# 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 : <=), 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.

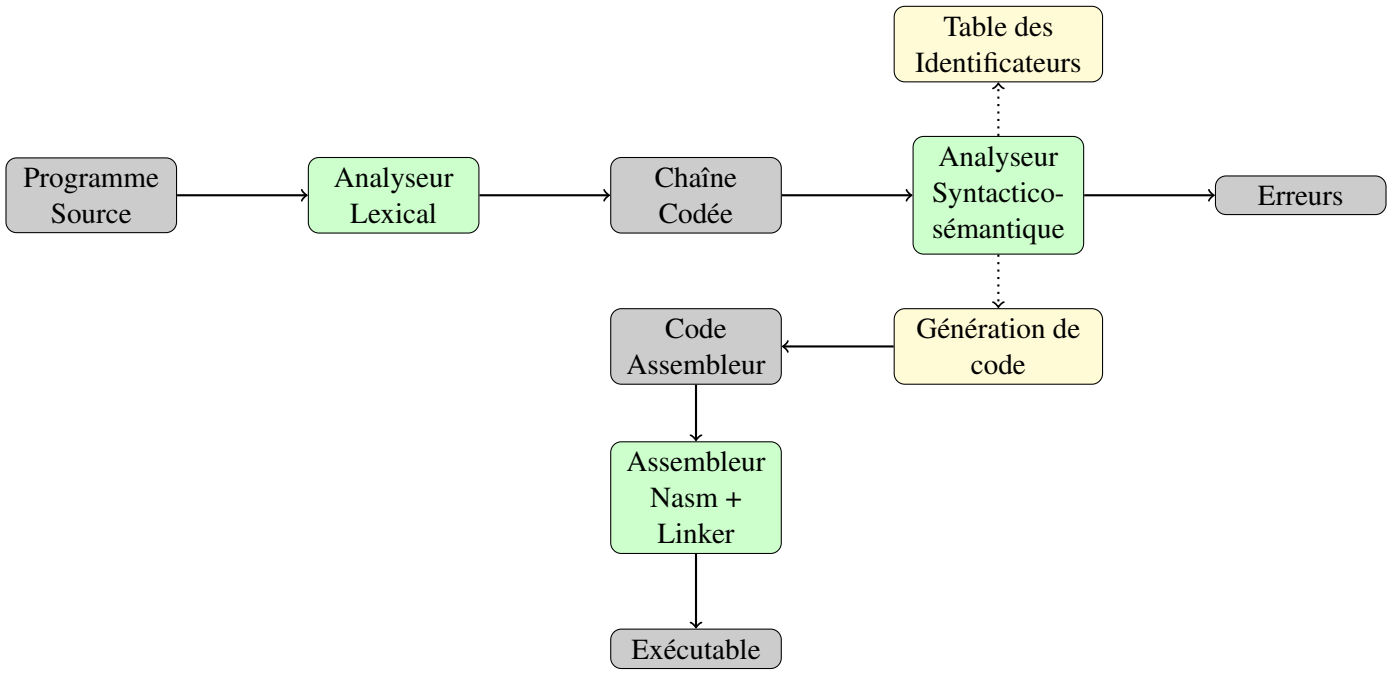



FIGURE 1 – Structure du compilateur NNP

# 2 Le langage à compiler

Cette section donne une 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'entré-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.

| 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.

- 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 considéré.
- symbole : certaines ensembles de caractères comme '<=' ou '>='.
- 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 : le langage cible choisit est le langage d'assemblage Nasm (cf annexe C) pour une architecture 32bits de type i386 (OS Linux). Le code généré devra donc être compilable directement par Nasm afin de générer un fichier exécutable. 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.
- 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).

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 ⟨*programme*⟩.

⟨*programme*⟩ : := ⟨*specifProgPrinc*⟩ `is` ⟨*corpsProgPrinc*⟩

⟨*corpsProgPrinc*⟩ : := ⟨*partieDecla*⟩ `begin` ⟨*suiteInstr*⟩ `end .`
| `begin` ⟨*suiteInstr*⟩ `end .`

⟨*specifProgPrinc*⟩ : := `procedure` ⟨*ident*⟩

⟨*partieDecla*⟩ : := ⟨*listeDeclaOp*⟩ ⟨*listeDeclaVar*⟩ | ⟨*listeDeclaVar*⟩ | ⟨*listeDeclaOp*⟩

⟨*listeDeclaOp*⟩ : := ⟨*declaOp*⟩ `;` ⟨*listeDeclaOp*⟩ | ⟨*declaOp*⟩ `;`

⟨*declaOp*⟩ : := ⟨*fonction*⟩ | ⟨*procedure*⟩

⟨*procedure*⟩ : := `procedure` ⟨*ident*⟩ ⟨*partieFormelle*⟩ `is` ⟨*corpsProc*⟩

⟨*fonction*⟩ : := `function` ⟨*ident*⟩ ⟨*partieFormelle*⟩ `return` ⟨*type*⟩ `is` ⟨*corpsFonct*⟩

⟨*corpsProc*⟩ : := ⟨*partieDeclaProc*⟩ `begin` ⟨*suiteInstr*⟩ `end`
| `begin` ⟨*suiteInstr*⟩ `end`

⟨*corpsFonct*⟩ : := ⟨*partieDeclaProc*⟩ `begin` ⟨*suiteInstrNonVide*⟩ `end`
| `begin` ⟨*suiteInstrNonVide*⟩ `end`

⟨*partieFormelle*⟩ : := `(` ⟨*listeSpecifFormelles*⟩ `)` | `( )`

⟨*listeSpecifFormelles*⟩ : := ⟨*specif*⟩ `;` ⟨*listeSpecifFormelles*⟩ | ⟨*specif*⟩

⟨*specif*⟩ : := ⟨*listeIdent*⟩ `:` ⟨*mode*⟩ ⟨*type*⟩
| ⟨*listeIdent*⟩ `:` ⟨*type*⟩

⟨*mode*⟩ : := `in` | `in out`

⟨*type*⟩ : := `integer` | `boolean`

⟨*partieDeclaProc*⟩ : := ⟨*listeDeclaVar*⟩

⟨*listeDeclaVar*⟩ : := ⟨*declaVar*⟩ ⟨*listeDeclaVar*⟩ | ⟨*declaVar*⟩

⟨*declaVar*⟩ : := ⟨*listeIdent*⟩ `:` ⟨*type*⟩ `;`

⟨*listeIdent*⟩ : := ⟨*ident*⟩ `,` ⟨*listeIdent*⟩ | ⟨*ident*⟩

⟨*suiteInstrNonVide*⟩ : := ⟨*instr*⟩ `;` ⟨*suiteInstrNonVide*⟩ |⟨*instr*⟩

⟨*suiteInstr*⟩ : := ⟨*suiteInstrNonVide*⟩ | $\epsilon$

⟨*instr*⟩ : := ⟨*affectation*⟩ | ⟨*boucle*⟩ | ⟨*altern*⟩ | ⟨*es*⟩ | ⟨*retour*⟩ | ⟨*appelProc*⟩

⟨*appelProc*⟩ : := ⟨*ident*⟩ `(` ⟨*listePe*⟩ `)` | ⟨*ident*⟩ `( )`

⟨*listePe*⟩ : := ⟨*expressions*⟩ `,` ⟨*listePe*⟩ | ⟨*expression*⟩

⟨*affectation*⟩ : := ⟨*ident*⟩ `:=` ⟨*expression*⟩

⟨*expression*⟩ : := ⟨*expression*⟩ `or` ⟨*exp1*⟩ | ⟨*exp1*⟩

⟨*exp1*⟩ : := ⟨*exp1*⟩ `and` ⟨*exp2*⟩ | ⟨*exp2*⟩

⟨*exp2*⟩ : := ⟨*exp2*⟩ ⟨*opRel*⟩ ⟨*exp3*⟩ | ⟨*exp3*⟩

⟨*opRel*⟩ : := = | /= | < | <= | > | >=

⟨*exp3*⟩ : := ⟨*exp3*⟩ ⟨*opAd*⟩ ⟨*exp4*⟩ | ⟨*exp4*⟩

⟨*opAd*⟩ : := + | −

⟨*exp4*⟩ : := ⟨*exp4*⟩ ⟨*opMult*⟩ ⟨*prim*⟩ | ⟨*prim*⟩

⟨*opMult*⟩ : := ⋆ | /

⟨*prim*⟩ : := ⟨*opUnaire*⟩ ⟨*elemPrim*⟩ | ⟨*elemPrim*⟩

⟨*opUnaire*⟩ : := + | − | `not`

⟨*elemPrim*⟩ : := ⟨*valeur*⟩ | `(` ⟨*expression*⟩ `)` | ⟨*ident*⟩ | ⟨*appelFonct*⟩

⟨*appelFonct*⟩ : := ⟨*ident*⟩ `(` ⟨*listePe*⟩ `)` | ⟨*ident*⟩ `(` `)`

⟨*valeur*⟩ : := ⟨*entier*⟩ | ⟨*valBool*⟩

⟨*valBool*⟩ : := `true` | `false`

⟨*es*⟩ : := `get (` ⟨*ident*⟩ `)`
| `put (` ⟨*expression*⟩ `)`

⟨*boucle*⟩ : := `while` ⟨*expression*⟩ `loop` ⟨*suiteInstr*⟩ `end`

⟨*altern*⟩ : := `if` ⟨*expression*⟩ `then` ⟨*suiteInstr*⟩ `end`
| `if` ⟨*expression*⟩ `then` ⟨*suiteInstr*⟩ `else` ⟨*suiteInstr*⟩ `end`

⟨*retour*⟩ : := `return` ⟨*expression*⟩

⟨*ident*⟩ : := ⟨*lettre*⟩ ⟨*listeLettreOuChiffre*⟩ | ⟨*lettre*⟩

⟨*listeLettreOuChiffre*⟩ : := ⟨*lettreOuChiffre*⟩ ⟨*listeLettreOuChiffre*⟩ | ⟨*lettreOuChiffre*⟩

⟨*lettreOuChiffre*⟩ : := ⟨*lettre*⟩ | ⟨*chiffre*⟩

⟨*lettre*⟩ : := a | . . . | z | A | . . . | Z

⟨*chiffre*⟩ : := 0 | 1 | . . . | 9

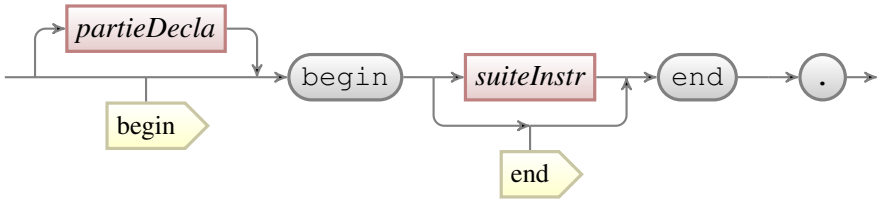
⟨*entier*⟩ : := ⟨*chiffre*⟩ ⟨*entier*⟩ | ⟨*chiffre*⟩

# B Diagrammes syntaxiques

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

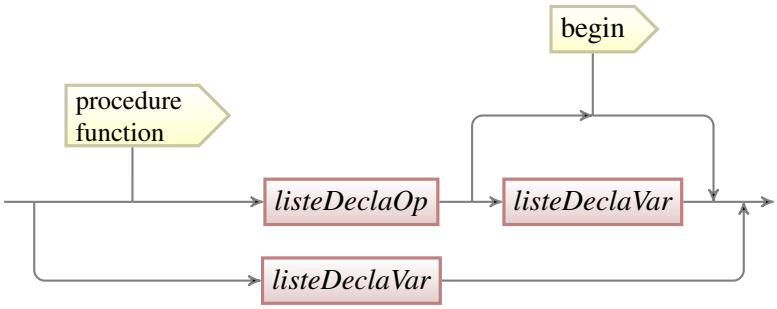
## B.1 programme

### B.2 corpsProgPrinc

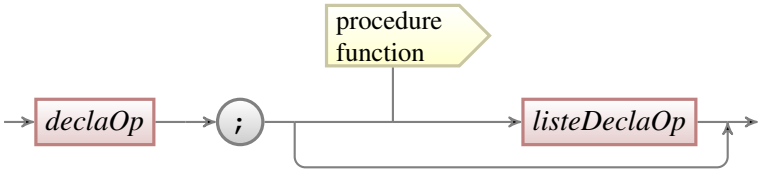



### B.3 specifProgPrinc

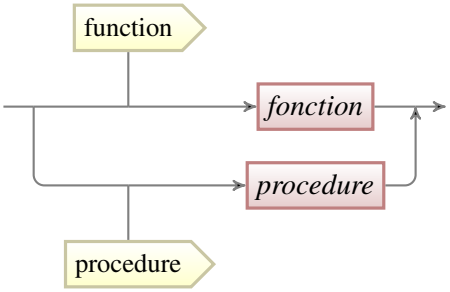

### B.4 partieDecla




### B.5 listeDeclaOp

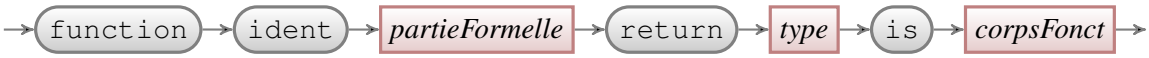



### B.6 declaOp

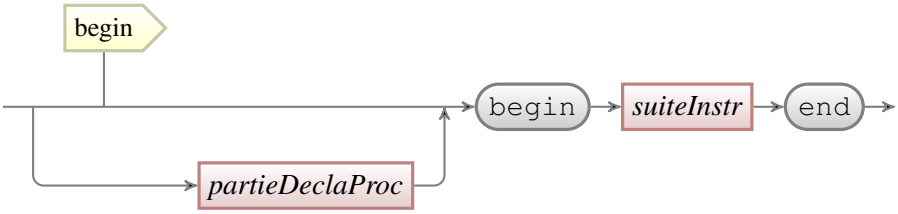



### B.7 procedure

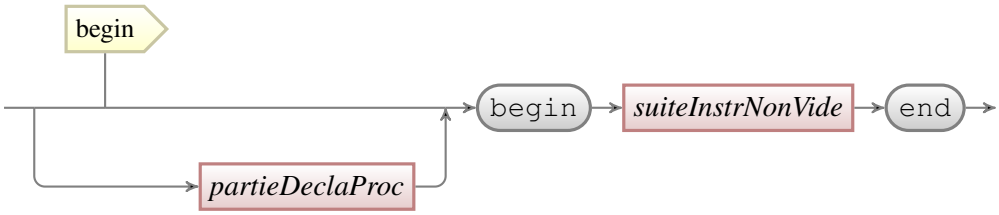
### B.8 fonction




### B.9 corpsProc

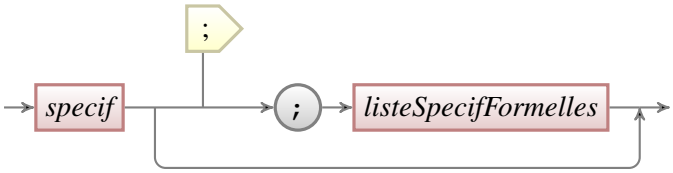



### B.10 corpsFonct

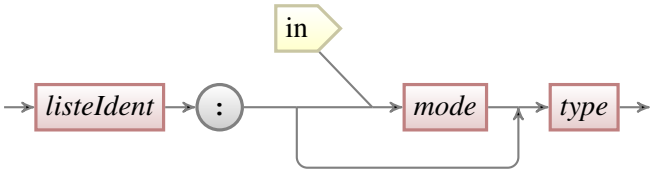



### B.11 partieFormelle

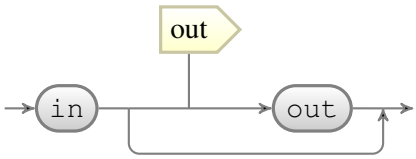

### B.12 listeSpecifFormelle



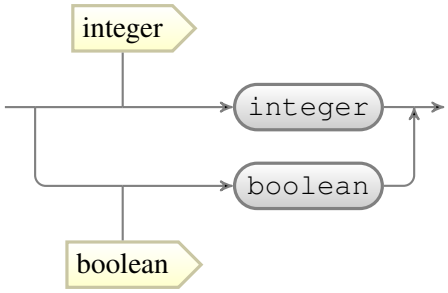

### B.13 specif



### B.14 mode

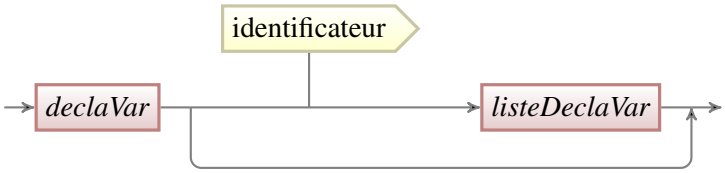



### B.15 type

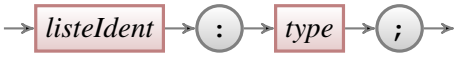



### B.16 partieDeclaProc

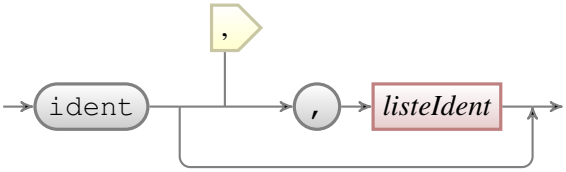

### B.17 listeDeclaVar



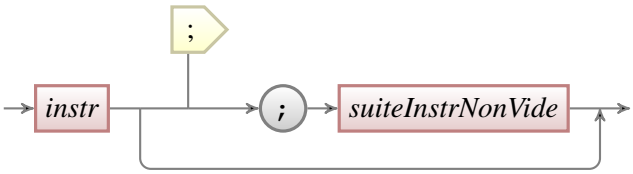

### B.18 declaVar



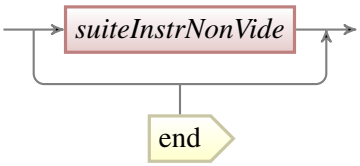

### B.19 listeIdent



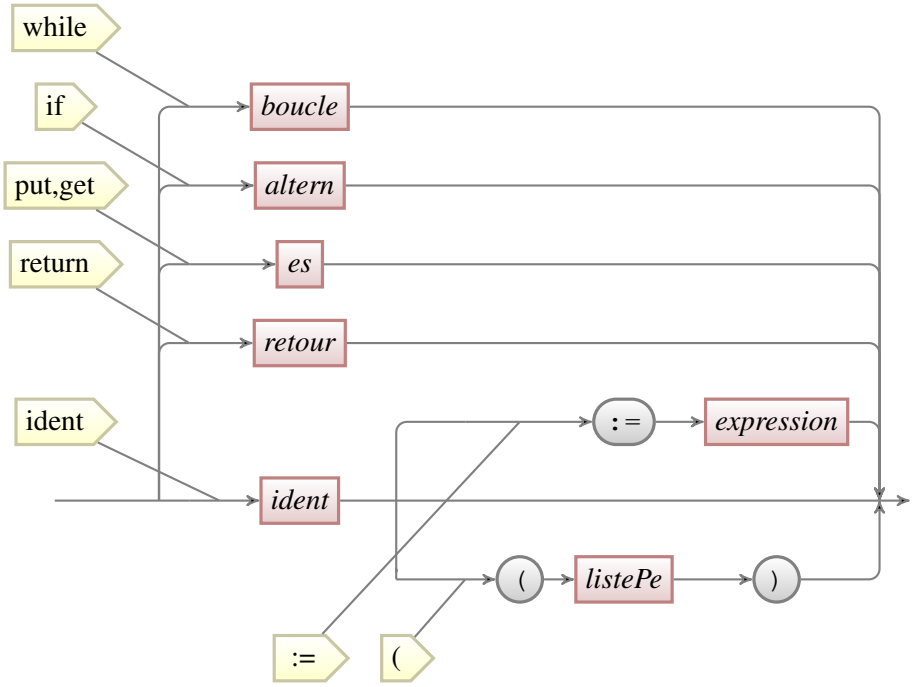

### B.20 suiteInstrNonVide



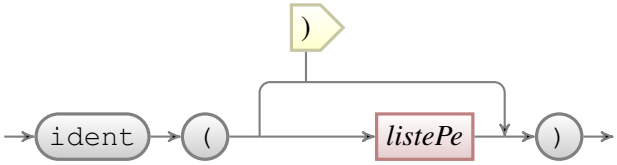
### B.21 suiteInstr



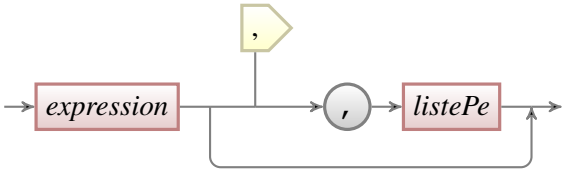

### B.22 instr



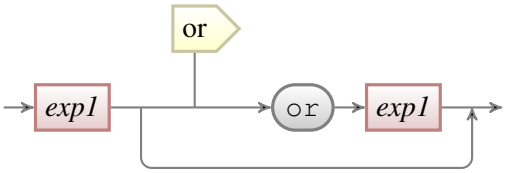

### B.23 appelProc



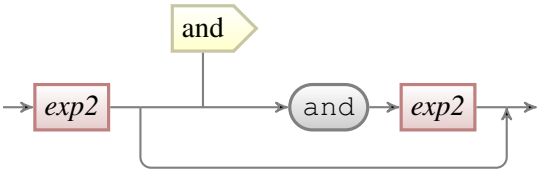

### B.24 listePe



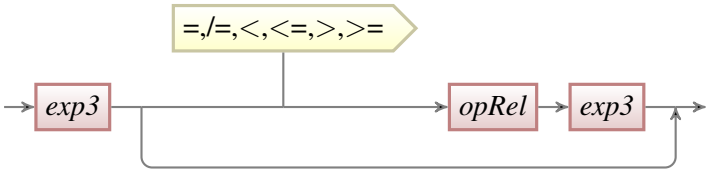
### B.25 expression



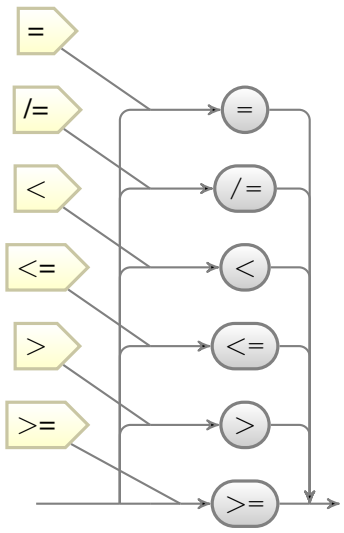

### B.26 exp1



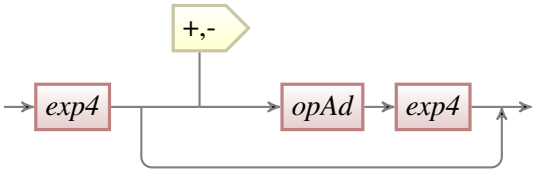

### B.27 exp2



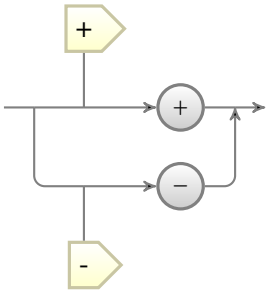

### B.28 opRel



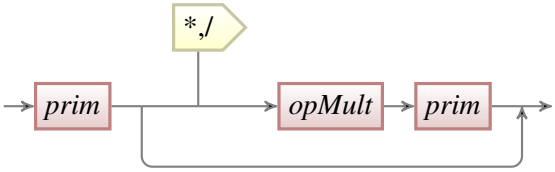

### B.29 exp3



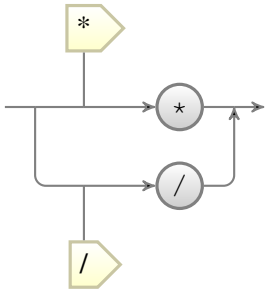
### B.30 opAd



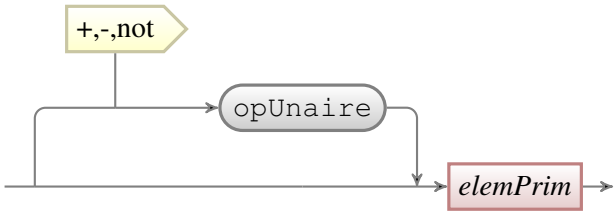

### B.31 exp4



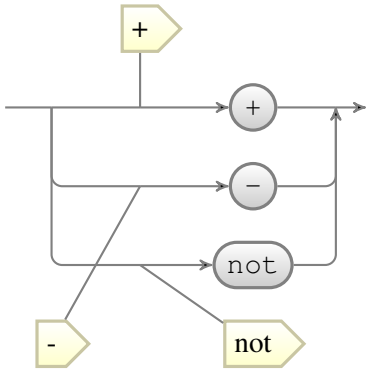

### B.32 opMult



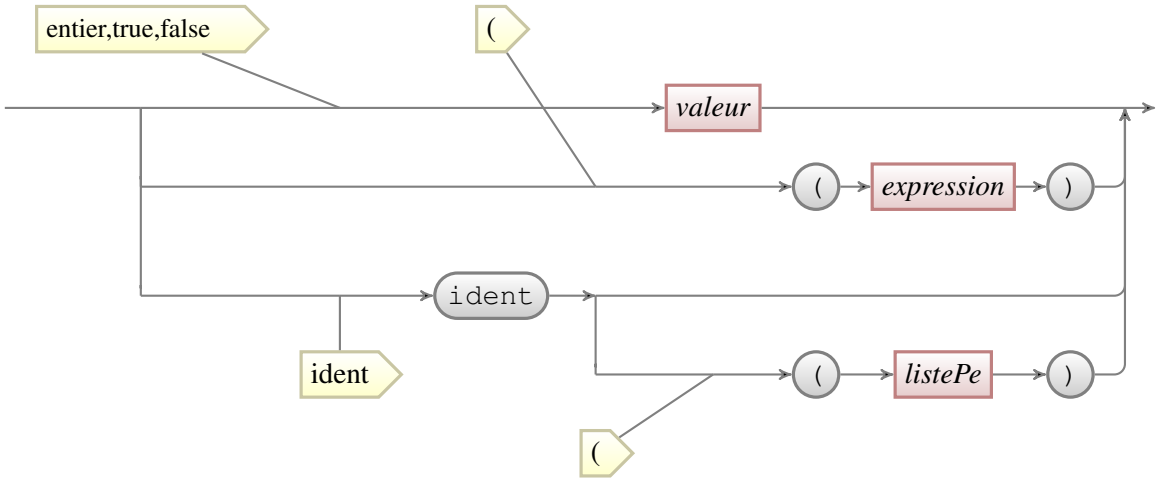

### B.33 prim



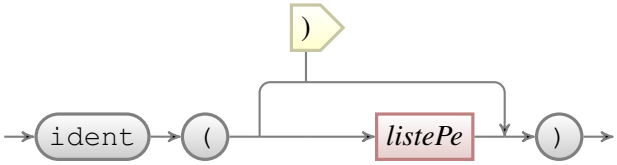
### B.34 opUnaire



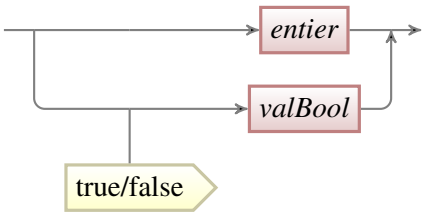

### B.35 elemPrim



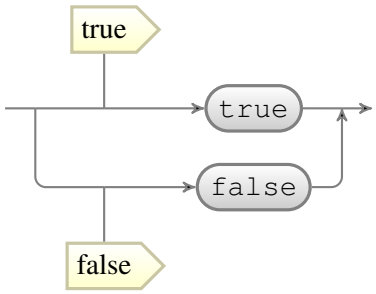

### B.36 appelFonct



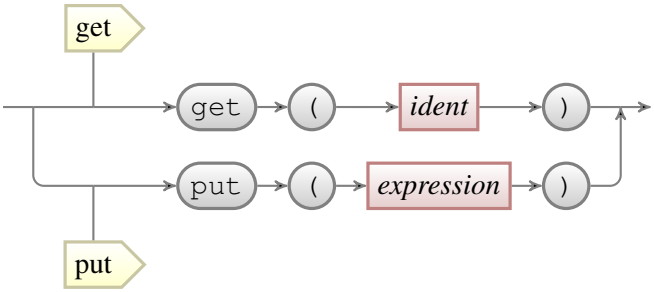

### B.37 valeur



### B.38 valBool

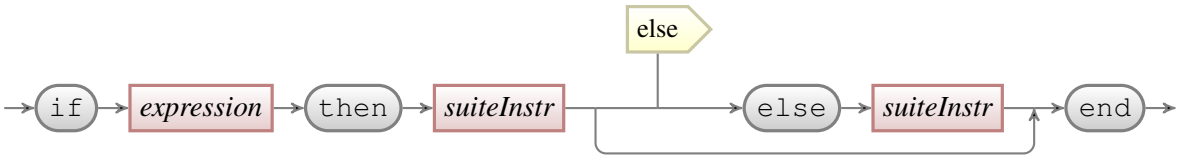



### B.39 es




### B.40 boucle


### B.41 altern




### B.42 retour


# C Exemple de code assembleur (Nasm)

## C.1 HelloWorld avec Nasm

L'exemple suivant montre la réalisation de l'exemple classique *Hello World !* en utilisant l'appel à la fonction *printf*.

```
01 : global main
02 : extern printf
03 :
04 : segment .data
05 : str: db 'Hello World!',10,0
```

```
06 :
08 : segment .text
09 :
10 : main:
11 :    push  dword str
12 :    call  printf
13 :    add   esp,4
14 :
15 :    mov   eax,1
16 :    mov   ebx,0
17 :    int   80h
```

L'appel à *printf* est réalisé aux lignes 11-13. La première ligne permet d'empiler l'adresse de la chaîne de caractère du format que l'on souhaite afficher. Ensuite, l'appel à la routine printf est réalisé et enfin, le pointeur de pile est restoré (compense le paramètre empilé avant l'appel à printf).

Les lignes 15-17 permettent la terminaison du programme en retournant le code 0. La valeur 1 placée dans EAX indique une demande de fin de programme. Le code de retour est lui placé dans EBX (ici 0). Enfin, une interruption du noyau est déclenchée.

## C.2 Compilation et exécution de l'exemple

La compilation et l'exécution de ce programme passe par l'exécution des commandes suivantes :

```
nasm -f elf32 helloworld.asm
gcc -m32 -o helloworld helloworld.o
./helloworld
```

## C.3 Quelques liens

- Un tutoriel pour débuter : `http://docs.cs.up.ac.za/programming/asm/derick_tut/`
- Nasm Quick Reference : `http://www.cs.uaf.edu/2006/fall/cs301/support/x86/`
- Quelques exemples de code (notamment avec printf) : `http://www.csee.umbc.edu/portal/help/nasm/sample.shtml`
- D'autres exemples (assez complet) : `http://cs.lmu.edu/~ray/notes/nasmexamples/`
- Un cours sur l'assembleur : `http://www.drpaulcarter.com/pcasm/`