ALORS <liste instructions> FIN_SI |
SI <condition> ALORS <liste instructions> SINON <liste instructions> FIN_SI

<terme> ::= APPELER(<id>(<liste parametres appel>)) | <nombre> |
VRAI | FAUX | <id> | <id>[<id>]

<condition> ::= <expression>{=|#|<|>}<expression> |
(<expression>{=|#|<|>}<expression> {ET|OU} <condition>)

<type> ::= ENTIER | CHAINE | TABLEAU | BOOLEEN

<nombre> ::= <chiffre> | <chiffre><nombre>

<chaine de caracteres> ::= {<symbole>|<lettre>|<chiffre>|+|-|/|*|=|#|<|>} |
{<symbole>|<lettre>|<chiffre>|+|-|/|*|=|#|<|>}<chaine de caracteres>

<id> ::= <lettre> | <lettre><suite id>

<suite id> ::= {<lettre>|<chiffre>} | {<lettre>|<chiffre>}<suite id>

<symbole> ::= , | . | : | ; | ? | ! | | [|] | ’ | (|)

<lettre> ::= a | b | ... | z | A | B | ... | Z

<chiffre> ::= 0 | 1 | ... | 9

Le principe de la méthode de la classe `ArbreBNF` en charge de la construction de l'arbre BNF est de parcourir la variable `contenu` de type `String` contenant le code source en langage NFP136, pour identifier dans cette chaîne de caractères les sous-chaînes constituant des terminaux ou des non-terminaux. À chaque fois qu'un non-terminal sera identifié, il faudra ensuite identifier la bonne règle de réécriture associée, parmi celles définies dans la grammaire, puis générer les appels récursifs à cette méthode qui seront nécessaires.

Par exemple, si on identifie dans le code source la présence d'un `SI`, alors cela signifie, du point de vue de notre grammaire BNF, qu'on a identifié la présence d'un non-terminal de type `<bloc si>`, sur lequel on appelle donc cette méthode. Il suffit ensuite de déterminer si ce non-terminal est associé à la première règle de réécriture (dans laquelle il n'y a pas de `SINON` associé au `SI`), ou bien à la seconde (dans laquelle il y a un `SINON` associé au `SI`), sans oublier de vérifier la présence des mots clés `ALORS` et `FIN_SI` (à la fin).

Une fois que cela est fait, on crée deux nouveaux nœuds dans l'arbre BNF, qui seront frères, et dont le père commun est le nœud de l'arbre BNF associé au non-terminal de type `<bloc si>` en cours de traitement : l'un associé au non-terminal de type `<condition>` dans la règle de réécriture, et l'autre associé au non-terminal de type `<liste instructions>` dans la règle de réécriture. Le cas échéant, c'est-à-dire si c'est la seconde règle de réécriture qui s'applique, il faudra aussi créer un troisième fils et lancer un appel récursif sur la partie de la chaîne de caractères qui correspond au non-terminal de type `<liste instructions>` qui se trouve après le `SINON`.

Dans tous les cas, on lance ensuite un appel récursif sur chacun de ces nœuds, en récupérant à chaque fois la valeur de retour. Cette valeur sera égale à `true` si l'appel s'est bien passé, et `false` s'il a généré une erreur (qui sera donc une erreur de syntaxe vis-à-vis de la grammaire BNF considérée).

Si cette valeur est égale à `false` pour l'un des fils d'un nœud, alors on la propagera (à l'aide d'un `return`) vers l'appel récursif précédent, de sorte qu'à la fin des appels elle atteigne la racine de l'arbre BNF.

On fournit en **Annexe 1** quelques indications sur l'implémentation de cette méthode selon le type de nœud (et donc de non-terminal) considéré.

Une fois que l'arbre BNF correspondant au programme a été construit (si une erreur se produit au cours de cette procédure, on affichera `"Erreur de compilation"`), il suffit ensuite de parcourir cet arbre récursivement pour générer le code JAVA associé. La méthode concernée pourra écrire directement dans le fichier de sortie (celui contenant le code JAVA), ou bien générer une variable de type `String` qui sera ensuite écrite dans ce fichier. Le nom de ce fichier sera celui de la classe associée (complété par le suffixe `.java`), qui sera lui-même le nom du programme (valeur du premier fils de la racine). Ainsi, la compilation du fichier `code.nfp136` génèrera le fichier `Test.java`.

Comme dans le cas de la construction de l'arbre BNF, le code JAVA associé à chaque noeud dépendra du type de ce nœud (cf **Annexe 2**).

Pour écrire dans un fichier, on peut, en intégrant toutes les instructions nécessaires au sein d'un bloc `try{...} catch(Exception e){...}`, utiliser une variable de type `BufferedWriter`. Par exemple, pour créer un fichier de nom "FichierSortie.java", il suffit d'initialiser une variable `plume` de type `BufferedWriter` (qui servira ensuite à écrire dedans), de la façon suivante :

```
BufferedWriter plume = new BufferedWriter(new FileWriter("FichierSortie.java"));
```

Pour écrire "Hello world!" dedans, on utilisera alors les lignes suivantes :

```
String phraseAEcrire = new String("Hello world!");
plume.write(phraseAEcrire, 0, phraseAEcrire.length());
```

Une fois qu'on aura fini d'écrire dans le fichier, la variable `plume` doit être fermée, à l'aide de l'instruction `plume.close();`.

Touches finales

Pour compléter le projet, il reste à écrire le (court) programme principal. Ce dernier doit récupérer un paramètre au moment de l'appel du programme (via la variable `args`) : le nom du fichier `.nfp136` contenant le code source.

Il faut dans un premier temps lire ce fichier, et stocker son contenu dans une unique variable de type `String` (cf Étape 1).

Dans un second temps, il faut construire l'arbre BNF associé, puis, si la construction de l'arbre s'est bien passée (sinon, on affichera `"Erreur de compilation"`), générer du code JAVA à partir de cet arbre (cf Étape 2).

Enfin, après génération du fichier `.java` correspondant au fichier source `.nfp136`, on obtiendra le fichier `.class` associé grâce aux lignes suivantes (qui sont à inclure dans un bloc `try{...} catch(Exception e){...}`, et se chargent de faire tous les appels système nécessaires) :

```
Runtime r = Runtime.getRuntime();
Process p = r.exec("javac MaClasse.java"); //la classe à compiler est "MaClasse"
p.waitFor(); //à l'issue de cet appel, le fichier "MaClasse.class" est généré
```

Remarques importantes. Il est **impossible** d'écrire correctement et du premier coup tous les fragments de code JAVA que la réalisation d'un tel compilateur nécessite (cf les exemples en **Annexe 3**). Il est donc **indispensable** d'écrire bout par bout ce projet, en compilant et en testant votre code au fur et à mesure. Les différentes règles de réécriture associées à la grammaire BNF définissant le langage NFP136 doivent donc être prises en compte progressivement, en partant de quelques règles seulement, puis en intégrant les suivantes une par une (après compilation et tests, et en suivant idéalement l'ordre suggéré en **Annexe 4**), jusqu'à les intégrer toutes.

Le projet est à rendre par e-mail, à l'adresse : `cedric.bentz@cnam.fr`. Lors du rendu, il faudra bien évidemment fournir le code source commenté de toutes vos classes (fichiers `.java`), et leur bytecode (fichiers `.class`).

Annexe 1 : construction de l'arbre BNF

Voici des détails concernant la construction de l'arbre BNF associé au code source écrit en langage NFP136, en fonction du type du nœud considéré. On note `valeur` la chaîne de caractères contenant la valeur du nœud courant :

- Si le nœud est de type `<programme>`, on découpera la chaîne `valeur` selon des espaces `" "`, puis on vérifiera que le premier morceau obtenu est `PROGRAMME` (sinon, on renvoie `false`), que le deuxième morceau est bien un non-terminal de type `<id>` (en créant un nouveau nœud fils ayant ce type et comme valeur la partie de la chaîne comprise entre le premier et le second espace, puis en faisant un appel récursif sur ce nœud), et enfin que le troisième morceau est bien un non-terminal de type `<corps programme>` (de nouveau, en créant un second nœud fils ayant ce type et comme valeur la partie de la chaîne située après le second espace, puis en faisant un appel récursif dessus).

- Si le nœud est de type `<corps programme>`, on découpera la chaîne `valeur` selon des espaces `" "`, et on vérifiera la présence de `DEBUT_MAIN` et `FIN_MAIN`. Le morceau de `valeur` qui se trouve avant `DEBUT_MAIN`, s'il existe, devra être un non-terminal de type `<liste declarations methodes>`, et celui entre `DEBUT_MAIN` et `FIN_MAIN` devra être un non-terminal de type `<liste instructions>`. De nouveau, on vérifiera que c'est bien le cas en créant un ou deux fils, qui seront frères, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type `<liste declarations methodes>`, on découpera la chaîne `valeur` selon des espaces `" "`, et on vérifiera la présence de `DEBUT` et `FIN`. Le morceau de `valeur` qui se trouve entre `DEBUT` et `FIN` devra être un non-terminal de type `<liste instructions>`, et, dans le morceau de `valeur` avant `DEBUT`, la partie avant `"("` devra être un non-terminal de type `<id>`, la partie entre `"("` et `")"` devra être un non-terminal de type `<liste parametres>`, et la partie après `"):"` devra être soit un non-terminal de type `<type>`, soit `RIEN`. Le morceau après `FIN`, s'il existe, devra être un non-terminal de type `<liste declarations methodes>`. De nouveau, on vérifiera que c'est bien le cas en créant quatre ou cinq fils, qui seront frères, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type `<liste parametres>`, on découpera la chaîne `valeur` selon des virgules `","`, puis, à l'aide d'une boucle, on découpera chacun des morceaux obtenus selon `":"`. On vérifiera ensuite qu'ainsi on obtient bien des couples de non-terminaux de types respectifs `<id>` et `<type>`. De nouveau, cette vérification se fera en créant un nombre de fils égal à deux fois le nombre de paramètres, qui seront tous frères, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type `<liste instructions>`, le processus est un peu plus complexe. Il faut réussir à identifier quel type d'instruction (et donc de non-terminal) est au début de la chaîne `valeur`. Si, après découpage de `valeur` selon des espaces `" "`, le premier morceau obtenu contient `":="`, alors, soit il contient en plus un `"`, auquel cas ce doit être un non-terminal de type `<affectation chaine>`, soit ce doit être un non-terminal de type `<affectation>`. Sinon, s'il contient `":"`, alors ce doit être un non-terminal de type `<declaration>`. Enfin, s'il est égal à `SI`, `POUR`, ou `TANT_QUE`, alors ce doit être un non-terminal de type `<bloc si>`, `<boucle pour>`, ou `<boucle tant que>`, respectivement. Dans tous les autres cas, ce doit être un non-terminal de type `<appel>`. À chaque fois, après avoir identifié le type du non-terminal, il faudra également identifier correctement la partie de la chaîne `valeur` concernée. Dans le cas de `<affectation>`, `<declaration>`, et `<appel>`, c'est assez simple. Dans le cas de `<affectation chaine>`, on découpera `valeur` selon `" :` le premier morceau devra alors être un non-terminal de type `<id>` suivi de `":="`, et le second morceau un non-terminal de type `<chaine de caracteres>`. Enfin, dans le cas de `<bloc si>`, `<boucle pour>`, ou `<boucle tant que>`, il faudra identifier le `FIN_SI`, `FIN_POUR` ou `FIN_TANT_QUE` correspondant, respectivement, en faisant attention aux instructions imbriquées (une `<boucle tant que>` dans une `<boucle tant que>`, ou un `<bloc si>` dans un `<bloc si>`, etc.). De nouveau, on vérifiera que tous les morceaux obtenus sont corrects en créant un fils dont le type correspond au non-terminal identifié, et, si `valeur` ne contient pas que ce non-terminal, un autre fils de type `<liste instructions>` qui sera son frère droit, puis en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type `<declaration>`, on procèdera comme dans le cas `<liste parametres>` (sauf qu'il n'y aura que deux fils).

- Si le nœud est de type `<affectation>`, on découpera la chaîne `valeur` selon `":="`. Ensuite, on vérifiera que le premier morceau obtenu est soit un non-terminal de type `<id>` (si ce morceau ne contient pas `"["`), soit un non-terminal de type `<id>`, suivi d'un `"["`, suivi d'un non-terminal de type `<id>`, suivi d'un `"]"` (en découpant ce morceau selon `"["`, à l'aide de `split("\\[")`, puis selon `"]"`), et en créant un ou deux nœuds fils de type `<id>`, respectivement. Enfin, on vérifiera que le second morceau obtenu est bien un non-terminal de type `<expression>`, en créant un nouveau nœud fils ayant ce type, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type `<affectation chaine>`, on découpera la chaîne `valeur` selon `":="`, puis on vérifiera que le premier morceau obtenu est un non-terminal de type `<id>`, et que le second morceau est `"`

suivi d'un non-terminal de type `<chaine de caracteres>` suivi d'un `"` (en découpant ce morceau selon `"`). De nouveau, on effectuera la vérification des morceaux obtenus en créant deux fils, qui seront frères, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type `<bloc si>` ou `<boucle tant que>`, le principe sera peu ou prou le même dans les 2 cas. Pour commencer, on découpera la chaîne `valeur` selon des espaces `" "`. Ensuite, on vérifiera que le premier morceau obtenu est égal à `SI` (respectivement `TANT_QUE`), le dernier à `FIN_SI` (respectivement `FIN_TANT_QUE`), et on déterminera l'indice du morceau égal à `ALORS` (respectivement `FAIRE`). Il faut faire attention à identifier le bon `ALORS` (respectivement le bon `FAIRE`), car il peut y avoir plusieurs `<bloc si>` ou `<boucle tant que>` imbriqués. Pour cela, il faut parcourir les morceaux et compter le nombre de `SI` et de `FIN_SI` (respectivement de `TANT_QUE` et de `FIN_TANT_QUE`) rencontrés : quand ces nombres diffèrent de $+1$, on a trouvé le bon. La partie comprise entre `SI` et `ALORS` (respectivement entre `TANT_QUE` et `FAIRE`) devra alors être un non-terminal de type `<condition>`, et celle comprise entre `ALORS` et `FIN_SI` (respectivement entre `FAIRE` et `FIN_TANT_QUE`) devra être un non-terminal de type `<liste instructions>`. Dans le cas où il y a un `SINON` correspondant au `SI`, il faudra identifier la partie comprise entre ce `SINON` (encore une fois, il faudra identifier le bon `SINON`, s'il y a plusieurs `<bloc si>` imbriqués) et `FIN_SI`, qui devra être un non-terminal de type `<liste instructions>`. De nouveau, on vérifiera que tous les morceaux obtenus sont corrects en créant deux ou trois fils, qui seront frères, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type `<boucle pour>`, la méthode sera similaire au cas précédent. Pour commencer, on découpera la chaîne `valeur` selon des espaces `" "`. Ensuite, on vérifiera que le premier morceau obtenu est égal à `POUR`, que le deuxième est un non-terminal de type `<id>`, que le troisième est égal à `ALLANT`, que le quatrième est égal à `DE`, que le cinquième est un non-terminal de type `<expression>`, que le sixième est égal à `A`, que le septième est un non-terminal de type `<expression>`, que le huitième est égal à `FAIRE`, et que la concaténation des suivants (hormis le dernier, égal à `FIN_POUR`) est un non-terminal de type `<liste instructions>`. De nouveau, on vérifiera que tous les morceaux obtenus sont corrects en créant quatre fils, qui seront frères, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type `<appel>`, on découpera la chaîne `valeur` selon `"("`. Si le premier morceau obtenu est `APPELER`, alors le deuxième morceau devra être un non-terminal de type `<id>`, et le troisième morceau, une fois découpé selon `")"`, devra être un non-terminal de

type <liste parametres appel>. Si le premier morceau obtenu est SAISIR ou SAISIR_ENTIER, alors le deuxième morceau, une fois découpé selon ")", devra être un non-terminal de type <id>. Si le premier morceau obtenu est RETOURNER, AFFICHER ou AFFICHER_LIGNE, alors le deuxième morceau, une fois découpé selon ")", devra être un non-terminal de type <expression>. Dans tous les autres cas, on renverra false. De nouveau, on vérifiera que tous les morceaux obtenus sont corrects en créant deux ou trois fils, qui seront tous frères, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type <liste parametres appel>, on découpera la chaîne valeur selon des virgules "," ; chaque morceau ainsi obtenu devra alors être un non-terminal de type <expression>. De nouveau, on vérifiera que c'est bien le cas en créant un nombre de fils égal au nombre de morceaux, qui seront tous frères, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type <expression>, alors on cherchera, dans la chaîne valeur, la présence d'un signe "+", "-", "/" ou "*" en-dehors d'un terme. Pour cela, on procèdera comme dans le cas de <bloc si> et <boucle tant que>, en parcourant valeur tout en comptant le nombre de "(" et de ")" rencontrés : si ces deux nombres sont égaux au moment où on trouve un signe "+", "-", "/" ou "*", alors cette expression contient bien deux termes. Si c'est le cas, la partie de valeur située avant le signe trouvé devra être un non-terminal de type <terme>, et celle située après également, ce qu'on vérifiera en créant trois fils, qui seront tous frères, et en faisant un appel récursif sur chacun des fils créés. Sinon, valeur devra être un non-terminal de type <terme>.

- Si le nœud est de type <terme>, on découpera la chaîne valeur selon "(". Si le premier morceau obtenu est APPELER, alors valeur devra être un non-terminal de type <appel>. Si valeur commence par un chiffre ('0' à '9'), alors ce devra être un non-terminal de type <nombre>. Si valeur contient "[", alors le premier morceau obtenu après l'avoir découpée selon "[" devra être un non-terminal de type <id>, et le second morceau (sans le dernier "]") devra également être un non-terminal de type <id>. Sinon, soit valeur vaudra VRAI ou FAUX, soit valeur devra être un non-terminal de type <id>. De nouveau, on vérifiera que tous les morceaux obtenus sont corrects en créant un ou deux nœuds fils, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type <condition>, on découpera la chaîne valeur selon des espaces " ". Si la chaîne commence par une parenthèse ouvrante "(", alors il doit s'agir d'une condition formée par au moins deux conditions reliées par un OU ou un ET. Dans ce cas, la partie du

premier morceau obtenu après découpage qui commence juste après la première `"("`, et qui s'arrête juste avant le premier signe `"="`, `"#"`, `"<"` ou `">"` trouvé, devra être un non-terminal de type `<expression>`, et la partie du premier morceau qui commence après ce signe devra également être un non-terminal de type `<expression>`. Le deuxième morceau obtenu après découpage de `valeur` selon `" "` devra être égal à `OU` ou `ET`, et le troisième morceau (sauf la dernière parenthèse fermante `")"`) devra être un non-terminal de type `<condition>`. Notons que, si la chaîne `valeur` ne commence pas par `"("`, alors on n'obtiendra qu'un seul morceau après l'avoir découpée, qui sera traité comme ci-dessus. De nouveau, on vérifiera que tous les morceaux obtenus sont corrects en créant de trois à cinq nœuds fils, tous frères, et en faisant un appel récursif sur chacun des fils créés.

- Si le nœud est de type `<type>`, on vérifiera que sa valeur est `ENTIER`, `CHAINE`, `TABLEAU` ou `BOOLEEN` (sans appel récursif).

- Si le nœud est de type `<nombre>`, on vérifiera que chaque caractère composant sa valeur est un chiffre, et est donc compris entre '0' et '9' (sans appel récursif).

- Si le nœud est de type `<chaine de caractères>`, on vérifiera que chaque caractère composant sa valeur est soit un chiffre, donc compris entre '0' et '9', soit une lettre, donc compris entre 'a' et 'z' ou entre 'A' et 'Z', soit un des caractères autorisés par la grammaire BNF (sans appel récursif).

- Si le nœud est de type `<id>`, on vérifiera que le premier caractère composant sa valeur est une lettre, donc compris entre 'a' et 'z' ou entre 'A' et 'Z', et que chacun des autres caractères la composant (s'ils existent) est soit un chiffre, donc compris entre '0' et '9', soit une lettre (sans appel récursif).

Annexe 2 : génération du code JAVA

Voici des indications concernant la génération du code JAVA à partir de l'arbre BNF, en fonction du type du nœud considéré :

- Si le nœud est de type `<programme>`, on écrira l'en-tête de la classe JAVA, à savoir `"import java.util.*;\n\npublic class "` (pour simplifier, on importera par défaut le package `util`), suivi de l'affichage de son premier fils (via un appel récursif), d'une accolade ouvrante (et d'un ou deux sauts de ligne), de l'affichage de son second fils (via un appel récursif), et finalement d'une accolade fermante.

- Si le nœud est de type `<corps programme>`, on écrira l'affichage de son premier fils (via un appel récursif), s'il a deux fils, puis on écrira l'en-tête `"public static void main(String[] args) "`, suivi d'une accolade ouvrante et d'un saut de ligne, de l'affichage de son second fils (ou de son premier fils, s'il n'en a qu'un), via un appel récursif, et finalement d'une accolade fermante suivie d'un saut de ligne.

- Si le nœud est de type `<liste declarations methodes>`, on écrira `"public static "`, suivi de `void` si la valeur de son troisième fils est `RIEN` et de l'affichage de ce troisième fils (via un appel récursif) sinon, suivi d'un espace, de l'affichage de son premier fils (via un appel récursif), d'une parenthèse ouvrante, de l'affichage de son deuxième fils (via un appel récursif), d'une parenthèse fermante, d'une accolade ouvrante et d'un saut de ligne, de l'affichage de son quatrième fils (via un appel récursif), d'une accolade fermante et d'un ou deux sauts de ligne, et de l'affichage de son cinquième fils (via un appel récursif) le cas échéant.

- Si le nœud est de type `<liste instructions>`, on écrira successivement l'affichage de tous ses fils (via des appels récursifs).

- Si le nœud est de type `<boucle tant que>`, on écrira `"while("`, suivi de l'affichage de son premier fils (via un appel récursif), d'une parenthèse fermante, d'une accolade ouvrante, d'un saut de ligne, de l'affichage de son deuxième fils (via un appel récursif), d'une accolade fermante, et d'un saut de ligne.

- Si le nœud est de type `<boucle pour>`, on écrira `"for("`, suivi de l'affichage de son premier fils (via un appel récursif), d'un signe égal `"="`, de l'affichage de son deuxième fils (via un appel récursif), d'un point virgule `";"`, de l'affichage (le deuxième) de son premier fils (via un appel récursif), d'un signe `"<="`, de l'affichage de son troisième fils (via un appel récursif), d'un point virgule `";"`, de l'affichage (le troisième) de son premier fils (via un appel récursif), de `"++) {\n"`, et enfin de l'affichage de son quatrième fils (via un appel récursif) et d'une accolade fermante `"}"`, elle-même suivie d'un saut de ligne.

- Si le nœud est de type `<bloc si>`, on écrira `"if("`, suivi de l'affichage de son 1er fils (via un appel récursif), de `")"`, de `"{"`, d'un saut de ligne, de l'affichage de son 2ème fils (via un appel récursif), de `"}"`, et d'un saut de ligne. On s'arrêtera là si le nœud n'a que 2 fils. Sinon, on écrira ensuite `"else"`, suivi de `"{"`, d'un saut de ligne, de l'affichage de son 3ème fils (via un appel récursif), de `"}"` et d'un saut de ligne.

- Si le nœud est de type `<liste parametres>`, on écrira l'affichage de son *deuxième* fils (via un appel récursif) suivi d'un espace et de l'affichage de son *premier* fils (via un appel récursif), puis une virgule et un espace suivis de l'affichage de son *quatrième* fils (via un appel récursif), d'un espace et de l'affichage de son *troisième* fils (via un appel récursif), etc.

- Si le nœud est de type `<affectation chaine>`, on écrira l'affichage de son premier fils (via un appel récursif) suivi du signe égal `"="`, d'un guillemet double (via `"\""`), de la valeur de son deuxième fils, d'un guillemet double, d'un point virgule `";"`, et enfin d'un saut de ligne.

- Si le nœud est de type `<affectation>`, on écrira l'affichage de son premier fils (via un appel récursif) suivi du signe égal `"="`, de l'affichage de son deuxième fils (via un appel récursif), d'un point virgule `";"`, et d'un saut de ligne, si le nœud possède deux fils. Sinon, on écrira l'affichage de son premier fils (via un appel récursif) suivi de `"["`, de l'affichage de son deuxième fils (via un appel récursif), de `"]"`, du signe égal `"="`, de l'affichage de son troisième fils (via un appel récursif), d'un point virgule `";"`, et d'un saut de ligne.

- Si le nœud est de type `<declaration>`, on écrira l'affichage de son *deuxième* fils (via un appel récursif) suivi d'un espace et de l'affichage de son *premier* fils (via un appel récursif). Si la valeur de son deuxième fils n'est pas `"TABLEAU"`, on écrira ensuite un point virgule `";"` suivi d'un saut de ligne. Sinon, on écrira `" = new int[1000]"`, suivi d'un point virgule `";"` et d'un saut de ligne.

- Si le nœud est de type `<expression>`, on écrira l'affichage de son premier fils (via un appel récursif) s'il n'a qu'un fils. Sinon, on écrira l'affichage de son premier fils (via un appel récursif), suivi de la valeur de son deuxième fils (soit `"+"`, `"-"`, `"/"`, ou `"*"`), et de l'affichage de son troisième fils (via un appel récursif).

- Si le nœud est de type `<terme>`, on écrira l'affichage de son premier fils (via un appel récursif), suivi de `"["`, de l'affichage de son second fils (via un appel récursif), et de `"]"`, si ce nœud possède deux fils. Dans le cas contraire, on écrira `true` si sa valeur est `"VRAI"`, `false` si sa valeur est `"FAUX"`, et l'affichage de son premier fils (via un appel récursif) dans tous les autres cas.

- Si le nœud est de type `<condition>`, on écrira l'affichage de son premier fils (via un appel récursif) suivi du signe associé à la condition et de l'affichage de son troisième fils (via un appel récursif), si le nœud possède trois fils. Sinon, on écrira une parenthèse ouvrante, suivie de l'affichage de son premier fils (via un appel récursif), du signe associé à la condition, de l'affichage de son troisième fils (via un appel récursif), de `"&&"` ou `"||"` (si la valeur de son quatrième fils est `"ET"` ou `"OU"`, respectivement), de l'affichage de son cinquième fils (via un appel récursif), et d'une parenthèse fermante. Il est à noter que le signe associé à la condition sera `"=="`, `"!="`, `"<"` ou `">"`, suivant que la valeur du deuxième fils de ce nœud est `"="`, `"#"`, `"<"` ou `">"`, respectivement.

- Si le nœud est de type `<appel>`, alors ce qu'on écrira dépendra de la valeur de son premier fils. S'il s'agit de `"APPELER"`, alors on écrira l'affichage de son deuxième fils (via un appel récursif), suivi d'une parenthèse ouvrante, de l'affichage de son troisième fils (via un appel récursif), d'une parenthèse fermante ; si le type de son père est `<liste instructions>` (et cet appel est donc la seule instruction d'une ligne), on écrira en plus un point virgule `";"` et un saut de ligne. S'il s'agit de `"SAISIR"`, alors on écrira l'affichage de son deuxième fils (via un appel récursif) suivi de `" = (new Scanner(System.in)).next();\n"`. S'il s'agit de `"SAISIR_ENTIER"`, alors on écrira la même chose que dans le cas de `"SAISIR"`, à ceci près qu'on remplacera `next()` par `nextInt()`. S'il s'agit de `"AFFICHER"`, alors on écrira `"System.out.print("`, suivi de l'affichage de son deuxième fils (via un appel récursif), d'une parenthèse fermante, d'un point virgule `";"`, et d'un saut de ligne. S'il s'agit de `"AFFICHER_LIGNE"`, alors on écrira `"System.out.println("`, suivi de l'affichage de son deuxième fils (via un appel récursif), d'une parenthèse fermante, d'un point virgule `";"`, et d'un saut de ligne. Enfin, s'il s'agit de `"RETOURNER"`, alors on écrira `"return("`, suivi de l'affichage de son deuxième fils (via un appel récursif), d'une parenthèse fermante, d'un point virgule `";"`, et d'un saut de ligne.

- Si le nœud est de type `<liste parametres appel>`, on écrira l'affichage de son premier fils (via un appel récursif), puis `", "` suivi de l'affichage de son deuxième fils (via un appel récursif), puis de nouveau `", "` suivi de l'affichage de son troisième fils (via un appel récursif), etc.

- Si le nœud est de type `<type>`, on écrira `"int"`, `"String"`, `"int[]"` ou `"boolean"`, suivant que la valeur de ce nœud est `"ENTIER"`, `"CHAINE"`, `"TABLEAU"` ou `"BOOLEEN"`, respectivement.

- Si le nœud est de type `<id>` ou `<nombre>` (ou même `<chaine de caracteres>`, le cas échéant), on écrira simplement la valeur du nœud.

Annexe 3 : exemples de code JAVA

Cette annexe fournit le bout de code JAVA associé à la 1ère étape de la construction de l'arbre BNF, c'est-à-dire au traitement de sa racine (qui est un nœud de type `<programme>`, d'après la grammaire BNF associée) :

```
if(type.equals("<programme>")) { //fils 1 = id. du progr., fils 2 = corps du progr.
 String[] valeurs = valeur.split(" "); //on découpe la chaîne valeur selon les " "
 if(!(valeurs[0].equals("PROGRAMME")&&valeurs.length>2))
  return false; //on renvoie faux si la règle de réécriture n'est pas respectée
 premierFils = new ArbreBNF(this, null, null, "<id>", valeurs[1]);
 if(!premierFils.constructionArbreBNF())
  return false; //on renvoie faux si le 1er fils n'est pas de type <id>
 String corps = new String(valeurs[2]);
 for(int i=3;i<=valeurs.length-1;i++) //corps = les autres morceaux de valeur
  corps = new String(corps + " " + valeurs[i]);
 premierFils.frereDroit=new ArbreBNF(this, null, null, "<corps programme>", corps);
 if(!premierFils.frereDroit.constructionArbreBNF())
  return false; //on renvoie faux si le reste n'est pas de type <corps programme>
}
```

Dans ce code, le type d'un nœud est appelé `type`, la chaîne contenant la valeur d'un nœud est appelée `valeur` (comme dans le reste du document), le premier fils d'un nœud est appelé `premierFils`, le frère droit d'un nœud est appelé `frereDroit`, et la méthode récursive en charge de la construction de l'arbre BNF est appelée `constructionArbreBNF`. En outre, dans le constructeur de la classe `ArbreBNF` considéré ici, le premier paramètre est le père du nœud, le deuxième son premier fils, le troisième son frère droit, le quatrième son type et le cinquième sa valeur.

La structure de cette méthode sera donc simplement de traiter un par un tous les types de nœuds possibles, à l'aide de `if` successifs (sur le modèle du `if(type.equals("<programme>"))` illustré ici), en lançant à chaque fois un appel récursif sur chacun des fils créés. Si une erreur syntaxique ou lexicale est identifiée en parcourant la chaîne `valeur` considérée, on renverra `false`. Sinon, on renverra `true` à la fin de la méthode.

De la même façon, si la méthode récursive qui génère le code JAVA associé à l'arbre BNF s'appelle `codeJava`, on générera le code JAVA associé à la racine (nœud `<programme>`) de l'arbre BNF à l'aide du code suivant :

```
if(type.equals("<programme>")) {
 String ligne = "import java.util.*;\n\npublic class " + premierFils.codeJava();
 ligne = ligne + " {\n\n" + premierFils.frereDroit.codeJava() + "}";
 return ligne;
}
```

Annexe 4 : intégration progressive des règles de réécriture

Voici une suggestion d'ordre dans lequel intégrer, une à une, les règles de réécriture de la grammaire BNF du langage NFP136 :

1. Écrire `<programme>`.
2. Écrire `<corps programme>` (sans y inclure `<liste declarations methodes>`, pour l'instant).
3. Écrire `<id>`, `<chaine de caracteres>`, `<type>`, et `<nombre>`.
4. Écrire `<declaration>`.
5. Écrire `<affectation chaine>`.
6. Écrire `<liste instructions>` (en y incluant seulement `<declaration>` et `<affectation chaine>`, pour l'instant).

   (On pourra tester le code écrit jusque là sur le fichier `prgmTest1.nfp136`.)
7. Écrire `<terme>` (sans y inclure `APPELER`, pour l'instant).
8. Écrire `<expression>`.
9. Écrire `<affectation>`, puis l'intégrer dans `<liste instructions>`.

   (On pourra tester le code écrit jusque là sur le fichier `prgmTest2.nfp136`.)
10. Écrire `<liste parametres>`.
11. Écrire `<liste declarations methodes>`, puis l'intégrer dans `<corps programme>`.

    (On pourra tester le code écrit jusque là sur le fichier `prgmTest3.nfp136`.)
12. Écrire `<liste parametres appel>`.
13. Écrire `<appel>` (et finir `<terme>`), puis l'intégrer dans `<liste instructions>`.

    (On pourra tester le code écrit jusque là sur le fichier `prgmTest4.nfp136`.)
14. Écrire `<boucle pour>`, puis l'intégrer dans `<liste instructions>`.

    (On pourra tester le code écrit jusque là sur le fichier `prgmTest5.nfp136`.)
15. Écrire `<condition>`.
16. Écrire `<boucle tant que>` et `<bloc si>`, puis finir `<liste instructions>`.

    (On pourra tester le code écrit sans `<bloc si>` sur le fichier `prgmTest6.nfp136`, puis avec `<bloc si>` sur tous les autres.)