
Théorie des langages

& outils pour la compilation

ESGI – 3 AL 2024/ 2025

Objectifs du cours

Le cours vise à

- créer son propre langage
- coder un interpréteur capable d'exécuter son code

Architecture d'un compilateur

La compilation est une transformation qui consiste à rendre un programme exécutable. Fondamentalement, il s'agit d'une traduction : le texte initial (le programme) est transformé en un texte compréhensible par la machine (dans un formalisme de plus bas niveau).

La sortie d'un compilateur est de nature très variable : un programme exécutable par un processeur physique (Pentium, G7), un fichier de code pour une machine virtuelle (JVM), un code abstrait destiné à être recompilé, ...

Le travail du compilateur se fait usuellement en plusieurs phases :

1. Analyse lexicale

Les caractères isolés du texte initial sont regroupés en unités lexicales appartenant au langage. La tâche de l'analyse lexicale est déléguée à celle de l'analyse lexicale en fournissant un mot lors de chaque appel.

2. Analyse syntaxique

La bonne formation structurelle de la suite d'unités lexicales est vérifiée relativement à la grammaire du langage.

3. Analyse sémantique

Les propriétés sémantiques à vérifier : la bonne déclaration des identificateurs, la vérification des types dans les opérandes relativement aux opérateurs, le besoin de conversions, le type et le nombre des arguments dans les fonctions, ...

4. Génération de code intermédiaire

un peu comme avec les machines abstraites, pour mutualiser les dernières étapes de la compilation avec des compilateurs d'autres langages source.

5. Optimisation de code

Consiste à transformer le code pour une exécution plus rapide.

6. Génération du code final

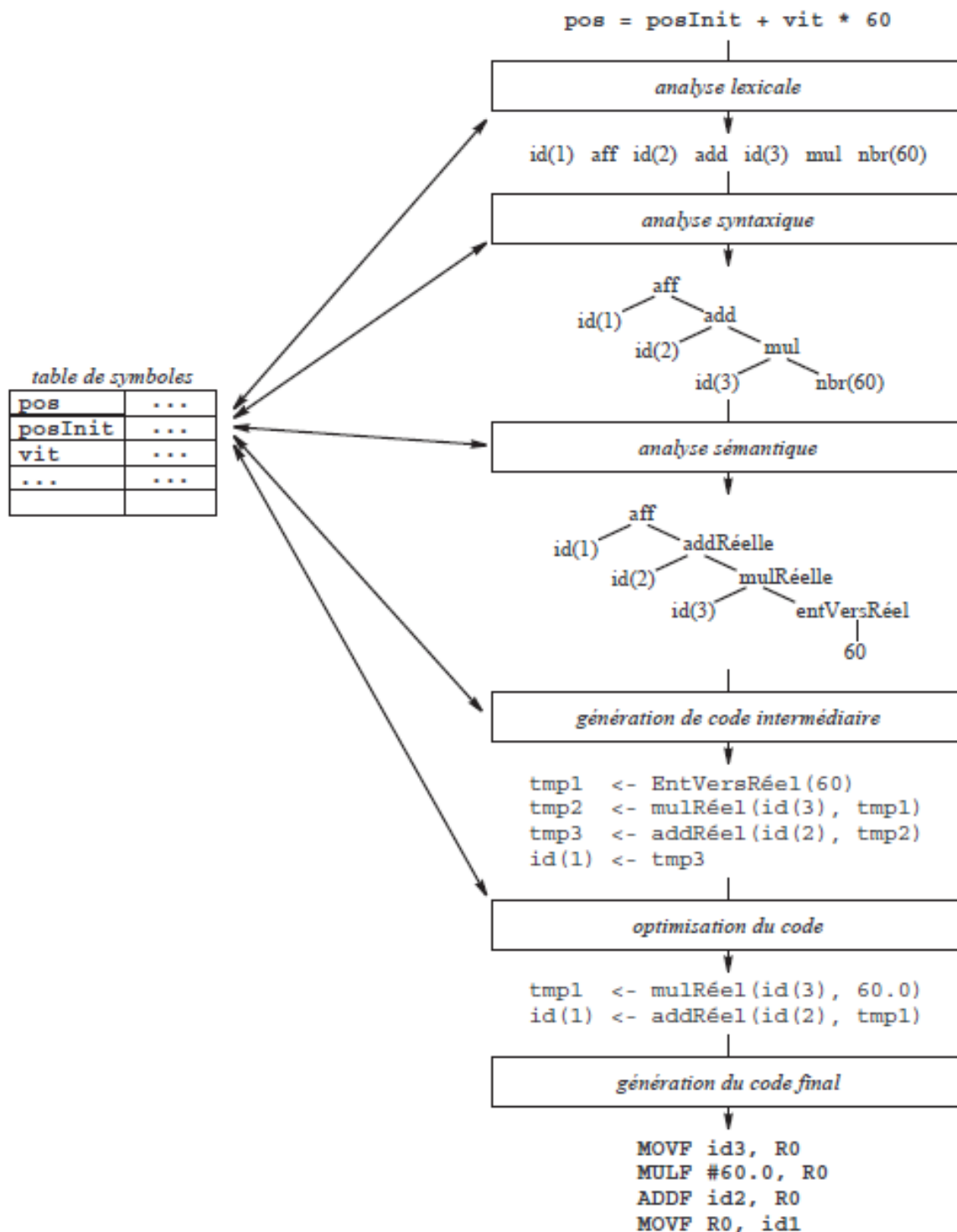
Nécessite la connaissance de la machine cible (réelle, virtuelle, ou abstraite), et notamment de ses possibilités en matières de registre, de pile, ...

Dans ce cours, notre étude se focalisera sur les 2 premières phases : l'analyse lexicale et l'analyse syntaxique. Au cours du projet de fin d'année, vous vous confronterez (de façon élémentaire) aux autres phases : il s'agira de définir votre propre langage de programmation, et évidemment, d'en concevoir un interpréteur permettant de l'opérationnaliser.

1. **Analyse lexicale.** Après avoir présenté les notions de bases sur les langages formels, on s'intéressera aux automates finis, une représentation « opérationnelle » des expressions régulières qui facilite significativement la modélisation du test d'appartenance d'une chaîne, ou l'énumération des chaînes couvertes par une expression régulière. Ensuite, nous

étudierons les différentes tâches mises en jeu dans un analyseur lexical, ainsi que l'utilitaire *lex*, un outil permettant de générer des analyseurs lexicaux.

2. **Analyse syntaxique.** Après avoir présenté les concepts de base des grammaires formelles (arbre de dérivation, automates à piles), on s'intéressera à leur capacité à modéliser de manière avantageuse les langages et à produire une analyse des chaînes d'entrée (les programmes dans le cas des langages informatiques). Ensuite, on verra les principes de l'analyse descendante LR(k) et le fonctionnement de *yacc*, un générateur d'analyseur syntaxique.



Objectifs du cours.....	2
Architecture d'un compilateur	2
1. Alphabets et langages	5
2. Les regex : une brève introduction.....	7
3. Grammaire : l'exemple du langage naturel.....	10
4. Langages et grammaires.....	12
5. Classification des grammaires : Hiérarchie de Chomsky.....	19
6. Analyse syntaxique ascendante	22
TP PLY1	31
TP PLY2	37
TP PLY3	44

Séances

1. Expressions régulières, Langages, Grammaires
2. Problème de reconnaissance d'un mot.
Arbre de dérivation
Analyse ascendante par décalage réduction et conflits
3. TP PLY1 (Yacc en python) : prise en main
4. TP PLY2 : Arbre de syntaxe abstrait + évaluation
5. TP PLY3 : Implémentation des fonctions
6. Algorithmes LR(0) et LR(1)
7. Pré soutenance projet

Frédéric Baudoin

baudoin2020@gmail.com

1. Alphabets et langages

Alphabet

On appelle alphabet un ensemble de symboles

- il ne peut pas y avoir un nombre infini de symboles
- on note souvent cet ensemble Σ ou V_T
- les symboles sont les éléments de base d'un langage

Chaînes

Une chaîne est une suite finie de symboles concaténés

Langage

Un langage L sur Σ est une partie de toutes les chaînes possibles (noté Σ^*) d'éléments de Σ

- C'est donc certaines séquences de symboles de Σ
- En pratique, les langages comprennent souvent un nombre **infini** de chaînes

Exemples de langages

Exemple 1

$$\Sigma = \{a\}$$

$$L = \{a, aa, aaa, aaaa, \dots\} = \{a, a^2, a^3, a^4, \dots\} = \{a^k, k > 0\}$$

Exemple 2

$$\Sigma = \{a, b\}$$

$$L = \{ab, aab, aaab, aaaab, \dots\} = \{ab, a^2b, a^3b, a^4b, \dots\} = \{a^k b, k > 0\}$$

Exemple 3

$$\Sigma = \{a, b, c\}$$

$$\Sigma^* = \text{l'ensemble des séquences de symboles } \in \{a, b, c\}$$

$$L = \{\text{mots de } \Sigma^* \text{ commençant par } a\}$$

Exemple 4 : Calculatrice

$$\Sigma = \{0, 1, 2, \dots, 9, +, -, \times, /\}$$

L : ensemble des expressions arithmétiques syntaxiquement correctes

Exemple 5 : cas du langage naturel

Σ : tous les mots possibles = ceux du dictionnaire + verbes conjugués + féminins + pluriels + espaces + ponctuation

L : ensemble des phrases (ou des textes) syntaxiquement bien formées

Exemple 6 : cas des langages de programmation – version 1

Σ : les identifiants, les chiffres, les opérateurs arithmétiques, les mots clefs réservés du langage

L : ensembles des programmes compilables

Exemple 7: cas des langages de programmation – version 2

Σ : les caractères

L : ensembles des programmes compilables

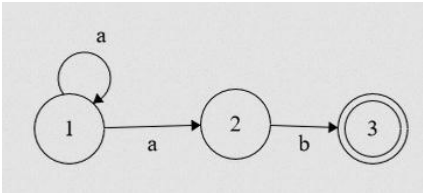
Exemple 8: langage des livres de 200 pages

Σ : les caractères

L : ensembles des livres de 200 pages

Représentation des langages

Comment spécifier un ensemble de chaînes ?

Avec des mots	$L = \{\text{mots sur } \{a, b, c\} \text{ commençant par } a\}$
Avec des ...	$L = \{ab, aab, aaab, aaaab, \dots\} = \{ab, a^2b, a^3b, a^4b, \dots\}$
Avec une propriété mathématique	$L = \{a^k b, k > 0\}$
Avec une regex	$L = a^+b$
Avec une grammaire	$S \rightarrow Ab$ $A \rightarrow Aa$ $A \rightarrow a$
Avec des automates finis	 <pre> graph LR start(()) --> 1((1)) 1 -- a --> 1 1 -- a --> 2((2)) 2 -- b --> 3(((3))) </pre>

2. Les regex : une brève introduction

Les caractères usuels et le joker.

Les caractères représentent avant tout eux même : la regex /chaîne/ match avec "chaîne"

Classe de caractères : Une classe de caractères désigne un ensemble de caractères. Elle peut se définir de plusieurs manières :

- avec la liste ou les plages de caractères en faisant partie « [abcdefgXZRSTU] » ou « [a-gXZR-U] » pour les minuscules de a à g, les majuscules entre R et U, le X ou le Z
- avec la liste ou les plages de caractères n'en faisant pas partie. « [^abcdefgXZRSTU] » ou « [^a-gXZR-U] » pour tout caractère sauf les minuscules de a à g, les majuscules entre R et U, le X et le Z

Joker : Suivant les environnements, le saut de ligne est, ou pas, inclus dans les caractères couverts par le joker.

Exercice : A quelle regex correspond n'importe quel nombre à 2 chiffres.

solution : <https://regex101.com/r/lsu2Hp/1>

Exercice : A quelle regex correspond les nombres entre 10 et 49

L'échappement

Lorsqu'on souhaite utiliser un caractère spécial en tant que tel, il faut le faire précéder d'un caractère dit *d'échappement* : « \ ». Par exemple, « \. » pour le point, ou « \/ » pour le slash

Exercice : A quelle regex correspond les dates du type "14.10.2004"

Alternatives : Fromage|Dessert ?

Pour exprimer un choix entre plusieurs possibilités, on utilise le caractère « | », ce qui correspond au OU logique.

Par exemple, voici la regex des nombres entiers formés de 1, 2 ou 3 chiffres :

/[0-9] | [0-9] [0-9] | [0-9] [0-9] [0-9]/

Exercice : A quelle regex correspond les nombres réels de la forme "1.2", "3." ou ".4" dans lesquels il n'y a pas plus d'un chiffre de part de d'autre du point.

Le groupage : Jean-(Paul|Pierre)

La regex /creme brulée|crème au caramel|creme au chocolat/ est équivalente à /creme (brulée | au (caramel | chocolat)) /

On peut ainsi réécrire la regex des nombres entiers formés de 1, 2 ou 3 chiffres :

/[0-9](| [0-9](| [0-9])) /

Zero, un, beaucoup

Les quantificateurs ont été créés pour faciliter la reconnaissance d'éléments dont on ne connaît pas la longueur. Ces constructions se placent immédiatement après l'élément que l'on veut répéter.

Au moins une fois « + » : Répète l'élément qui le précède au moins une fois

Exercice : A quelle regex correspond l'ensemble des fractions, comme "43/455" ou "3/56" ?

Exercice : A quelle regex correspond les mots composés, non accentués, comme "porte-clefs" ?

Exercice : A quelle regex correspond les expressions additives de nombres entiers positifs, comme "1+2" ou "34+3445+5277+7355+0" ?

Au plus une fois « ? » : Répète l'élément qui le précède 0 ou une fois

Zero ou plus « * » : Répète l'élément qui le précède 0 ou plusieurs fois.

Récapitulatif

Appellation	syntaxe regex	matching
classe de caractères	[a-dk1-9]	tous les caractères entre a et d, ou le k, ou tous les caractères entre le 1 et le 9
le joker	.	n'importe quel caractère (sauf un saut de ligne)
le OU	a b	a ou b
quantificateur *	a*	Répète l'élément qui le précède 0 ou plusieurs fois.
quantificateur +	a+	Répète l'élément qui le précède 1 ou plusieurs fois.
la puissance	a{n}	Répète l'élément qui le précède n fois.
les caractères échappés	\. \ (\) * \+	. () * +

Priorité des opérateurs de regex et besoin de parenthésage

Du plus au moins prioritaire : les quantificateurs, la concaténation, le OU

Par exemple (ab)+ = {ab, abab, ababab, ...} alors que

ab+ = {ab, abb, abbb}

Exercice Regex

1. Ecrire la regex des noms de variables (un mot composé d'une suite de chiffre, lettre ou "_", ne commençant pas par un chiffre)
2. Ecrire la regex des numero de téléphone portable
3. Ecrire la regex des nombre entiers sans 0 inutile (10 mais pas 010)
4. Ecrire la regex des nombre décimaux sans 0 inutile (10.2 mais pas 010.2 ni 10.20, 2.0 ou 0.2 mais pas 2. ni .2)
5. Ecrire la regex des adresse email
6. Ecrire la regex des expressions additives d'entiers
 - a. V1 : 12+6+3+44 mais aussi 12
 - b. V2 sans les expressions sans opérateur (sans le 12)

3. Grammaire : l'exemple du langage naturel

Alphabet et règles de grammaire

On cherche à vérifier si la syntaxe de la phrase suivante est conforme à la grammaire de la langue française.

P = *le vieux chat attrape le petit rat*

Attention, ici, l'alphabet va désigner, non pas les 26 lettres de la langue française, mais l'ensemble de tous les mots existants = un dictionnaire + des verbes conjugués, des formes plurielles et féminines des noms et adjectifs) :

$$\Sigma = \{le, vieux, petit, chat, rat, attrape\}$$

On dispose des règles de grammaire suivante :

- | | |
|---|----------------------|
| 1. PHRASE \rightarrow GROUPE SUJET VERBE GROUPE COMPLÉMENT OBJET | } Règles syntaxiques |
| 2. GROUPE SUJET \rightarrow GROUPE NOMINAL | |
| 3. GROUPE COMPLÉMENT OBJET \rightarrow GROUPE NOMINAL | |
| 4. GROUPE NOMINAL \rightarrow ARTICLE NOM ADJECTIF ARTICLE ADJECTIF NOM | |
| 5. ARTICLE \rightarrow <i>le</i> | } Règles lexicales |
| 6. VERBE \rightarrow <i>attrape</i> | |
| 7. NOM \rightarrow <i>chat</i> <i>rat</i> ¹ | |
| 8. ADJECTIF \rightarrow <i>vieux</i> <i>petit</i> | |

Opérations de dérivation

Étant donné un ensemble de règles de grammaire et une phrase à tester P, comment vérifier que P est conforme à la grammaire ?

Réponse : en générant P à l'aide des règles. Il s'agit d'un **processus de réécriture** (replaces)

N° de la règle	Résultat
-	PHRASE
1	GROUPE SUJET VERBE GROUPE COMPLÉMENT OBJET
2	GROUPE NOMINAL VERBE GROUPE COMPLÉMENT OBJET
2	GROUPE NOMINAL VERBE GROUPE NOMINAL
4b	ARTICLE ADJECTIF NOM VERBE GROUPE NOMINAL
4b	ARTICLE ADJECTIF NOM VERBE ARTICLE ADJECTIF NOM
5	<i>le</i> ADJECTIF NOM VERBE ARTICLE ADJECTIF NOM
...	...
...	<i>le vieux</i> NOM <i>attrape le</i> ADJECTIF NOM
7a	<i>le vieux chat attrape le</i> ADJECTIF NOM
8b	<i>le vieux chat attrape le petit</i> NOM
7b	<i>le vieux chat attrape le petit rat</i>

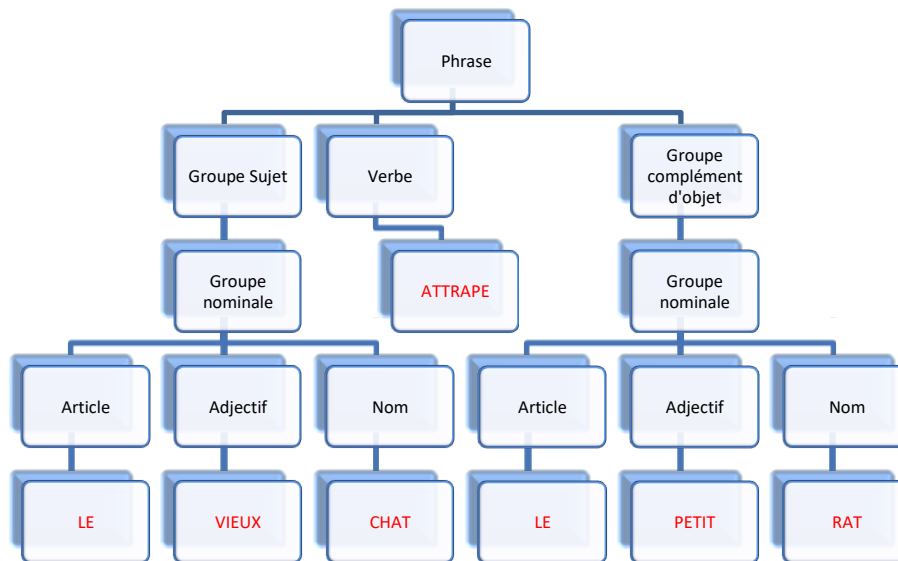
¹ Équivalent à NOM \rightarrow *chat* ou NOM \rightarrow *rat*

Résultat de l'analyse syntaxique :

On arrive bien à obtenir notre phrase P en réécrivant le symbole de départ PHRASE . On dit que P dérive de PHRASE, OU QUE P est obtenu par dérivation à partir de PHRASE. Ceci signifie que P est une phrase conforme à la grammaire, P fait partie des phrases bien formées relativement à la grammaire.

Il faut bien comprendre que l'opération de base dans la programmation à partir de grammaires est l'opération de réécriture (vs l'instanciation pour la programmation classique)

On peut représenter cette dérivation à l'aide d'un arbre. La phrase à analyser coïncide avec la frontière de l'arbre :



Il faut bien comprendre que l'arbre syntaxique d'une phrase a 2 rôles :

1. L'arbre assure la bonne formation de celle-ci
2. L'arbre de dérivation est le support idéal pour des traitements ultérieurs : calculs, traduction,

Exercice

Installez python 3 et exécutez le fichier python CFGGram.py²

1. Etoffe la grammaire
 - Ajouter les compléments circonstanciels de lieu (CCL) : Derrière la barrière, Tout près de là, ...
 - Ajoutez les compléments circonstanciels de manière (CCM) : doucement, comme s'il n'y avait pas de lendemain, ...
 - Ajoutez une autre fonction syntaxique
https://fr.wikipedia.org/wiki/Cat%C3%A9gorie:Fonction_syntaxique
2. Ajouter une règle de grammaire qui double les phrases produites (répéter le print)
3. Afficher 3 groupes nominaux

² Pour une bonne introduction à python : <http://python-liesse.enseeiht.fr/documents/python3.pdf>

4. Langages et grammaires

Grammaires

Définition : une grammaire $G = \langle V_T, V_n, S \in V_n, P \rangle$ où :

V_T (ou Σ) : Alphabet terminal
 V_n : Alphabet non terminal (ou alphabet des variables)
 $S \in V_n$: Symbole de Start (démarrage)
 P : Ensembles de règles de production

Les règles doivent comporter au moins un non terminal à gauche :

$P : \{x \rightarrow y\}$ avec :

$$x \in (\Sigma \cup V_n)^* \cdot V_n^+ \cdot (\Sigma \cup V_n)^*$$

$$y \in (\Sigma \cup V_n)^*$$

Dérivation immédiate

Soient u et $v \in (\Sigma \cup V_n)^*$

On note $u \rightarrow v$ si et seulement si $\exists p, s$ tels que $u = p.x.s$ et $v = p.y.s$ et $x \rightarrow y \in P$

Dérivation

On note $u \xRightarrow[G]{\Rightarrow} v$ (ou simplement $u \Rightarrow v$) si et seulement si il existe n dérivations immédiates à l'aide de règles de la grammaire G telles que $u \rightarrow \dots \rightarrow \dots \rightarrow v$

L'opérateur de dérivation correspond à la fermeture transitive de l'opérateur de dérivation immédiate.

Langage engendré par une grammaire

On note $L(G)$ le langage engendré par une grammaire G . $L(G)$ est défini comme :

$$L(G) = \{w \in \Sigma^*, S \xRightarrow[G]{\Rightarrow} w\}$$

Exemple :

Soit G une grammaire définie comme :

Σ ou $V_T = \{a, b\}$ Alphabet terminal
 $V_n = S, A$ Alphabet non terminal (variables)
 $S \in V$: Start (démarrage)
 P : Règles de production
 $S \rightarrow abA \mid \varepsilon$
 $A \rightarrow Aa \mid \varepsilon$

Les dérivations successives sont de la forme $S \rightarrow abA \rightarrow abAa \rightarrow abAaa \rightarrow abaaa$

Exercice : Expliquer pourquoi le langage engendré est $ab \cdot a^* \cup \{\varepsilon\} = L(G)$

Arbre de dérivation d'un mot

Considérons la grammaire suivante $G = \langle \{a, b\}, \{S, A, B\}, S, \text{règles} \rangle$ avec les règles suivantes :

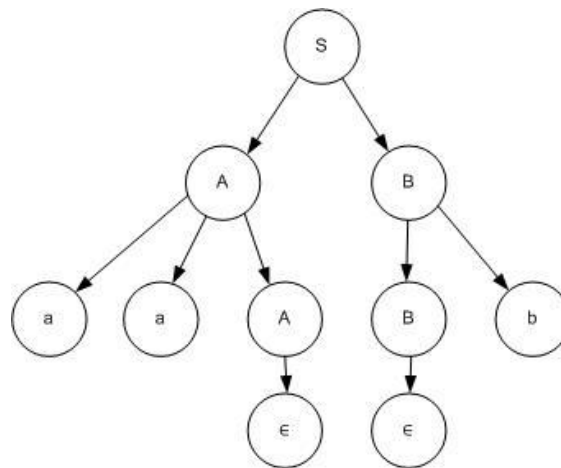
N°	Règle
1	$S \rightarrow AB$
2	$A \rightarrow aaA$
3	$A \rightarrow \varepsilon$
4	$B \rightarrow Bb$
5	$B \rightarrow \varepsilon$

Ainsi que le mot $w = aab$.

On peut considérer,

- soit la dérivation gauche (leftmost : $S \rightarrow AB \rightarrow aaAB \rightarrow aaB \rightarrow aab$),
- soit la dérivation droite (rightmost : $S \rightarrow AB \rightarrow Ab \rightarrow aaAb \rightarrow aab$)

Si on représente une dérivation avec un arbre, les 2 dérivation donnent lieu au même arbre : l'ordre d'application des règles ne change pas l'arbre résultant.



Grammaires ambiguës

Une grammaire est dite *ambiguë* s'il existe un mot du langage engendré qui peut être obtenu par 2 arbres de dérivation différents.

Grammaires équivalentes

Deux grammaires sont dites

- *faiblement équivalentes* si elles engendrent le même langage
- *fortement équivalentes* si elles produisent le même langage et les mêmes dérivation

Exercices :Exercice 1.

Trouver les grammaires (règles de production) qui engendrent les langages suivants :

$$L_1 = \{ a^n, n \geq 0 \}$$

$$L_2 = \{ a^n, n > 0 \}$$

$$L_3 = \{ \text{les mots sur } \{a, b\} \text{ qui comportent un seul } a \text{ au maximum} \} = \{ w \in \{a, b\}^*, |w|_a < 2 \}$$

$$L_4 = \{ a^n b^m, n, m \geq 0 \}$$

$$L_5 = \text{les mots sur } a \text{ et } b \text{ où } b \text{ est immédiatement suivi d'un } a$$

$$L_6 = \{ \text{les mots composés de :} \}$$

- un a, suivi de
- (une série non vide de b séparés par des c) ou un d, suivi de
- une série (possiblement vide) de f

Exercice 2

$$L_1 = \{ a^n b^n, n \geq 0 \}$$

$$L_2 = \{ \text{Palindromes sur } \Sigma = \{a, b\} \}$$

$$L_3 = \{ \omega \in \{a, b\}^*, |\omega|_a \% 3 = 0 \} \text{ où } |\omega|_a \text{ désigne le nombre de } a \text{ dans } \omega$$

$$L_{32} = \{ a^n b^{m+n} c^m, n > 0 \}$$

$$L_4 = \{ a^n b^n c^n, n > 0 \}$$

$$L_5 = \{ a^n b^m c^{m+n}, n > 0 \}$$

$$L_6 = \{ a^n b^n c^n d^n, n > 0 \}$$

Exercice 3

Déterminer les langages engendrés par les grammaires suivantes

$G_1 :$

$$S \rightarrow aSbb \mid \varepsilon$$

$G_2 :$

$$S \rightarrow aB \mid bA$$

$$A \rightarrow a|aS|bAA$$

$$B \rightarrow b|bS|aBB$$

G_3 :

$$S \rightarrow aSb|bSa|SS|\epsilon$$

Exercice 4 Langage de parenthèses

Soit $L = \text{le langage de Dyck} = \{ \text{séquences de parenthèses bien équilibrées} \}$

Trouver une grammaire G qui engendre L . G est elle ambiguë ?

Si oui, trouver une grammaire G' faiblement équivalente à G mais non ambiguë.

Exercice 5 Expressions arithmétiques

- Ecrire la grammaire des expressions arithmétique bien formées.
 $(1 + 34)/(5 * 9) \in L(G)$
 $+25 - (\quad) \notin L(G)$
- Incorporez le moins unaire (toujours entre parenthèses) : $(-1) + 25/(4 + (-2) * 3) \in L(G)$

Exercice 6 Signatures des fonctions en Java

Ecrire la grammaire des signatures de fonction en java.

Exemple :

```
void toto()
```

```
char tata(float f, int[][] z)
```

```
char tata(float f, int[] z [])
```

```
int toutou(int c2)
```

```
void afficherNotes(Etudiant e)
```

Remarque :

- les identifiants (noms des variables et des fonctions) doivent commencer par une minuscule ou un _
- Les noms de classe doivent commencer par une majuscule
- les noms de classes commencent par une majuscule
- on se limitera aux types primitifs boolean, int et char
- on n'oubliera pas les droits d'accès (private, public, protected), ainsi que les qualificateurs (static)
- on pourra utiliser les règles lexicales suivantes :
 - $C \rightarrow 0|1| \dots | 9$
 - $L \rightarrow \text{Maj} | \text{Min} | _$

- Maj $\rightarrow A|B|\dots|Z$
- Min $\rightarrow a|b|\dots|z$
- TypePrim $\rightarrow \text{int}|\text{char}|\text{Boolean}$
- qualify $\rightarrow \text{static}|\epsilon$
- access $\rightarrow \text{private}|\text{public}|\text{protected}|\epsilon$

Exercice 7 Commandes SQL

On souhaite écrire une grammaire chargée de vérifier la syntaxe des 2 commandes SQL suivantes (*select* et *insert*) avec des spécifications simplifiées. On ne s'intéressera pas à l'aspect sémantique dans les commandes (par exemple, savoir si un identificateur est un nom de table ou de champ, ou vérifier l'harmonisation des types dans l'expression des contraintes)

Commande *select* : (select ... from ...) OU (select ... from ... where ...)

une partie obligatoire

entre *select* et *from* : *, un identificateur, ou une liste d'identificateurs séparés par des virgules

juste après le *from* : un identificateur, ou une liste d'identificateurs séparés par des virgules

une partie optionnelle :

après le *where* : une expression booléenne non ambiguë, c'est-à-dire des égalités, des non-égalités ou des comparaisons entre des termes qui sont des littéraux ou des identificateurs. On peut combiner les expressions booléennes avec les opérateurs logiques classiques (AND ou OR). Des parenthèses sont nécessaires (car `(true or true) and false` ne vaut pas `true or (true and false)`)

Les littéraux sont soit des nombres, soit la référence null, soit des chaînes de caractères entre apostrophes.

Les identificateurs (noms de champ, noms de tables, ...) sont des suites de lettres ou de chiffres commençant obligatoirement par une lettre.

La commande est clôturée par un point-virgule

Commande *insert into* : (insert into ...(...) values (...)) OU (insert into ... values (...))

entre le *into* et le *values* : on a un identificateur, suivi ou non d'une liste d'identificateurs entre parenthèses séparés par une virgule.

après le *values* : on a une liste de littéraux entre parenthèses séparés par des virgules

La commande est clôturée par un point-virgule

Exemples de commandes SQL bien formées :

```
select abc from table1, table2 ;
select abc, def from table1, table2 ;
select * from table1, table2;
select abc, def from table1 where abc==der+1;
select abc, def from table1 where 1==2 AND (toto!=null OR
toto<'tata');
```

Et aussi avec le select imbriqué :

```
select abc, def from select abc, def from table1, table2 ;
;
insert into table1 values ('test', 5);
insert into table1 (champ1, champ2) values ('test', 5);
insert into table1 (champ1, champ2, champ3) values ('test', 5);
insert into table1 (champ1, champ2) values ('43', 5, null);
```

Exemples d'expressions SQL mal formées :

```
select abc, from table1, table2 ;
malformation : pas de virgule si un seul champ (abc)
```

```
select 1abc, def from table1, table2 ;
malformation : pas d'identificateur commençant par un chiffre
```

```
insert into (champ1, champ2) values ('test', 2);
malformation : pas d'identificateur avant les parenthèse entre le
into et le values
```

On rappelle que l'alphabet terminal se compose de tous les symboles présents dans les commandes. Les mots clefs réservés (select, insert, into, values, AND, OR, null, ...) seront considérés comme des symboles élémentaires de l'alphabet terminal.

On aura donc

$$V_t = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, \dots, z, A, \dots, Z, (,), :, <, >, \text{AND}, \text{OR}, =, !, *, \text{select}, \text{insert}, \text{into}, \text{values}, \text{null}\}$$

Pour les éléments de l'alphabet non terminal, on choisira les conventions de nommage suivantes :

- L pour lettre
- C pour chiffre
- A pour caractère alphanumérique (une lettre ou un chiffre)
- ID pour les identificateurs
- LID pour liste d'identificateurs

- LITT pour littéraux
 - LLITT pour liste de littéraux
 - TERM pour les termes (littéraux ou identificateurs)
 - OPR pour opérateur relationnel
 - OPB pour opérateur booléen (OU et ET), pas de NON
 - EXB pour expression booléenne
 - S pour le symbole de départ
1. Ecrire la grammaire des 2 commandes SQL Select et Insert
 2. Ecrire la grammaire d'un script SQL uniquement constitué d'une succession de requêtes Select et Insert

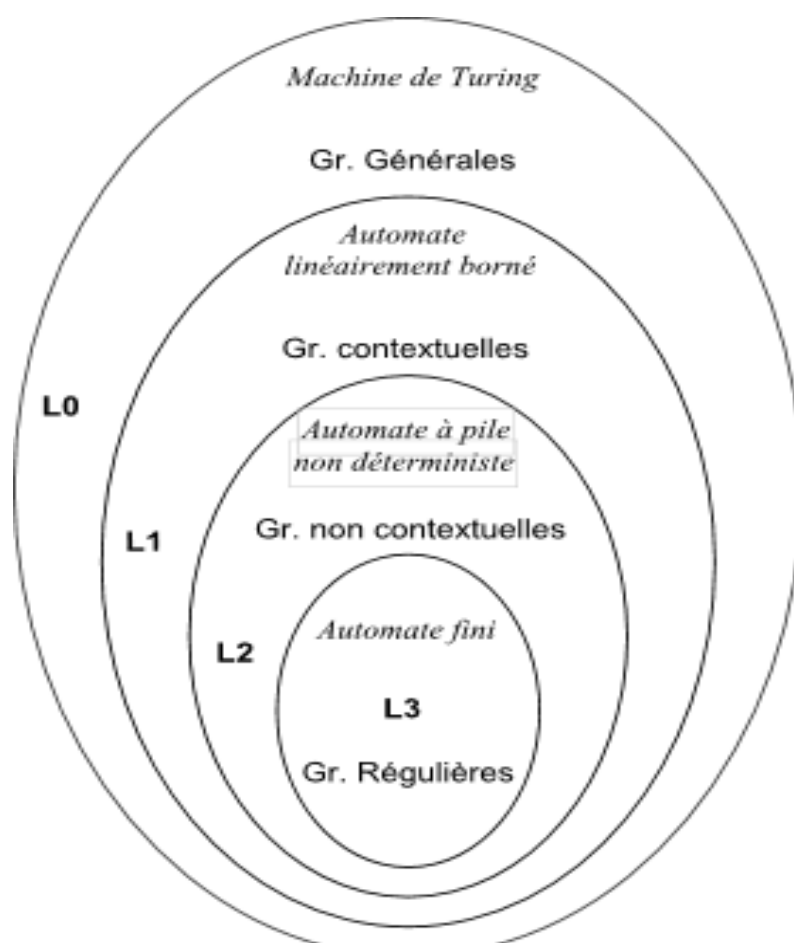
Exercice 8 LOGO

Ecrire une grammaire simplifiée du langage LOGO et générer aléatoirement des programmes LOGO avec le fichier python CFGGram.py

Exercice 9 L-systems

<https://runestone.academy/runestone/books/published/thinkcspy/Strings/TurtlesandStringsandLSystems.html>

5. Classification des grammaires : Hiérarchie de Chomsky



Type 0 : Grammaires générales

Les règles sont de la forme :

$\gamma \rightarrow \beta$ où $\gamma, \beta \in (V_n \cup V_t)^*$ avec γ contient au moins un non terminal

Les reconnaisseurs sont les Machines de Turing. La non appartenance d'un mot n'est pas prouvable (on ne peut pas prouver que la machine ne va pas s'arrêter : on ne sait pas combien de temps attendre avant de pouvoir conclure).

Classe de langages : **langages récursivement énumérables.**

Type 1 : Grammaires contextuelles

Les règles sont de la forme :

$\gamma A \beta \rightarrow \gamma \alpha \beta$ où $A \in V_n$ et $\alpha, \gamma, \beta \in (V_n \cup V_t)^*$ avec $\gamma \neq \varepsilon$

Remarque terminologique : $\gamma \dots \beta$ est vu comme le contexte d'application de la règle $A \rightarrow \alpha$

Les reconnaisseurs sont les automates linéairement bornés (= une Machines de Turing avec une mémoire linéairement borné). La non appartenance d'un mot n'est pas prouvable, elle est indécidable.

Classe de langages : **langages contextuels**

Type 2 : Grammaire hors contexte (ou grammaire algébrique)

Les règles sont de la forme :

$A \rightarrow \omega$ où $A \in V_n$ et $\omega \in (V_n \cup V_t)^*$

Remarque terminologique : le non terminal A est remplacé indépendamment de son contexte, c'est-à-dire de sa place dans le mot.

Les reconnaisseurs sont des automates à piles. L'appartenance et le non –appartenance sont directement lisibles sur l'état d'aboutissement du mot : final ou non final. Le problème est décidable.

Classe de langages : **langages non contextuels ou langages algébriques**

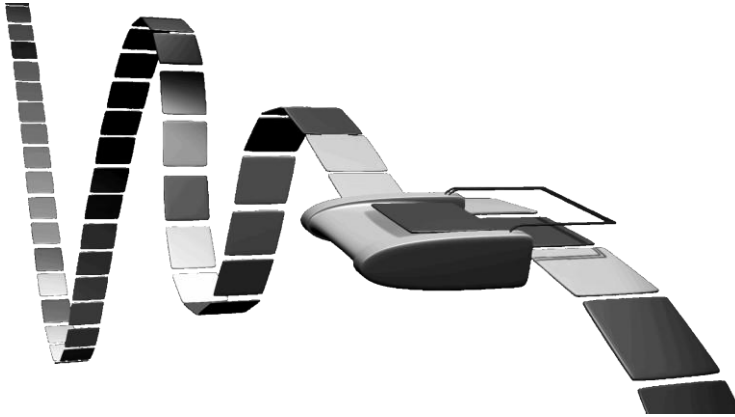
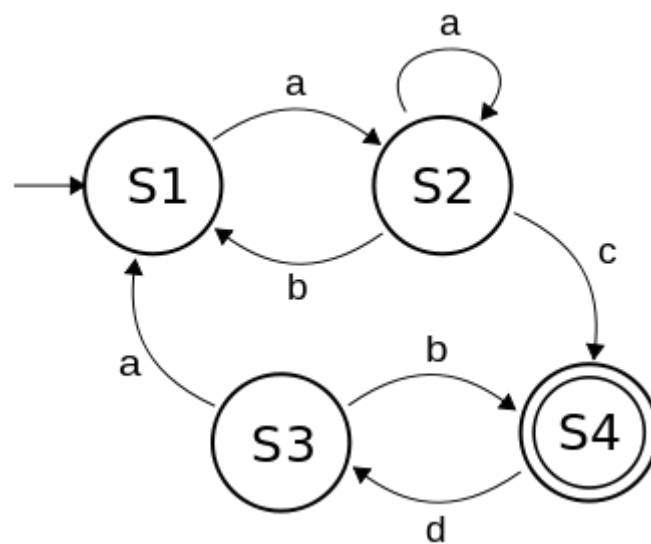
Types 3 : Grammaires régulières

Les règles sont de la forme :

$A \rightarrow aB$ où $A, B \in V_n$ et $a \in V_t$
 $A \rightarrow a$

Les reconnaisseurs sont des automates d'états finis. L'appartenance et le non –appartenance sont directement lisibles sur l'état d'aboutissement du mot : final ou non final. Le problème est décidable.

Classe de langages : **langages réguliers ou langages rationnels**

Machine de Turing**Automate à états finis**

6. Analyse syntaxique ascendante

Le problème de reconnaissance

Soient une grammaire G et un mot ω . Le problème de reconnaissance consiste à déterminer si $\omega \in L(G)$ ou si $\omega \notin L(G)$

Donc, s'il existe un chemin de dérivation tel que $S \Rightarrow^{(*)} \omega$.

C'est le rôle de l'analyse syntaxique :

Entrée :

- une grammaire hors contexte G
- une chaîne d'entrée w

Sortie :

- réussite : w est accepté par G . Dans ce cas, l'analyseur fournit l'arbre d'analyse associé à w
- échec : w n'est pas accepté, il ne fait pas partie du langage engendré par la grammaire G

Il est impératif de concevoir l'analyse syntaxique comme la (tentative de) construction de l'arbre d'analyse de la chaîne d'entrée.

Réductions

Dans les analyseurs ascendants, la reconnaissance de la chaîne d'entrée se fait de bas en haut : des feuilles de l'arbre à sa racine. On cherche à savoir s'il est possible de *réduire* la chaîne d'entrée (w) au symbole de départ (S), par une application judicieuse des règles de la grammaire. A chaque étape de la réduction, si une sous chaîne correspond à la partie droite d'une règle, elle est remplacée par le non terminal qui constitue la partie gauche de la règle. Les décisions clés à prendre au cours de l'analyse descendante concernent le moment où il faut réduire, et la production à utiliser pour réduire.

Par exemple, pour la grammaire G des expressions de variables (id) :

$$E \rightarrow E + T \mid T$$

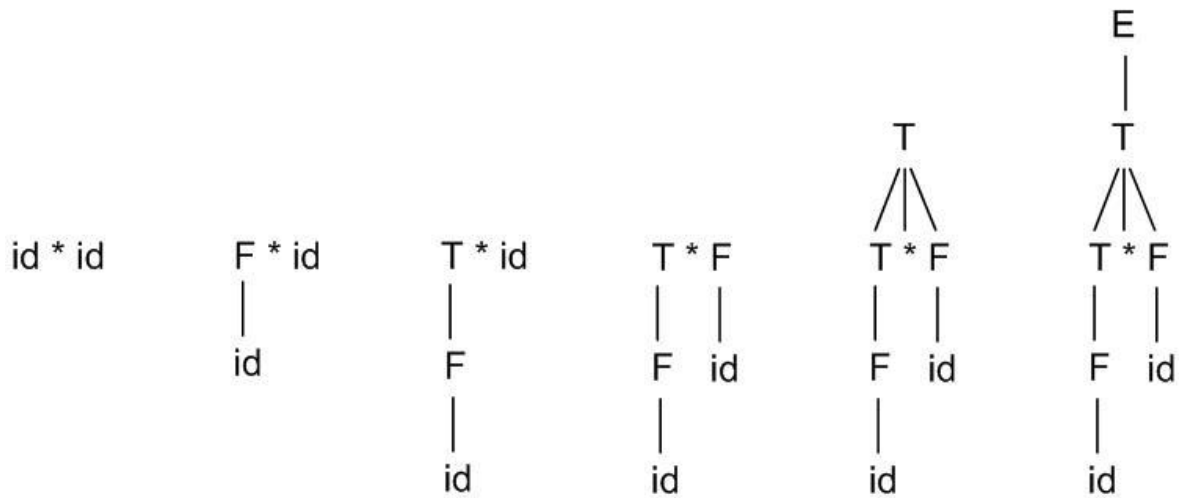
$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Les étapes de la réduction d'une chaîne d'entrée $w = id * id$:

id * id, F * id, T * id, T * F, T, E

Chaque étape correspond à une phase de la construction de l'arbre de syntaxe de la chaîne *w*.



1. la première réduction réduit le ***id*** le plus à gauche et le transforme en *F*
2. La seconde réduit *F* en *T*.
3. ici on a le choix :
 - a. soit réduire *T* en *E*
 - b. soit réduire le second ***id*** en *F*. C'est cette option qui est choisie car c'est la seule qui conduise à l'analyse de la chaîne *w* : sinon, on reste bloqué avec *E*id*
4. Ensuite *T*F* en *T*
5. Et enfin *T* en *E*

Les étapes d'une réduction correspondent aux étapes d'une dérivation droite de la chaîne d'entrée mais dans l'ordre inverse :

$$E \rightarrow T \rightarrow T * F \rightarrow T * id \rightarrow id * id$$

Analyseurs syntaxiques par décalage-réduction

Principe :

- une pile stocke des symboles grammaticaux (des proto phrases)
- un tampon d'entrée stocke la partie de la chaîne restant à analyser.

Etat initial de l'analyse : Au départ, la pile est vide et la chaîne w est sur l'entrée :

Pile	Entrée
\$	$id_1 * id_2 \$$

Étapes intermédiaires : au cours d'un parcours gauche-droite de la chaîne d'entrée, l'analyseur *décale* des symboles terminaux de la chaîne d'entrée vers la pile, jusqu'à être en mesure de *réduire* le haut de la pile en la partie gauche de la règle appropriée. Ce cycle est itéré.

Etat final de l'analyse : l'entrée est vide et la pile est réduite au symbole de départ. L'analyseur annonce le succès de l'analyse.

Pile	Entrée
$\$ \$$	$\$$

Exemple :

Pile	Entrée	Actions
\$	$id_1 * id_2 \$$	décaler
$\$ id_1$	$* id_2 \$$	Réduire par $F \rightarrow id$
$\$ F$	$* id_2 \$$	Réduire par $T \rightarrow F$
$\$ T$	$* id_2 \$$	décaler
$\$ T^*$	$id_2 \$$	décaler
$\$ T^* id_2$	\$	Réduire par $F \rightarrow id$
$\$ T^* F$	\$	Réduire par $T \rightarrow T * F$
$\$ T$	\$	Réduire par $E \rightarrow T$
$\$ E$	\$	Analyse réussie

Symbole de pré-vision : La majorité des analyseurs sont LR(k). Cela signifie que le choix de l'analyseur syntaxique (par exemple le fait de décider s'il faut réduire ou décaler à l'étape 4) est basé sur les k prochains symboles de la chaîne d'entrée. Attention, il ne s'agit pas d'un conflit car une seule des possibilités conduit à l'analyse de la chaîne. Dans ce cours, on ne va considérer que les grammaires LR(1), c'est-à-dire où un seul symbole de pré-vision est considéré (ici ce symbole vaut *).

Exercice : analyse par décalage-réduction

Soit G la grammaire suivante qui définit les expressions arithmétiques restreintes aux nombres formés des chiffres de 1 à 4, à l'addition et à la multiplication.

$$S \rightarrow S+S \mid S*S \mid N$$

$$N \rightarrow NC \mid C$$

$$C \rightarrow 1 \mid 2 \mid 3 \mid 4$$

Les symboles sont catégorisés selon :

- symboles terminaux (tokens) : 1, 2, 3, 4, +, *
- symboles non terminaux : S, N et C

Soit w la chaîne d'entrée à analyser : $w = 12+3*4$

1. Donner 2 arbres dérivation pour le mot w ; Pour chaque arbre, on précisera les étapes d'une analyse ascendante par décalage réduction de la chaîne w (pour les réductions, on précisera juste le numéro de la règle utilisée parmi les 9 règles de la grammaire, numérotées dans l'ordre : par exemple, la règle $S \rightarrow S*S$ est la règle numéro 2)
2. Lequel des 2 arbres vous semble il le plus approprié pour de futurs traitements (comme calculer la valeur de l'expression par exemple)

Conflits pendant une analyse par décalage réduction

Conflit n°1 : le conflit décalage réduction

Par exemple, dans le cas des règles :

$$inst \rightarrow \mathbf{si\ expr\ alors\ inst} \mid \mathbf{si\ expr\ alors\ inst\ sinon\ instr} \mid \mathbf{autre\ ...}$$

Dans la configuration suivante :

Pile	Entrée
\$... si expr alors inst	sinon \$

On ne peut pas dire s'il faut

- réduire en inst puis rechercher un autre alors pour décaler le sinon
- décaler le sinon et rechercher une inst à droite pour réduire en inst.

Conflit n°2 : le conflit réduction/réduction

Ce conflit advient dans le cas où plusieurs règle la même une partie droite, lorsque celle-ci apparaît au sommet de la pile :

$$\begin{aligned} accesTableau &\rightarrow identificateur \quad (' listeExpr')' \\ appelFonction &\rightarrow identificateur \quad (' listeExpr')' \end{aligned}$$

Utiliser des grammaires ambiguës

Grammaire ambiguë des expressions arithmétiques : $E \rightarrow E + E \mid E * E \mid (E) \mid id$

Grammaire non ambiguë des expressions arithmétiques :

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Cette dernière grammaire donne la priorité à * par rapport à + (par ex. $2+3*4=2+(3*4)$) et rend les 2 opérateurs associatifs à gauche (par ex. $2+2+2=(2+2)+2$) (le vérifier !)

Si l'on souhaite utiliser des grammaires ambiguës (plus concises que leurs homologues non ambigus), il faut pouvoir spécifier dans tous les cas des règles de désambiguïsation qui n'autorisent qu'un seul arbre de syntaxe. Dans la section sur Yacc, on verra comment procéder à de tels spécifications.

Utiliser des grammaires ambiguës³

1. Règle de résolution des conflits : comportement par défaut

Conflit réduction/réduction : c'est la réduction listée en premier qui est choisie

Conflit réduction/décalage : c'est le décalage qui est choisi

2. Règle de résolution des conflits : mécanisme général

Dans la partie 2, on peut définir des associativités et des priorités.

```
%left '+' '-'
%left '*' '/'
%right '^'
%nonassoc '<'
```

Cela signifie :

- les terminaux '+' '-' '*' '/' sont associatifs à gauche.
- '+' '-' ont même priorité, mais celle-ci est moindre que celle de '*' '/' car ces derniers sont listés après
- le terminal '^' est associatif à droite
- '<' est non associatif (on ne peut pas combiner 2 occurrences de suite)

Avec ces associativités et ces priorités, les conflits sont résolus :

Conflit réduction/décalage :

- on associe à chaque règle une *priorité qui est celle de son terminal le plus à droite*.
- pour résoudre le conflit, on compare les priorités de la règle, et du terminal de pré-vision
 - o si la priorité de la règle est supérieure, on réduit. Si elle est inférieure, on décale.
 - o si la priorité de la règle est égale,
 - on réduit si le terminal est associatif à gauche
 - on décale sinon.

Dans les cas où le terminal le plus à droite d'une règle ne fournit pas la bonne priorité, on peut forcer une priorité en ajoutant à une production l'étiquette `%prec <terminal>`. La priorité et l'associativité est alors la même que celle de <terminal>.

Conflit réduction/ réduction: la réduction apparaissant en premier dans le fichier ply est choisie

³ Invoquer yacc avec l'option `-v` permet de produire un fichier `y.output` qui récence les conflits rencontrés durant l'analyse.

Exercice : indiquer (sans exécuter le code) ce qui est affiché sur la console

```
import ply.lex as lex
import ply.yacc as yacc

tokens = (
    'A', 'B', 'NUMBER'
)

t_A = r'a'
t_B = r'b'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

lexer = lex.lex()

precedence = (
    ('right', 'A'),
    ('left', 'B'),
)

def p_rules(t):
    '''S : S A S
       | S B S
       | NUMBER'''
    if len(t) is 2 : t[0] = t[1]
    elif t[2] == 'a' : t[0] = 2*t[1] + t[3]
    elif t[2] == 'b': t[0] = t[1] - 2*t[3]
    print(t[0])

parser = yacc.yacc()
parser.parse('1 a 2')
print()
parser.parse('1 a 2 a 3')
print()
parser.parse('1 a 2 b 3')
```

Input \ Priorités	precedence = (('right','A'), ('left','B'),)	precedence = (('right','B'), ('left','A'),)
1 a 2		
1 a 2 a 3		
1 a 2 b 3		
1 a 2 b 3 a 4		

TP PLY1

Python Lex Yacc

Prise en main

Installation

- Python 3
- PLY : <https://github.com/dabeaz/ply>
 - o `python3 -m pip install ply`

Pour installer PLY sur votre machine pour python2 / 3, procédez comme suit:

1. Téléchargez le code source à partir d' [ici](#) .
2. Décompressez le fichier zip téléchargé
3. Naviguez dans le dossier `ply-3.10` décompressé
4. Exécutez la commande suivante dans votre terminal: `python setup.py install`

<https://riptutorial.com/fr/python/example/31582/premiers-pas-avec-ply>

- Manuel : <https://www.dabeaz.com/ply/ply.html>
- testez votre installation sur le fichier `calcBase.py`

Notions de TD :

- Arbre de dérivation
- Problème de génération vs problème de reconnaissance
- analyse ascendante par décalage réduction

calcBase.py :

PARTIE LEXICALE

```
tokens = (
    'NUMBER','MINUS',
    'PLUS','TIMES','DIVIDE',
    'LPAREN','RPAREN'
)
```

```
t_PLUS = r'\+'
t_MINUS = r'\-'
t_TIMES = r'\*'
t_DIVIDE = r'\/'
t_LPAREN = r'\('
t_RPAREN = r'\)'
```

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

PARTIE SYNTAXIQUE

```
def p_statement_expr(p):
    'statement : expression'
    print(p[1])
```

```
def p_expression_binop_plus(p):
    'expression : expression PLUS expression'
    p[0] = p[1] + p[3]
```

```
def p_expression_binop_times(p):
    'expression : expression TIMES expression'
    p[0] = p[1] * p[3]
```

```
def p_expression_binop_divide_and_minus(p):
    '''expression : expression MINUS expression
    | expression DIVIDE expression'''
    if p[2] == '-': p[0] = p[1] - p[3]
    else : p[0] = p[1] / p[3]
```

```
def p_expression_group(p):
    'expression : LPAREN expression RPAREN'
    p[0] = p[2]
```

```
def p_expression_number(p):
    'expression : NUMBER'
    p[0] = p[1]
```

```
s = input('calc > ')
```


yacc.parse(s)

Grammaire :

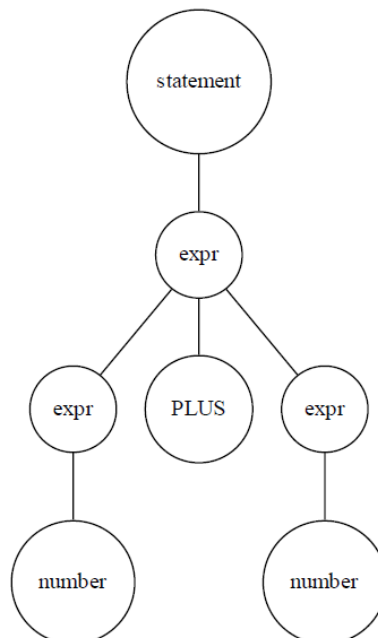
statement \rightarrow expression
 expression \rightarrow expression PLUS expression
 expression \rightarrow expression TIMES expression
 expression \rightarrow expression MINUS expression | expression DIVIDE expression
 expression \rightarrow LPAREN expression RPAREN
 expression \rightarrow NUMBER

Exemple d'analyse

Input : « 12+3 »

Tokens : NUMBER PLUS NUMBER

Pile	Entrée	Actions
\$	NUMBER PLUS NUMBER\$	décalage
NUMBER	PLUS NUMBER\$	Reduction par ' expression : NUMBER '
expression	PLUS NUMBER\$	2 décalages
expressionPLUS NUMBER	\$	Reduction par ' expression : NUMBER '
expression PLUS expression	\$	Reduction par ' expression : expression PLUS expression '
expression	\$	Réduction par ' statement : expression '
statement	\$	OK analyse terminée



Exercices de TP

Exercice 1 Calcul booléen

On souhaite adapter le calculateur aux opérateurs booléens entre expressions arithmétiques :

0 & 3 vaut 0, 1 & 3 vaut 1 (ou 3) ...

Attention, pour le code python utiliser les opérateurs *and* et *or* (et pas & et |)

- Partie lexicale : Ajouter les token '&' et '\|'
- Partie syntaxique : Ajouter une fonction `def p_expression_binop_bool(p):`
 - o Dans la docstring, ajouter les règles de productions correspondant aux expressions booléennes
 - o adapter le code python

Exercice 2 Multiligne

On souhaite permettre le calcul de plusieurs expressions, séparées par les ';' :

- Ajouter le token semicolon
- Ajouter un symbol non terminal START, ainsi qu'une règle indiquant que le start se compose d'une série de statement (un bloc d'instructions) dans une nouvelle fonction `p_bloc(p)`

Exercice 3 Variables et affectation

1. Ajouter un token NAME pour les noms de variables
2. En utilisant un dictionnaire python⁴, mettez en œuvre l'affectation de variable (`x=1+1`), ainsi que le calcul d'expressions comprenant des variables (`x+2+1`)

Exercice 4 Calcul booléen 2

On souhaite adapter le calculateur aux expressions booléennes formées, non plus seulement de combinaison de true et false, mais aussi de comparaison entre expressions arithmétiques. Par exemple : `1+3<2` qui doit être conduire à **False**

⁴ initialisation : `names={}`

Exercice 5 Print

Faites en sorte que le résultat d'une expression ne soit affichée sur la console que si elle figure en argument d'une fonction PRINT : `print(1+2)` ;

Du coup, il existe 2 statements : le print et l'affectation

Impératif : Pour ne pas confondre les mots clefs du langage (print) avec les noms de variable, voir le manuel section 4.3 token specification : reserved https://www.dabeaz.com/ply/ply.html#ply_nn6

Exercice 6 Conflits et précédences

Ajouter des règles de précedence afin de supprimer tous vos conflits shift/reduce à l'aide d'une instruction comme :

```
precedence = (
    ('nonassoc', 'C'),
    ('right', 'A', 'B'),
    ('left', 'D'),
)
```

Dans le cas ci-dessus, on a

1 C 2 C 3 qui conduit à une erreur car C n'est pas associatif

1 C 2 A 3 = 1 C (2 A 3) car A est plus prioritaire que C

1 A 2 A 3 = 1 A (2 A 3) car A est associatif à droite

1 A 2 B 3 = 1 A (2 B 3) car A et B ont même priorité et sont associatifs à droite

1 D 2 B 3 = (1 D 2) B 3 car D est plus prioritaire que B

Adaptez l'instruction de précedence à votre calculateur de manière à obtenir les comportements attendus dans les cas suivants :

- $a+b*c = a+(b*c)$
- $a+b<c = (a+b)<c$
- $a\&b|c = a\&(b|c)$
- $a|b\&c = a|(b\&c)$
- $1<x \& x<3 = (1<x) \& (x<3)$
- $1<x<3 = \text{impossible}$

Exercice 7.1 Bonus

Faites en sorte que $1 < x < 3 = (1 < x) \ \& \ (x < 3)$

Si vous voyez apparaître des conflits `reduce/reduce`, placez la règle à exécuter en premier dans votre fichier `ply`.

Exercice 7.2 Bonus

Ajouter les fonctionnalités suivantes :

- `x++`
- `x+=1`
- `//` les commentaires simple ligne
- `/*` les multi ...
... lignes `*/`
- la saisie clavier `saisir(x)` ou `x = saisieClavier()`
- Le print de String `PRINTSTRING(« toto »)`

TP PLY2

Arbre de syntaxe concret (Arbre de dérivation)

vs

Arbre de syntaxe abstrait (AST)

Fonctions de construction et d'évaluation

Préliminaires

Arbres comme un tuple (python)

En python, un tuple contient, comme une liste, un ensemble ordonné de valeurs. A la différence des listes, les tuples ne sont pas modifiables. On accède aux éléments du tuple comme pour une liste :

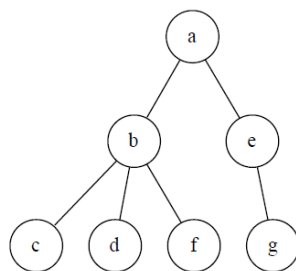
```
montuple=(3, 6, 2)
print(montuple[1])
```

On va coder les arbres sous forme d'un tuple t avec la convention suivante :

- $t[0]$ désigne le sommet de l'arbre
- $t[1], t[2], \dots, t[\text{len}(t)-1]$ désignent les sous arbre (les fils) s de l'arbre t (de gauche à droite)

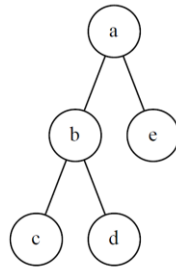
Le nombre de fils est quelconque :

```
tree = ( 'a', ('b', 'c', 'd', 'f'), ('e', 'g') )
```



Dans les arbres dits *binaires* chaque nœud a 2 fils, et donc $\text{len}(t)=3$.

Par exemple, avec $\text{tree}=('a', ('b', 'c', 'd'), 'e')$



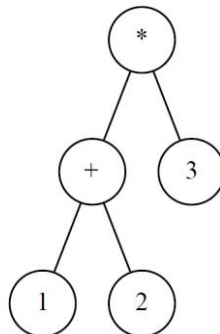
Arbres binaires d'expressions arithmétiques

Les expressions arithmétiques peuvent être stockées dans **des arbres binaires** :

- les nœuds représentent des opérateurs arithmétiques
- les feuilles représentent des nombres

Par exemple,

tree=('*', ('+', 1, 2), 3)



L'avantage de la représentation (en arbre binaire) est l'absence d'ambiguïté des expressions : les parenthèses sont inutiles. Cette notation est très proche de la notation préfixée des expressions arithmétiques

https://fr.wikipedia.org/wiki/Notations_infix%C3%A9e,_pr%C3%A9fix%C3%A9e,_polonaise_et_postfix%C3%A9e

Exercice (algo)

- Ecrire les arbres des expressions suivantes sous la forme de tuple python
 - $2*3+5*6$
 - $1+2+3$
 - $1+2-3$
 - $(1+2)*(3+4)*(5+6)$

Remarque : vous devriez retomber sur la notion d'associativité d'un opérateur : à gauche ou à droite

- Ecrire une fonction **eval** qui prend en paramètre le tuple d'une expression arithmétique et qui en renvoie la valeur.

Par exemple, `eval (('*', ('+', 1, 2), 3))` doit renvoyer 9

Vous aurez surement besoin de tester le type de tree : `type(tree)` vaut soit tuple, soit int (feuilles)

- **Bonus :**
 - Ecrire une fonction *tuple2ExprPrefix* qui prend en paramètre un tel tuple et qui affiche les expressions dans une forme préfixée. Idem avec *tuple2ExprPostfix*
 - Remarque : pour print sans passer à la ligne en python : `print(t, end="")`
 - Ecrire de même une fonction *tuple2ExprInfix*. Est-il possible d'éviter certaines parenthèses ? Y a-t-il un rapport avec le système de précédences de PLY ?

Ensuite, installez GraphViz et produisez un visuel des arbres de l'exercice précédent.

Visualisation d'arbres avec GraphViz

- télécharger l'utilitaire GraphViz ([Stable 2.38 Windows install packages](#))
- Ajouter le chemin du répertoire bin (qui contient dot.exe) à votre variable d'environnement PATH
- installez le module python graphviz (`py -m pip install graphviz`)
- testez le script [genereTreeGraphviz2.py](#) (et fermez le pdf à *chaque fois* avant de relancer)

windows : `set PATH=%PATH%;C:\Program Files (x86)\Graphviz2.38\bin`

mac : `brew install graphviz` puis `pip install` dans le projet

Exercices de TP

A. Construire l'arbre de dérivation = Arbre de syntaxe concret

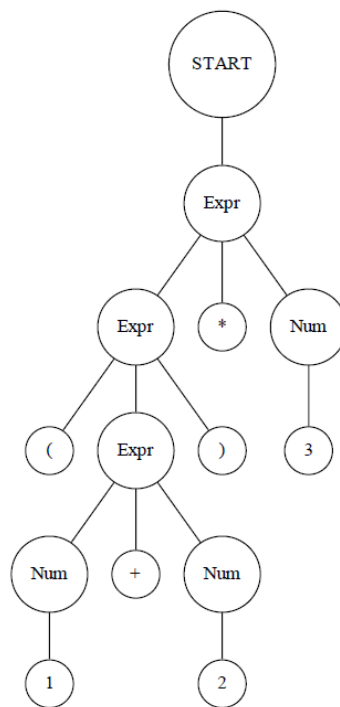
On considère la grammaire simplifiée suivante pour les expressions arithmétiques (nombres, +, * et parenthèses):

Start \rightarrow Expr

Expr \rightarrow Expr + Expr | Expr * Expr | (Expr) | Number

avec input = '(1+2)*3'

- a. Complétez le fichier *calcExprePourASC.py* de manière à ce qu'il affiche l'arbre de dérivation de l'input.



Correction = *calcExpreAvecASC.py*

- b. Complétez le fichier *calcExpreAvecASC.py* avec une fonction `eval(tree)` où `tree` est le tuple de l'arbre de dérivation, qui renvoie ou affiche le résultat

Correction = *calcExpreAvecASCavecEvalCorrection.py*

Arbre de dérivation (syntaxe concrète)

spécifique au langage, difficile à évaluer

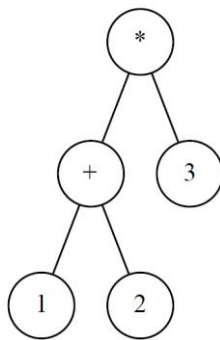
VS

Arbre de syntaxe abstraite (AST)

plus compact, sans marqueur syntaxique

B. Construire l'arbre de syntaxe abstrait (AST)

- c. Modifier le fichier *calcExpreAvecASCavecEvalCorrection.py* de manière à obtenir un arbre de syntaxe abstrait. Coder la fonction eval associée.



correction = *calcExpreAvecASTavecEvalCorrection.py*

Mini interpréteur avec AST - construction

Pour la suite des exercices, vous devez partir d'une version de votre calculateur qui doit impérativement inclure :

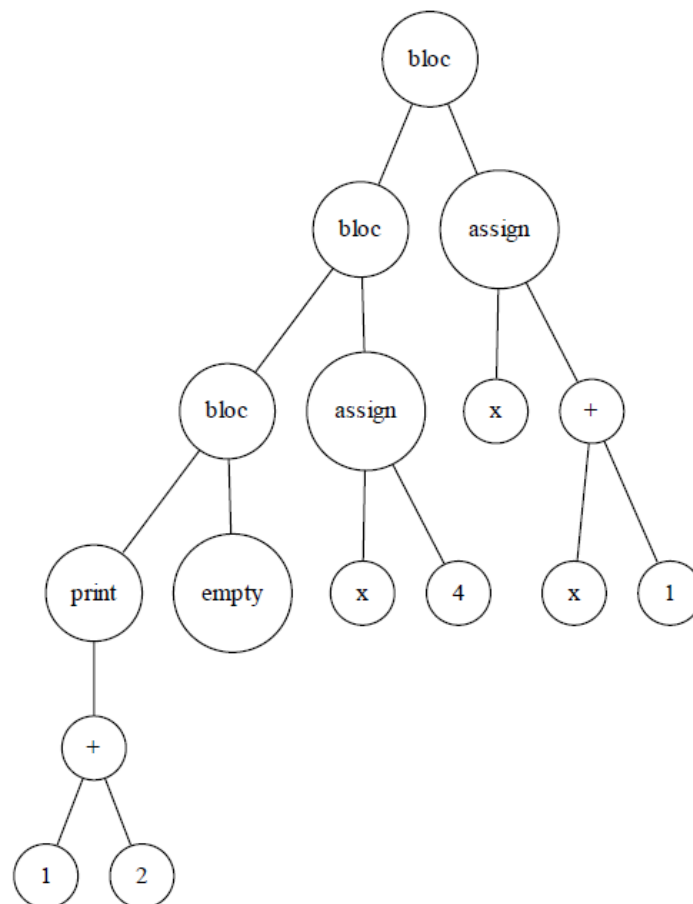
- le calcul d'expressions arithmétiques et booléennes
- le multiligne (bloc)
- l'affectation et les variables dans les expressions
- le print
- aucun conflit shift/reduce

Adaptez votre calculateur de manière à ce qu'il construise et affiche l'arbre correspondant à l'input

Remarque : aucun traitement calculatoire n'est attendu à ce stade.

1. **Input** : `s='print(1+2);x=4;x=x+1;'`

2. **Arbre** :



Remarque : il se peut que le nœud bloc de la première instruction soit positionné en haut de l'arbre

Mini interpréteur avec AST - Evaluation

Pour l'évaluation de votre AST, vous allez devoir coder 2 fonctions :

- une fonction **evalExpr** qui prend en paramètre les tuples d'expressions arithmétiques et booléennes, et qui renvoie leur valeur
- une fonction **evalInst**, pour l'évaluation des instructions, qui ne renvoie pas de valeur. Intuitivement, évaluer une instruction, c'est l'exécuter.

Voici des traces d'exécution possible pour le tuple :

```
tree=('bloc', ('bloc', ('bloc', ('print', ('+', 1, 2)), 'empty'),
('assign', 'x', 4)), ('assign', 'x', ('+', 'x', 1)))
```

- evalInst de ('bloc', ('bloc', ('bloc', ('print', ('+', 1, 2)), 'empty'), ('assign', 'x', 4)), ('assign', 'x', ('+', 'x', 1)))
- evalInst de ('bloc', ('bloc', ('print', ('+', 1, 2)), 'empty'), ('assign', 'x', 4))
- evalInst de ('bloc', ('print', ('+', 1, 2)), 'empty')
- evalInst de ('print', ('+', 1, 2))
- evalExpr de ('+', 1, 2)
- evalExpr de 1
- evalExpr de 2
- CALC> 3
- evalInst de empty
- evalInst de ('assign', 'x', 4)
- evalExpr de 4
- evalInst de ('assign', 'x', ('+', 'x', 1))
- evalExpr de ('+', 'x', 1)
- evalExpr de x
- evalExpr de 1

Mini interpréteur avec AST - IF, WHILE ET FOR

- Incorporez les IF et les boucles dans votre mini interpréteur.
- Testez votre interpréteur sur la suite de Fibonacci . Le code de la suite devra être lu sur un fichier extérieur.
- Bonus : incorporez les x++, x+=1, printString(« toto »)

TP PLY3

Fonctions

Prérequis : pour ce TP, vous devez avoir un calculateur :

- sans conflits
- avec un AST et 2 fonctions d'évaluation evalExpr et evalInst
- avec les if et les boucles while (et for eventuellement)

Principe général de gestion des fonctions :

- **définition des fonctions** : stockées sous la forme d'un tuple dans un dict *functions*

Par exemple :

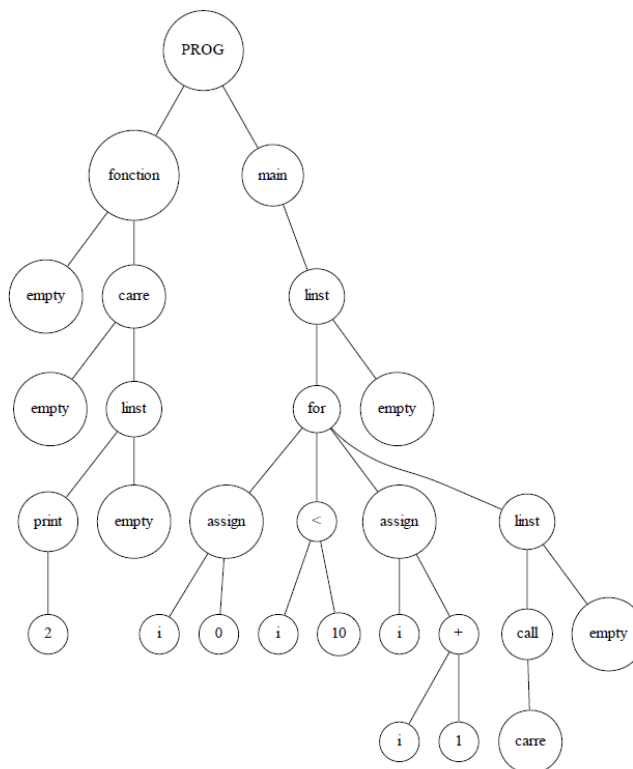
```
{'carre': ('empty', ('bloc', ('print', 2), 'empty'))}
```

```
{NOM : (paramètres, corps)}
```

- **appel des fonctions** : nouvelle instruction : CALL

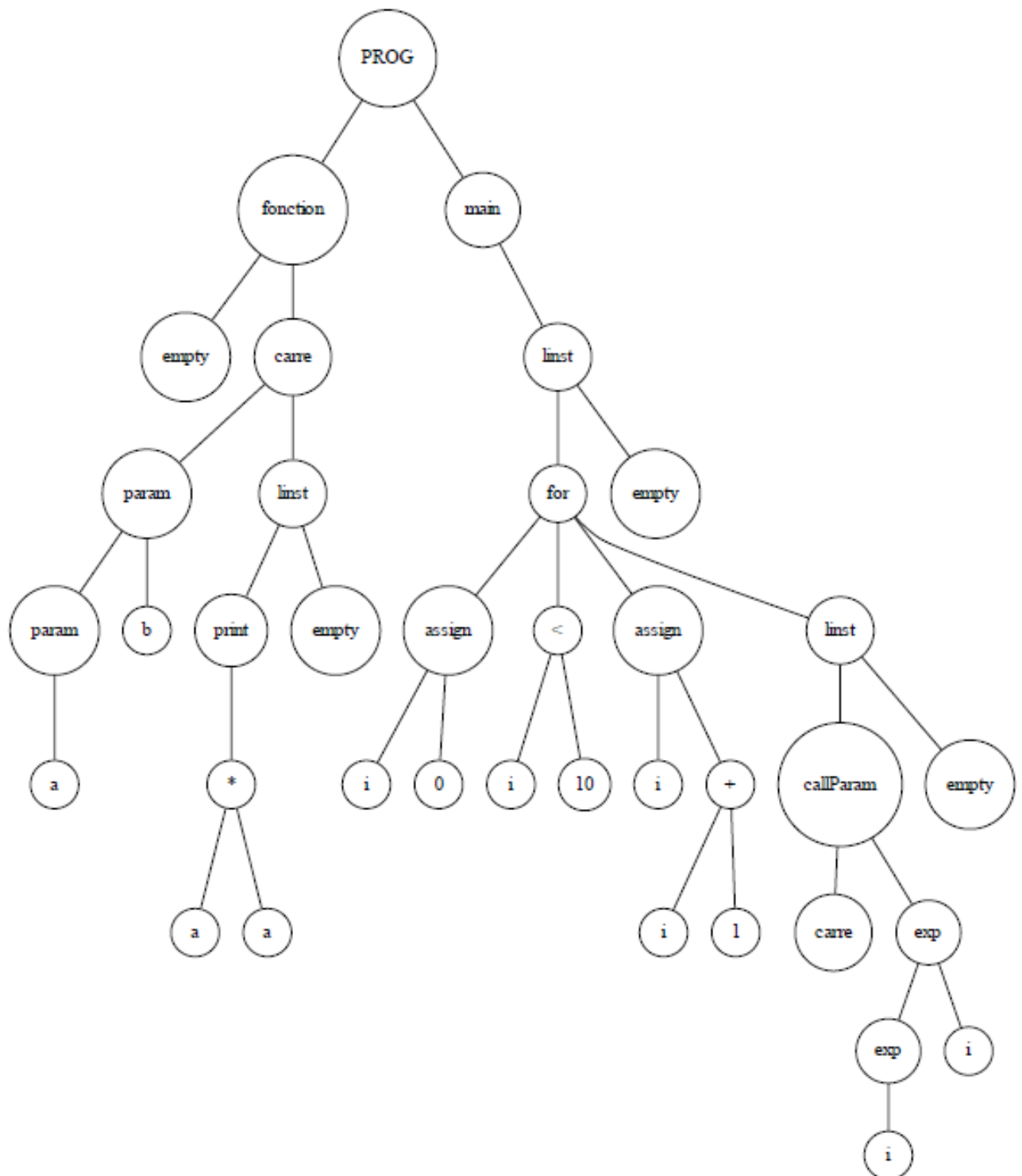
1. Fonctions void sans paramètre

s='fonction carre(){print(2);}for(i=0;i<10;i=i+1){carre();}'



2. Fonctions void avec paramètre

s='fonction carre(a,b){print(a*a);}for(i=0;i<10;i=i+1){carre(i, i);}'



3. Fonctions avec paramètres et return

Indications :

1. Stockage : Distinguer les fonctions *void* des fonctions *non void* : ajouter un attribut dans le dict functions ou faire 2 dict
2. AST et grammaire :

Distinguer les :

- FonctionValue et CallValue (evalExpr)
- FonctionVoid et CallVoid (evalInst)

3. Cas d'une valeur à retourner : plusieurs choix possibles

Méthode	Avec return explicite et forcé en fin de bloc	Avec return explicite et coupe circuit	Avec return implicite : exemple la valeur se trouve dans une variable qui a le même nom que la fonction
Déclaration	FonctionValue toto(a, b){ c=a+b; Return c; }	FonctionValue toto(a, b){ c=a+b; Return c; Print(1); }	FonctionValue toto(a, b){ toto=a+b; Print(1); }
Console suite à print(toto(1,2));	3	3 et pas de 1	1 et 3

4. Gestion du scope des variables

```
FonctionVoid f(a){print(a+1);return ;}
```

```
FonctionValue g(a){a=a+1 ;b=1 ;f(a*2) ;return a ;}
```

```
Main(){x=2 ;print(g(x)) ;}
```

Principe de la pile d'exécution

FonctionVoid f(a) { print(a+1) ; return ;} FonctionValue g(a) { a=a+1 ;b=1 ; f(a*2) ; return a ;} <u>Main()</u> { x=2 ; print(g(x)) ;}	FonctionVoid f(a) { print(a+1) ; return ;} FonctionValue g(a) { a=a+1 ; b=1 ; f(a*2) ; return a ;} Main(){ <u>x=2 ;</u> print(g(x)) ;}	FonctionVoid f(a) { print(a+1) ; return ;} <u>FonctionValue g(a)</u> { { a=a+1 ; b=1 ; f(a*2) ; return a ;} Main(){ x=2 ; print(<u>g(x)</u>) ;}	FonctionVoid f(a) { print(a+1) ; return ;} FonctionValue g(a) { <u>a=a+1 ;</u> b=1 ; f(a*2) ; return a ;} Main(){ x=2 ; print(g(x)) ;}	FonctionVoid f(a) { print(a+1) ; return ;} FonctionValue g(a) { a=a+1 ; b=1 ; f(a*2) ; return a ;} Main(){ x=2 ; print(g(x)) ;}	FonctionVoid f(a) { print(a+1) ; return ; } FonctionValue g(a) { a=a+1 ; b=1 ; f(a*2) ; return a ;} Main(){ x=2 ; print(g(x)) ;}
Main, {}	Main, {x :2}	g, {a :2} Main, {x :2}	g, {a :3, b :1} Main, {x :2}	f, {a :6} g, {a :3, b :1} Main, {x :2}	g, {a :3, b :1} Main, {x :2}

FonctionVoid f(a) { print(a+1) ; return ;} FonctionValue g(a) { a=a+1 ; b=1 ; f(a*2) ; return a ; } <u>Main()</u> { x=2 ; print(g(x)) ;}
Main, {x :2}

cf : https://fr.wikipedia.org/wiki/Pile_d%27ex%C3%A9cution

https://fr.wikipedia.org/wiki/Trace_d%27appels

4. Récursivité terminale

Dans une fonction récursive, si le return ne comprend que l'appel récursif (et aucune opération supplémentaire), on dit que la récursion est terminale

https://fr.wikipedia.org/wiki/R%C3%A9cursion_terminale

Suite de Fibonacci : fonction récursive

```
def fiboRec(n):  
    if n == 0 or n == 1 : return n  
    return fiboRec(n-1)+fiboRec(n-2)
```

```
print(fiboRec(9))
```

Suite de Fibonacci : fonction récursive terminale

```
def fibo(n, som, som2):  
    if n != 1:  
        return fibo(n-1, som+som2, som)  
    return som
```

```
print(fibo(9, 1, 0))
```

5. Variables globales

Principe : si on ne trouve pas une variable (une key du dict), on va chercher plus loin dans la pile