

1 Présentation générale

Partant d'un analyseur lexical et d'un analyseur syntaxique, l'objectif de ce projet est de réaliser un compilateur pour le langage NILNOVI PROCÉDURAL. La structure du compilateur est illustrée sur la figure 1. La première brique est l'analyseur lexical qui est chargé d'interpréter le code source du programme et de le segmenter en unités lexicales. Les différentes unités lexicales reconnues sont des types suivants : mot-clé (ex : *while*), symbole (ex : \leq), caractère (ex : `' : '`), entier (ex : 23), identificateur et fin de l'entrée. La suite de ces éléments reconnus par l'analyseur lexical constitue la chaîne codée qui est fournie en entrée de l'analyseur syntactico-sémantique.

Ce dernier est chargé de réaliser l'analyse syntaxique (vérifier que l'agencement des unités lexicales est correct) et l'analyse sémantique (la sémantique des différentes unités lexicales les unes par rapport aux autres est correcte, par exemple en terme de types). Lors de l'analyse syntaxique, la table des identificateurs est complétée. Elle stocke les différentes informations nécessaires aux vérifications sémantiques et à la génération de code. Enfin, l'analyseur syntaxique pilote la génération de code qui s'effectue en parallèle.

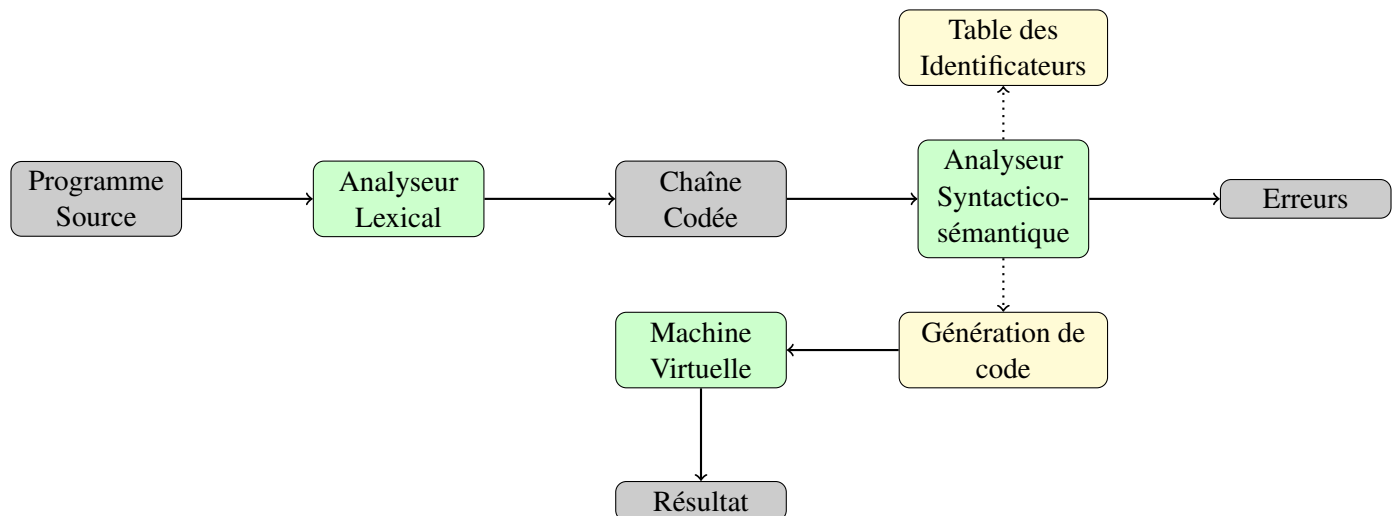


FIGURE 1 – Structure du compilateur NNP

2 Le langage à compiler

Cette section donne un exemple de programme rédigé en NILNOVI PROCÉDURAL. Il s'agit ici d'un programme réalisant le calcul de la fonction factorielle. La définition détaillée du langage est celle réalisée en cours et disponible dans le polycopié de cours.

Le langage permet la définition de fonctions et de procédures qui peuvent prendre des paramètres d'entrée (*in*) ou d'entrée-sortie (*in out*). La définition des méthodes intervient avant la déclaration des variables globales, ce qui interdit leur utilisation dans les méthodes définies. De manière générale, tout élément doit être défini avant utilisation, ce qui interdit notamment l'usage de la récursivité croisée. Enfin, les constructions classiques sont utilisables (alternative simple/double, boucle *while*).

```

01 : procedure pp is
02 :   function fact(i: integer) return integer is
03 :     //pré : i>=0
04 :     //post : résultat : i!
05 :   begin
06 :     if i=0 then
07 :       return 1
08 :     else
09 :       return i*fact(i-1)
10 :     end
11 :   end;
12 :   procedure p(i: integer) is
13 :     //pré : i>=0
14 :     //post : fact(0) à fact(i-1) sont affichés
15 :     j: integer;
16 :   begin
17 :     j:=0;
18 :     while j/=i loop
19 :       //fact(0) à fact(j-1) sont affichés
20 :       put (fact(j));
21 :       j:=j+1
22 :     end
23 :   end;
24 :   k: integer;
25 : begin
26 :   get(k) ;
27 :   if k>=0 then
28 :     p(k)
29 :   end
30 : end.

```

3 Analyseur lexical

Le rôle de l'analyseur lexical présenté ici est, partant d'un programme source NILNOVI PROCÉDURAL, de construire une structure appelée "chaîne codée". Alors qu'un programme source est constitué d'une suite de caractères, une chaîne codée est une liste d'éléments lexicaux (comme les entiers, les mots clés, les identificateurs, certains caractères particuliers) où les caractères ont perdus leur individualité et peuvent avoir été regroupés (comme dans un identificateur) ou même ont disparus (comme dans la représentation d'un entier). L'analyseur lexical de NILNOVI PROCÉDURAL a également comme rôle annexe :

- de lire le programme source depuis un fichier,
- de faire disparaître les éléments inutiles à l'analyse syntaxique comme les espaces, les tabulations et les commentaires.

NILNOVI PROCÉDURAL distingue 6 types d'unités lexicales. Chaque type est modélisé par une classe, elles ont en commun les numéros de ligne et de colonne où débute l'unité en question.

- entier : contient la représentation binaire de l'entier considéré.
- identificateur : contient la chaîne constituant l'identificateur.
- mot-clé : contient la chaîne de caractères constituant le mot clé.
- caractère : pour les unités qui ne sont ni des entiers ni des identificateurs ni des mots clés, contient le caractère

Numéro	Type	Valeur
1	Keyword	procedure
2	Identifier	pp
3	Keyword	is
4	Identifier	i
5	Character	,
6	Identifier	som
7	Character	:
8	Type	integer
9	Character	;
10	Keyword	begin
11	Identifier	som
12	Keyword	:=
13	Integer	0
14	Character	;
15	Identifier	i
16	Keyword	:=
17	Integer	23
18	Keyword	end
19	Character	.
20	Fel	

TABLE 1 – Chaîne codée produite par l’analyseur lexicale.

considéré.

- symbole : certains ensembles de caractères comme ' \leq ' ou ' \geq '.
- fel : unité fictive achevant la chaîne codée.

L’analyse lexicale du programme suivant rédigé en NILNOVI PROCÉDURAL produit la chaîne codée présentée dans la table 1.

```

01 : procedure pp is
03 :      // un exemple
03 :      i, som: integer;
05 : begin
06 :      som := 0
07 :      i := 23
08 : end.

```

4 Analyseur syntaxique

Pendant le cours, nous avons étudié la grammaire NILNOVI PROCÉDURAL présentée en annexe A. La grammaire du cours est une grammaire LR(1). Cette grammaire a été "raffinée" en grammaire LL(1) afin de faciliter son codage. Cette grammaire LL(1) a elle-même été "raffinée" en diagrammes syntaxiques présentés dans l’annexe B. Des "panneaux d’aiguillage" ont été apposés pour vérifier et exploiter le caractère déterministe de la description. Ces diagrammes ont alors été codés en Python.

5 Fourniture et travail à réaliser

Une implémentation de l'analyseur lexical ainsi que de l'analyseur syntaxique est fournie. L'analyseur syntaxique est à compléter, il ne réalise pour l'instant que l'analyse simple du texte fourni par rapport aux règles de la grammaire. En particulier, la génération de code est à réaliser ainsi que les vérifications sémantiques (notamment la vérification des types). Le langage utilisé pour la réalisation de ces deux briques logicielles est le *Python*¹. Il ne s'agit pas d'une contrainte du développement, vous pouvez donc effectuer une réécriture dans un autre langage si vous le souhaitez.

Le travail à réaliser comprend donc les tâches suivantes :

- construction de la table des identificateurs : certaines informations doivent être stockées afin de rendre la compilation possible (par ex : noms, type, adresse des identificateurs)
- génération du code objet. Afin de faciliter la réalisation, cette tâche peut être avantageusement découpée en sous-étapes :
 - identifier, spécifier et placer les points de génération dans l'analyseur syntaxique,
 - coder les points de génération,
 - insérer les appels aux points de génération.
- conception et écriture de la machine virtuelle : cette dernière pourra être implantée dans le langage de votre choix. Elle traitera les instructions du code objet données en entrée et exécutera le programme objet.
- vérifications sémantiques : il s'agit d'identifier les diagrammes syntaxiques qui doivent retourner un résultat afin de permettre les vérifications nécessaires (par exemple de type).

Pour le déroulement du projet, vous devez créer un dépôt Git sur <https://gitlab.enssat.fr> et y gérer vos sources. Ce dépôt Git comprend également un Wiki qui devra être mis à jour à chaque séance avec un état de l'avancement des tâches (prévues, en cours, réalisées) ainsi que l'affectation des différentes tâches à chaque membre du projet.

Vous devez inviter l'utilisateur *lolive* à rejoindre votre dépôt.

Le bonne tenue du Wiki, la mise à jour du dépôt, la documentation du code, notamment, seront pris en compte dans l'évaluation.

Une démonstration de la réalisation sera effectuée à la fin du projet et permettra d'évaluer le niveau de fonctionnalité du logiciel.

Le projet est à réaliser par groupes de 5 personnes. La date de rendu du projet est fixée au **05 juin 2020**.

1. De nombreux exemples sont disponibles sur la toile. Notamment, le site officiel contient une documentation assez riche <http://www.python.org>.

Annexes

A Grammaire du langage

Les règles suivantes définissent la grammaire du langage NILNOVI PROCÉDURAL. La règle racine est $\langle \text{programme} \rangle$.

$\langle \text{programme} \rangle ::= \langle \text{specifProgPrinc} \rangle \text{ is } \langle \text{corpsProgPrinc} \rangle$
 $\langle \text{corpsProgPrinc} \rangle ::= \langle \text{partieDecla} \rangle \text{ begin } \langle \text{suiteInstr} \rangle \text{ end } .$
 $\quad | \text{ begin } \langle \text{suiteInstr} \rangle \text{ end } .$
 $\langle \text{specifProgPrinc} \rangle ::= \text{procedure } \langle \text{ident} \rangle$
 $\langle \text{partieDecla} \rangle ::= \langle \text{listeDeclaOp} \rangle \langle \text{listeDeclaVar} \rangle | \langle \text{listeDeclaVar} \rangle | \langle \text{listeDeclaOp} \rangle$
 $\langle \text{listeDeclaOp} \rangle ::= \langle \text{declaOp} \rangle ; \langle \text{listeDeclaOp} \rangle | \langle \text{declaOp} \rangle ;$
 $\langle \text{declaOp} \rangle ::= \langle \text{fonction} \rangle | \langle \text{procedure} \rangle$
 $\langle \text{procedure} \rangle ::= \text{procedure } \langle \text{ident} \rangle \langle \text{partieFormelle} \rangle \text{ is } \langle \text{corpsProc} \rangle$
 $\langle \text{fonction} \rangle ::= \text{function } \langle \text{ident} \rangle \langle \text{partieFormelle} \rangle \text{ return } \langle \text{type} \rangle \text{ is } \langle \text{corpsFonct} \rangle$
 $\langle \text{corpsProc} \rangle ::= \langle \text{partieDeclaProc} \rangle \text{ begin } \langle \text{suiteInstr} \rangle \text{ end}$
 $\quad | \text{ begin } \langle \text{suiteInstr} \rangle \text{ end}$
 $\langle \text{corpsFonct} \rangle ::= \langle \text{partieDeclaProc} \rangle \text{ begin } \langle \text{suiteInstrNonVide} \rangle \text{ end}$
 $\quad | \text{ begin } \langle \text{suiteInstrNonVide} \rangle \text{ end}$
 $\langle \text{partieFormelle} \rangle ::= (\langle \text{listeSpecifFormelles} \rangle) | ()$
 $\langle \text{listeSpecifFormelles} \rangle ::= \langle \text{specif} \rangle ; \langle \text{listeSpecifFormelles} \rangle | \langle \text{specif} \rangle$
 $\langle \text{specif} \rangle ::= \langle \text{listeIdent} \rangle : \langle \text{mode} \rangle \langle \text{type} \rangle$
 $\quad | \langle \text{listeIdent} \rangle : \langle \text{type} \rangle$
 $\langle \text{mode} \rangle ::= \text{in} | \text{in out}$
 $\langle \text{type} \rangle ::= \text{integer} | \text{boolean}$
 $\langle \text{partieDeclaProc} \rangle ::= \langle \text{listeDeclaVar} \rangle$
 $\langle \text{listeDeclaVar} \rangle ::= \langle \text{declaVar} \rangle \langle \text{listeDeclaVar} \rangle | \langle \text{declaVar} \rangle$
 $\langle \text{declaVar} \rangle ::= \langle \text{listeIdent} \rangle : \langle \text{type} \rangle ;$
 $\langle \text{listeIdent} \rangle ::= \langle \text{ident} \rangle , \langle \text{listeIdent} \rangle | \langle \text{ident} \rangle$
 $\langle \text{suiteInstrNonVide} \rangle ::= \langle \text{instr} \rangle ; \langle \text{suiteInstrNonVide} \rangle | \langle \text{instr} \rangle$
 $\langle \text{suiteInstr} \rangle ::= \langle \text{suiteInstrNonVide} \rangle | \epsilon$
 $\langle \text{instr} \rangle ::= \langle \text{affectation} \rangle | \langle \text{boucle} \rangle | \langle \text{altern} \rangle | \langle \text{es} \rangle | \langle \text{retour} \rangle | \langle \text{appelProc} \rangle$
 $\langle \text{appelProc} \rangle ::= \langle \text{ident} \rangle (\langle \text{listePe} \rangle) | \langle \text{ident} \rangle ()$
 $\langle \text{listePe} \rangle ::= \langle \text{expressions} \rangle , \langle \text{listePe} \rangle | \langle \text{expression} \rangle$
 $\langle \text{affectation} \rangle ::= \langle \text{ident} \rangle := \langle \text{expression} \rangle$

$\langle expression \rangle ::= \langle expression \rangle \text{ or } \langle exp1 \rangle \mid \langle exp1 \rangle$
 $\langle exp1 \rangle ::= \langle exp1 \rangle \text{ and } \langle exp2 \rangle \mid \langle exp2 \rangle$
 $\langle exp2 \rangle ::= \langle exp2 \rangle \langle opRel \rangle \langle exp3 \rangle \mid \langle exp3 \rangle$
 $\langle opRel \rangle ::= = \mid \neq \mid < \mid \leq \mid > \mid \geq$
 $\langle exp3 \rangle ::= \langle exp3 \rangle \langle opAd \rangle \langle exp4 \rangle \mid \langle exp4 \rangle$
 $\langle opAd \rangle ::= + \mid -$
 $\langle exp4 \rangle ::= \langle exp4 \rangle \langle opMult \rangle \langle prim \rangle \mid \langle prim \rangle$
 $\langle opMult \rangle ::= * \mid /$
 $\langle prim \rangle ::= \langle opUnaire \rangle \langle elemPrim \rangle \mid \langle elemPrim \rangle$
 $\langle opUnaire \rangle ::= + \mid - \mid \text{not}$
 $\langle elemPrim \rangle ::= \langle valeur \rangle \mid (\langle expression \rangle) \mid \langle ident \rangle \mid \langle appelFonct \rangle$
 $\langle appelFonct \rangle ::= \langle ident \rangle (\langle listePe \rangle) \mid \langle ident \rangle ()$
 $\langle valeur \rangle ::= \langle entier \rangle \mid \langle valBool \rangle$
 $\langle valBool \rangle ::= \text{true} \mid \text{false}$
 $\langle es \rangle ::= \text{get} (\langle ident \rangle)$
 $\quad \mid \text{put} (\langle expression \rangle)$
 $\langle boucle \rangle ::= \text{while } \langle expression \rangle \text{ loop } \langle suiteInstr \rangle \text{ end}$
 $\langle altern \rangle ::= \text{if } \langle expression \rangle \text{ then } \langle suiteInstr \rangle \text{ end}$
 $\quad \mid \text{if } \langle expression \rangle \text{ then } \langle suiteInstr \rangle \text{ else } \langle suiteInstr \rangle \text{ end}$
 $\langle retour \rangle ::= \text{return } \langle expression \rangle$
 $\langle ident \rangle ::= \langle lettre \rangle \langle listeLettreOuChiffre \rangle \mid \langle lettre \rangle$
 $\langle listeLettreOuChiffre \rangle ::= \langle lettreOuChiffre \rangle \langle listeLettreOuChiffre \rangle \mid \langle lettreOuChiffre \rangle$
 $\langle lettreOuChiffre \rangle ::= \langle lettre \rangle \mid \langle chiffre \rangle$
 $\langle lettre \rangle ::= a \mid \dots \mid z \mid A \mid \dots \mid Z$
 $\langle chiffre \rangle ::= 0 \mid 1 \mid \dots \mid 9$
 $\langle entier \rangle ::= \langle chiffre \rangle \langle entier \rangle \mid \langle chiffre \rangle$

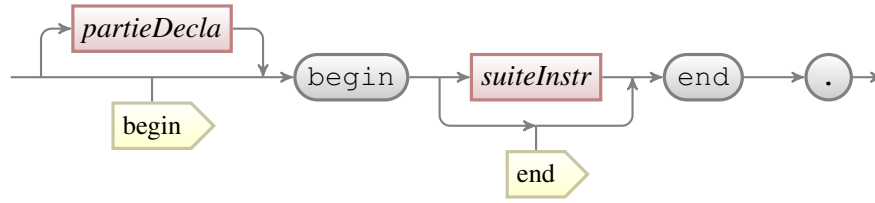
B Diagrammes syntaxiques

Cette partie décrit les différents diagrammes syntaxiques du langage. Ils correspondent à la traduction des règles de la grammaire pour NNP.

B.1 programme



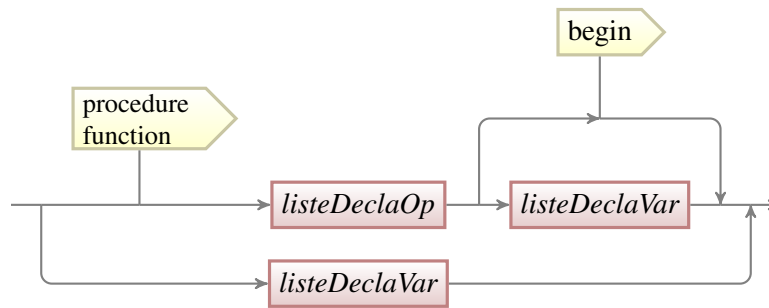
B.2 corpsProgPrinc



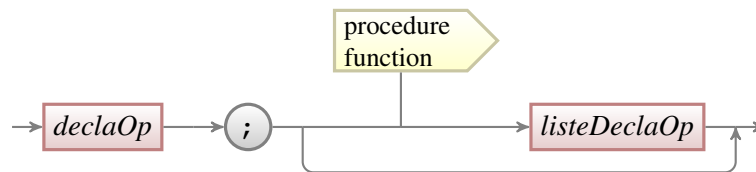
B.3 specifProgPrinc



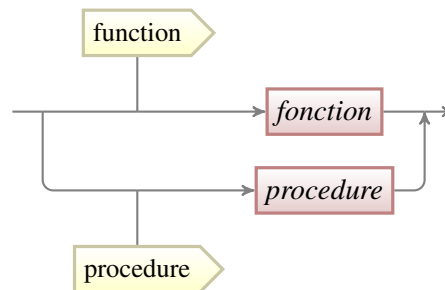
B.4 partieDecla



B.5 listeDeclaOp



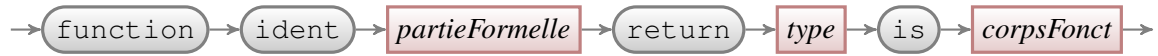
B.6 declaOp



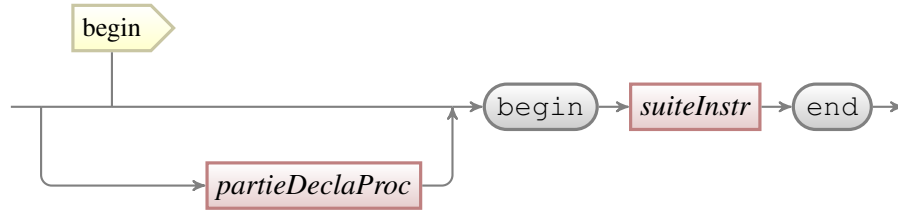
B.7 procedure



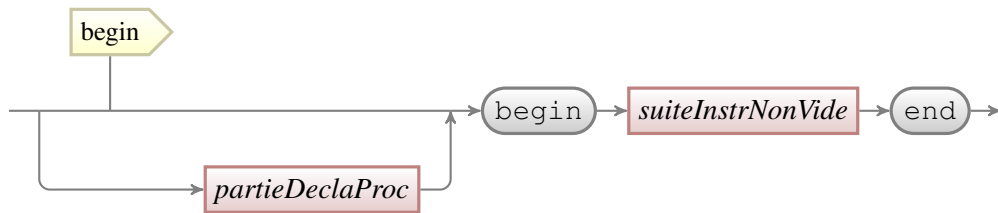
B.8 fonction



B.9 corpsProc



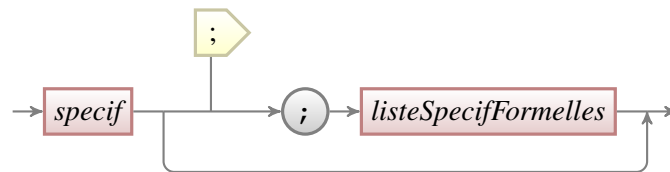
B.10 corpsFonct



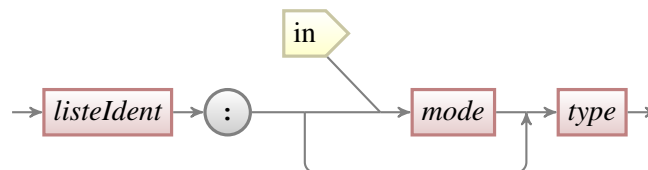
B.11 partieFormelle



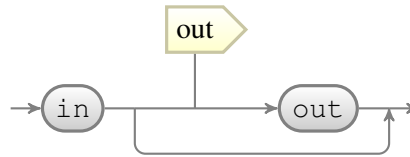
B.12 listeSpecifFormelle



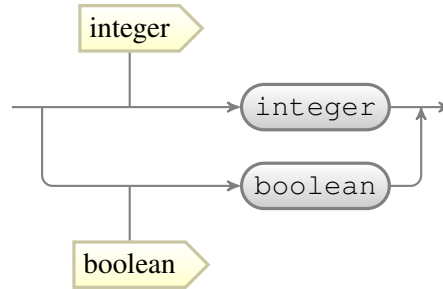
B.13 specif



B.14 mode



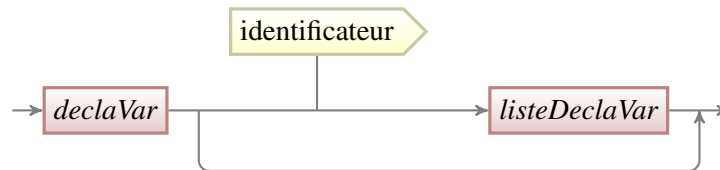
B.15 type



B.16 partieDeclaProc



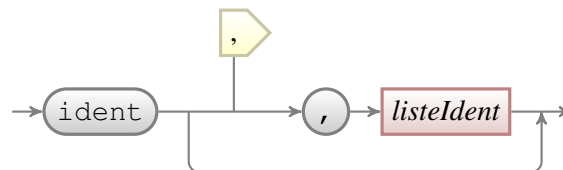
B.17 listeDeclaVar



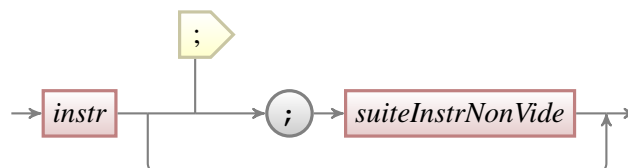
B.18 declaVar



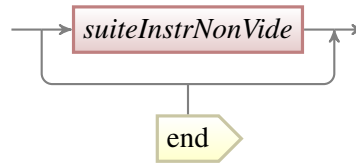
B.19 listeIdent



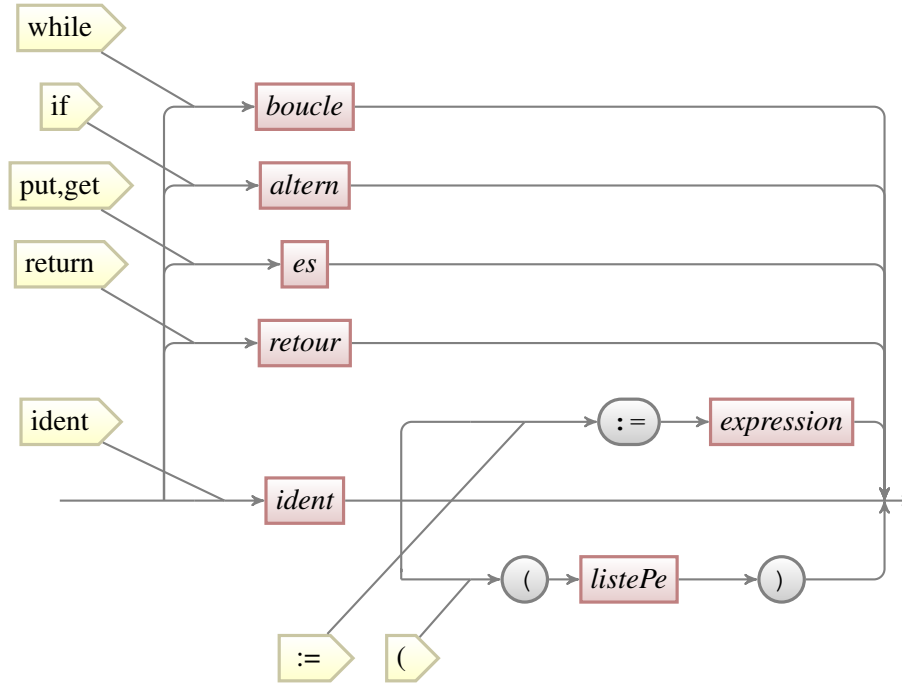
B.20 suiteInstrNonVide



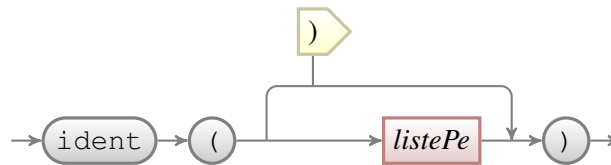
B.21 suiteInstr



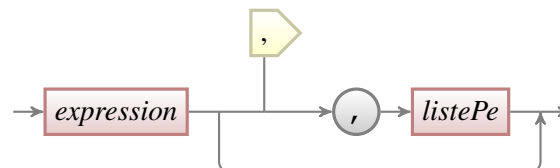
B.22 instr



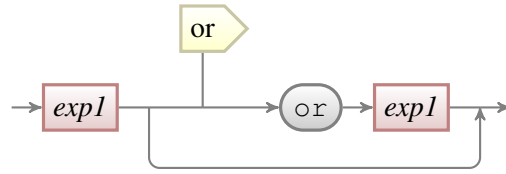
B.23 appelProc



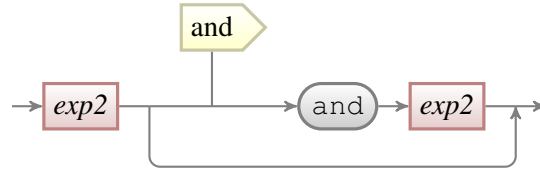
B.24 listePe



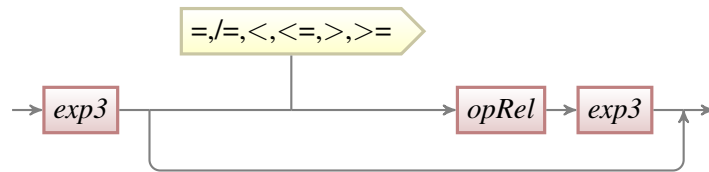
B.25 expression



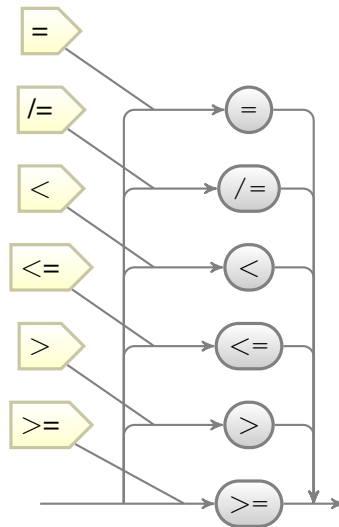
B.26 exp1



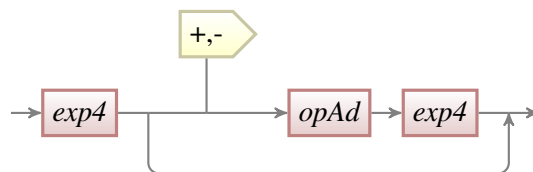
B.27 exp2



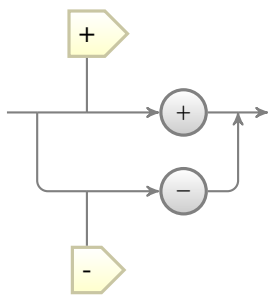
B.28 opRel



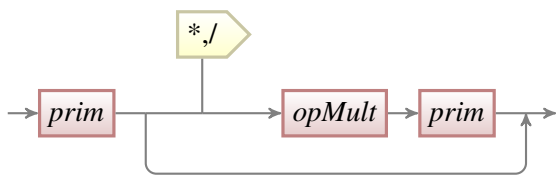
B.29 exp3



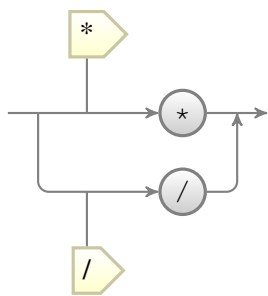
B.30 opAd



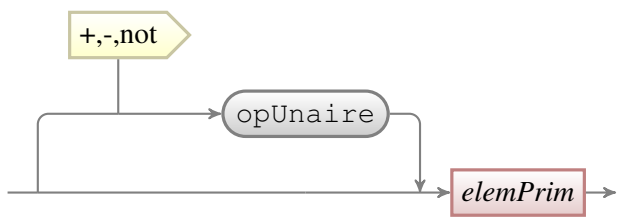
B.31 exp4



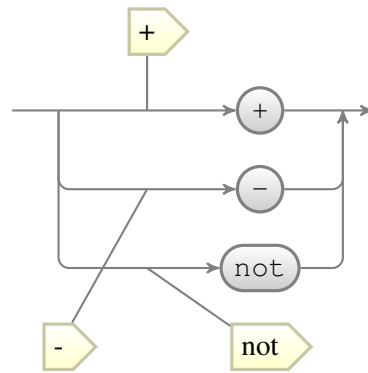
B.32 opMult



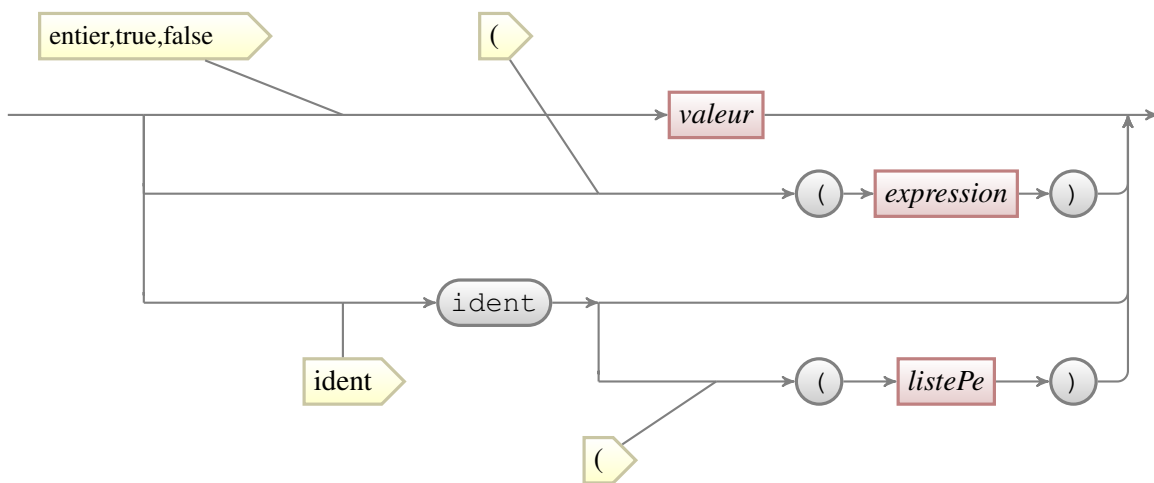
B.33 prim



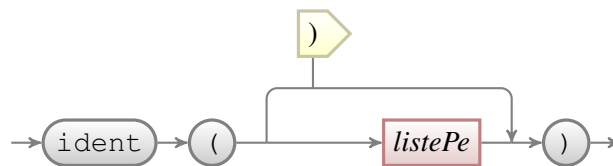
B.34 opUnaire



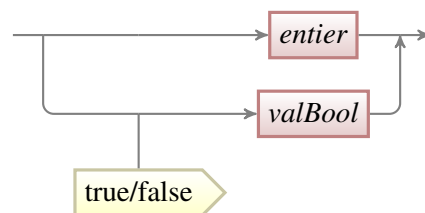
B.35 elemPrim



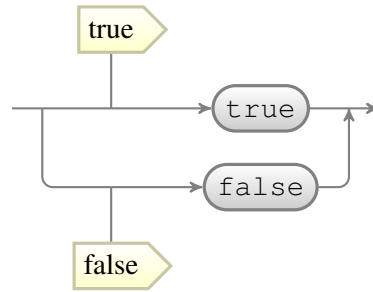
B.36 appelFonct



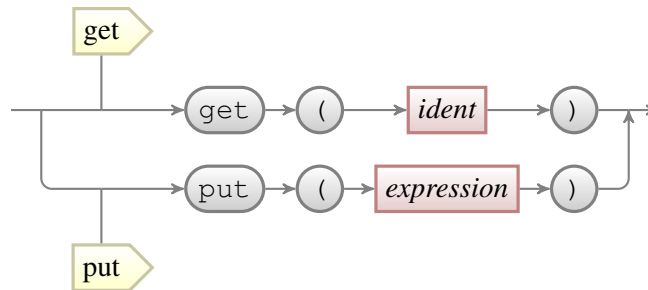
B.37 valeur



B.38 valBool



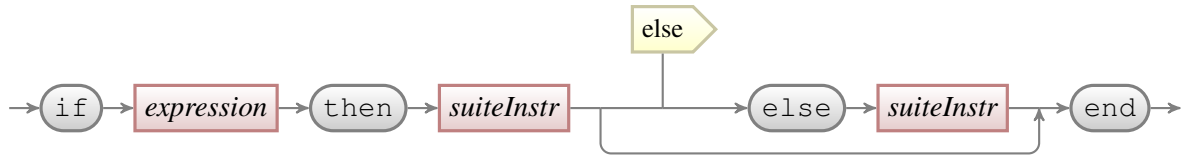
B.39 es



B.40 boucle



B.41 altern



B.42 retour

