Université de Paris-Sud
Centre d'Orsay
DEUG MIAS
M3 — spécialité Informatique

Interprétation des langages de programmation

Jean-Pierre Jouannaud
Claude Marché
Ralf Treinen

Adresses des auteurs:

Jean-Pierre Jouannaud
L.R.I., Bâtiment 490
Université de Paris-Sud
91405 Orsay cedex
France

Tél : 01 69 15 69 05
Bureau 148
Email : Jean-Pierre.Jouannaud@lri.fr
WWW : http://www.lri.fr/Francais/Recherche/demons/membres/jouannau.html

Claude Marché
L.R.I., Bâtiment 490
Université de Paris-Sud
91405 Orsay cedex
France

Tél : 01 69 15 64 85
Bureau 118
Email : Claude.Marche@lri.fr
WWW : http://www.lri.fr/~marche/

Ralf Treinen
L.R.I., Bâtiment 490
Université de Paris-Sud
91405 Orsay cedex
France

Tél : 01 69 15 65 92
Bureau 152
Email : Ralf.Treinen@lri.fr
WWW : http://www.lri.fr/~treinen/

Table des matières

Introduction

La notion de langage est issue de la linguistique, qui est la science dont l'objet d'étude est le langage naturel, le français par exemple. Les linguistes cherchent à décrire les langues naturelles par des règles permettant d'engendrer les phrases correctes des langages en question. Ces mécanismes de description portent le nom familier de *grammaire*.

L'informatique a emprunté à la linguistique la notion de langage, ainsi que leur description par des grammaires. Les langages informatiques doivent toutefois satisfaire des critères contraignants auxquels les langages naturels ne sont pas assujettis, car les langages informatiques servent à décrire des calculs, et il est impératif qu'une phrase d'un langage informatique définisse un calcul bien précis, c'est-à-dire unique. Les langages naturels sont dits *ambigus*, à l'inverse des langages informatiques qui ne doivent pas l'être.

La notion de modèle de calcul est bien sûr au cœur de la discipline Informatique, même si elle fut à l'origine introduite par des mathématiciens dans les années 1930. Les modèles de calculs sont nombreux, il en existe au moins un par style de langage, impératif comme PASCAL, fonctionnel (on dit aussi applicatif) comme ML et ses dialectes, logique comme PROLOG, par contraintes comme CHIP, orienté objets comme JAVA. Ces modèles de calculs sont dits *universels* en ce sens qu'ils permettent de décrire n'importe quel calcul exécutable sur un ordinateur. La description et le calcul lui-même dépendent bien sûr du modèle utilisé.

Notre but dans ce cours n'est pas d'étudier les modèles de calcul précités, mais de comprendre quels sont les mécanismes permettant d'associer un calcul à un programme écrit dans un langage impératif comme PASCAL ou applicatif comme CAML. Il n'est pas non plus question d'aborder le mécanisme d'exécution d'un programme PASCAL ou CAML arbitraire, nous nous contenterons de fragments élémentaires de ces langages, et travaillerons à partir d'exemples simples, en particulier l'interprétation des expressions arithmétiques, qui a le mérite d'être proche du fonctionnement d'une calculette de poche de bas de gamme.

L'interprétation d'un programme se passe en plusieurs phases que nous considérerons comme étant bien distinctes de manière à en faciliter la description et la compréhension. Ces différentes phases sont visualisées sur la figure 1. Les deux premières phases sont dites *syntaxiques*, elles ont pour but de vérifier la bonne utilisation des règles de construction des programmes PASCAL ou ML ; alors que les autres phases sont dites *sémantiques*, elles ont pour but d'associer à chaque programme le calcul qu'il définit.

La première phase est *l'analyse lexicale*. La donnée de l'analyse lexicale est une suite de caractères composant un programme. Cette suite de caractères se présente comme un flot continu sans structure, on parle de flot de caractères. Le rôle de l'analyse lexicale est de reconnaître des groupes de caractères appelés *unités lexicales* par analogie avec les langues naturelles où les mots forment des unités lexicales (lexique est d'ailleurs un synonyme de dictionnaire). Pour les langages informatiques, les unités lexicales sont les noms de variables, les constantes numériques comme `3.141592`, les mots clés comme `while`, les opérateurs arithmétiques et logiques comme `+`, `or` ou `>=`. Le résultat de l'analyse lexicale est semblable à sa donnée, il s'agit d'un nouveau flot de caractères appelés *tokens*, dont les espacements ont en général été éliminés. L'analyse lexicale est en général faite à l'aide d'un outil originalement disponible sous les systèmes d'exploitation de la famille UNIX, le logiciel LEX. L'utilisation

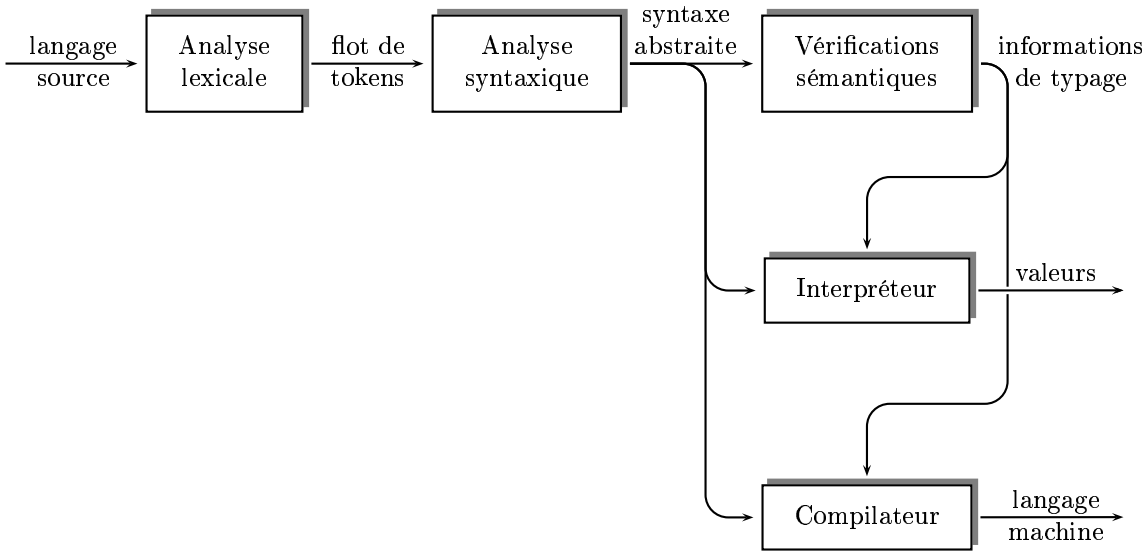


FIG. 1 – *Les différentes phases de l'interprétation d'un langage*

de LEX demande une connaissance superficielle des mécanismes mis en jeu, qui reposent sur les notions de grammaires régulières, automates et expressions rationnelles. Ces notions, et leur relations, seront abordées dans le cours par le biais de nombreux exemples simples.

La seconde phase est l'analyse syntaxique. La donnée est cette fois le flot de tokens résultat de l'analyse lexicale. Ce flot de tokens est transformé en un *arbre de syntaxe abstraite* qui décrit les différentes opérations à effectuer ainsi que leur enchaînement. L'analyse syntaxique est en général faite à l'aide d'un autre outil analogue à LEX, le logiciel YACC. L'utilisation de YACC demande une connaissance superficielle des mécanismes mis en jeu, qui reposent sur les notions de grammaire hors-contexte et d'analyse ascendante. Ces notions seront abordées dans le cours par le biais de l'exemple des expressions arithmétiques.

Les phases ultérieures ont pour but de *calculer* la signification d'un programme, on dit aussi sa *sémantique*, ce qui sera effectué à partir de l'arbre de syntaxe abstraite du programme, dont l'unicité sera donc une propriété essentielle. Ces calculs ont une structure très particulière, on parle d'homomorphisme, car ils respectent la structure de l'arbre de syntaxe abstraite. On dit aussi que la sémantique d'un programme est *compositionelle*, car la sémantique du tout (l'arbre) est une fonction de la sémantique de ses parties (les sous-arbres). Le formalisme utilisé pour décrire ces calculs est celui des *règles sémantiques*, un formalisme très général dont le but est de décrire des calculs s'effectuant sur des arbres. Ce formalisme trouve ses racines en logique mathématique, ce qui explique en partie le vocabulaire que nous utiliserons.

La troisième phase (vérifications sémantiques) opère de nombreux contrôles, en particulier des contrôles de types et des déclarations des variables préalablement à leur utilisation, ainsi que des calculs simples. Ces contrôles sont décrits par des règles sémantiques particulières appelées *jugements de typage*. Les informations obtenues lors de ces calculs pourront servir à l'interprétation elle-même. Nous nous contenterons d'effectuer des contrôles de type, ce que nous décrirons une fois de plus par le biais d'exemples simples pris en particulier dans le domaine du typage des expressions arithmétiques.

La quatrième phase est celle de l'interprétation elle-même, dont le principe est d'exé-

cuter pas à pas les instructions du programme telles qu'elles sont représentées dans l'arbre syntaxique, à partir des données fournies par l'utilisateur. Cette phase est en fait la plus complexe, car elle implique la mise en œuvre de mécanismes de gestion de la mémoire. Ces mécanismes dépendent de la catégorie de variables présentes dans le programme (variables locales ou globales, variables simples ou structurées, variables de taille fixes ou non, tous renseignements qui se trouvent dans la table des symboles), de la possibilité pour le programme d'appeler d'autres programmes ou de s'appeler lui-même récursivement, etc. La gestion de la mémoire comporte plusieurs aspects : construction de l'environnement de liaison des variables (plusieurs copies d'une même variable peuvent coexister, correspondant à plusieurs sections ou copies d'un même programme); récupération des zones de mémoire inutilisées (ramassage de miettes), et compactage de la mémoire si nécessaire afin d'en éviter la fragmentation. Ces deux derniers aspects sont généralement omis dans un interprète, c'est-à-dire traités automatiquement par les mécanismes correspondants du langage d'implantation. Cette phase peut être faite par des outils dans les cas simples, mais nécessite en tout cas une bonne compréhension des mécanismes mis en œuvre. Il est hors de question de traiter en détail les mécanismes complexes de gestion de la mémoire nécessités par un langage même aussi simple que PASCAL ou CAML. Nous nous contenterons une fois de plus d'étudier l'interprétation de sous-ensembles simplifiés de ces langages, contenant en particulier le langage des expressions arithmétiques.

La compilation est un mécanisme encore plus complexe que l'interprétation, dont le but est de traduire le programme du *langage source* vers le langage de la *machine cible* de manière à obtenir des performances élevées. L'interprétation est utilisée dans au moins trois circonstances distinctes : pour l'exécution de langages prototypes, dont la définition est encore sujette à modification ; pour l'exécution de langages d'interfaces, pour lesquels la rapidité n'est pas un problème crucial ; lors de la mise au point des programmes, les interprètes se prêtant mieux que les compilateurs à la conception d'outils de déverminage sophistiqués.

Grammaires et automates ne sont pas des outils spécifiques de l'interprétation (ou la compilation) de programmes informatiques, même si leur étude a été en grande partie historiquement motivée par cette application. Les grammaires sont bien sûr au cœur de la linguistique, comme nous l'avons déjà dit. De leur côté, les automates ont envahi de nombreux champs de la science informatique, et même au delà. Ils sont utilisés dans des domaines aussi variés que la preuve de correction de circuits, la recherche de mots dans un dictionnaire, le traitement automatisé des langues naturelles, le traitement automatisé des images, la modélisation de certaines formes de jeux, le modèle d'exécution des langages parallèles synchrones, la théorie des groupes dits automatiques, la théorie des nombres, la théorie des réseaux cristallins, le traitement du génome, et bien sûr l'automatique. Enfin, le langage étant un moyen de communication privilégié, des langages ad'hoc sont également utilisés dans un très grand nombre de domaines non scientifiques. Un exemple intéressant est fourni par la lecture de ce polycopié : chaque définition est introduite par le mot clé **Définition** écrit en caractères gras, chaque figure est pourvue d'un titre, les énumérations sont numérotées de manière consistante, etc. Ce polycopié suit donc un certain nombre de règles typographiques que l'on retrouve à des variantes près, dans tous les textes mathématiques. De fait, ce polycopié a été écrit en LaTeX, un langage spécialisé dans l'écriture de documents. Le compilateur LaTeX prend un texte écrit par l'utilisateur et le compile en un texte Postscript. Postscript est un autre langage, en fait le langage machine des imprimantes modernes, en particulier les imprimantes laser. L'exécution du code Postscript engendré produit donc comme résultat l'impression du document.

Un dernier exemple est le langage HTML (Hyper Text Markup Language). Ce n'est pas un langage de programmation mais un langage de description d'hypertexte, c'est-à-dire de pages de texte accompagné de « liens » pointant sur d'autres pages de texte. C'est dans ce langage que sont écrites les pages du *World Wide Web* qui peuvent être ainsi visualisées sur

n'importe quelle type d'ordinateur.

Le document est structuré en deux parties : syntaxe, puis sémantique.

La première partie comporte cinq chapitres. Le chapitre 1 décrit les notions de mots et de langage formel sur un alphabet, ainsi que les opérations importantes sur les mots et les langages utilisées dans la suite. Il introduit les grammaires, un formalisme génératif pour décrire des langages. Le chapitre 2 étudie plus à fond les grammaires régulières et le mécanisme de reconnaissance des langages associés, les automates. Il débouche sur l'algorithme de minimisation des automates déterministes. Le chapitre 3 continue avec l'étude des expressions rationnelles, un outil commode de description des langages réguliers, et son application à l'analyse lexicale. Il se conclut avec une description rapide du logiciel LEX. Le chapitre 4 continue avec une étude plus approfondie des grammaires générales, en introduisant les notions de dérivations et d'arbre syntaxique, et débouche sur les principes généraux d'analyse syntaxique, et leur application dans le cadre du logiciel YACC. Enfin, le chapitre 5 décrit la notion de syntaxe abstraite, qui est à la base de la description de la sémantique des langages telle que nous l'exposons dans la seconde partie du document. Nous y étudions également comment calculer la syntaxe abstraite d'un langage à l'aide de YACC.

La seconde partie comporte cinq chapitres. Le chapitre 1, décrit la notion de règle sémantique, un formalisme adapté à la description de la sémantique des langages, de même que les grammaires sont adaptées à la description de leur syntaxe. Pour chaque langage traité, nous étudions successivement deux aspects : le typage, et l'évaluation, qui donnent lieu à ce que l'on appelle des jugements de typage et des jugements d'évaluation. Le chapitre 2 traite du langage des expressions arithmétiques et booléennes, et introduit les règles de typage. Le chapitre 3 traite les définitions de variables et donc les problèmes de portée, en introduisant la notion fondamentale *d'environnement*. Le chapitre 4 traite les expressions fonctionnelles, dans le cas particulier simple des fonctions à un unique argument, et aborde les problèmes liés à la récursivité. Enfin, le chapitre 5 décrit un langage fonctionnel simple qui réunit les divers aspects abordés séparément dans les chapitres précédents.

La bonne compréhension de ce cours nécessite une pratique rudimentaire des langages PASCAL et CAML ou des langages analogues, ainsi que la connaissance des notions de graphe et d'arbre étiqueté. Sinon, le cours ne demande aucune connaissance mathématique particulière. En particulier, et c'est délibéré, il ne contient aucune preuve des théorèmes ou propriétés énoncés. Le lecteur familier avec le raisonnement mathématique est toutefois invité à suppléer à ce manque.

Exercice 0.1
Trouver d'autres exemples d'utilisation de langages ad'hoc, dans les sciences traditionnelles (biologie, chimie, ...) ou dans des domaines non-scientifiques.
Correction : *Le code génétique en biologie, c'est bon ? La notation* $C_5H_{17}O_{bidule}$ *que je sais plus comment ca s'appelle en chimie, c'est bon aussi ? La langue de bois en politique ?*

Première partie

Syntaxe

Chapitre 1

Notion de langage

Le but premier de ce chapitre est définir la notion de langage. Cette notion est très simple, et commune à tous les champs disciplinaires qui l'utilisent. Un langage est tout simplement un ensemble fini ou infini de mots formés sur un certain alphabet, qui pour nous sera en général le jeu de caractères ASCII ou l'un de ses sous-ensembles. Le second but est d'introduire un procédé permettant de descrire des langages, les grammaires.

1.1 Ensemble des mots sur un alphabet donné

Définition 1.1 *Étant donné un ensemble fini V appelé* vocabulaire *ou* alphabet *dont les éléments sont appelés des* symboles *ou* lettres*, on appelle* mot *sur V toute suite finie $u = \{\alpha_i\}_{i\in[1..n]}$ d'éléments de V. n s'appelle la* longueur *du mot u et est notée $|u|$.*

D'une manière générale, dans ce cours, nous utiliserons des lettres grecques pour dénoter des symboles, et les lettres u, v et w pour dénoter des mots.

La notation d'un mot sous forme d'une suite n'étant pas très pratique, un mot $u = \{\alpha_i\}_{i\in[1..n]}$ sera en général noté simplement par juxtaposition de ses lettres : $u = \alpha_1 \cdots \alpha_n$, et le mot vide (de longueur nulle) sera alors noté par le symbole ε.

L'ensemble des mots sur V est muni d'une opération binaire de composition interne appelée *produit de concaténation*. Le produit de concaténation de deux mots u et v, noté $u \times v$ ou en abrégé uv (la notation $u \cdot v$ est aussi utilisée), est défini comme suit :

$$\{\alpha_i\}_{i\in[1..m]} \times \{\beta_i\}_{i\in[1..n]} = \{\gamma_i\}_{i\in[1..m+n]} \text{ tel que } \begin{cases} \gamma_i = \alpha_i, \text{ pour tout } i \in [1..m] \\ \gamma_{m+i} = \beta_i, \text{ pour tout } i \in [1..n] \end{cases}$$

Le produit de concaténation est bien sûr associatif : $(uv)w = u(vw)$, et possède un élément neutre, le mot vide : $u\varepsilon = \varepsilon u = u$.

Le produit de concaténation nous permet de définir par récurrence les *puissances* d'un mot u:

$$u^0 = \varepsilon, \quad u^{p+1} = u.u^p$$

La puissance p fois du mot $u = \{\alpha_i\}_{i\in[1..n]}$ est donc le mot $v = \{\beta_i\}_{i\in[1..np]}$ de longueur np tel que $\beta_{kn+i} = \alpha_i$, pour tout $k \in [0..p-1]$ et $i \in [1..n]$, ou encore, avec la notation par juxtaposition :

$$(\alpha_1 \cdots \alpha_n)^p = \underbrace{(\alpha_1 \cdots \alpha_n) \times \cdots \times (\alpha_1 \cdots \alpha_n)}_{p \text{ fois}}$$

1.2 Langages

Définition 1.2 *On appelle* langage *sur un vocabulaire V tout ensemble de mots sur V.*

Les opérations fondamentales sur les langages sont :

- le produit de concaténation : $L_1 \times L_2 = \{u \times v \mid u \in L_1,\ v \in L_2\}$, noté aussi L_1L_2 en abrégé ;
- la puissance : $L^0 = \{\varepsilon\}$ et $L^{p+1} = L \times L^p$ c'est-à-dire $L^p = \{u_1 \cdots u_p \mid u_1, \ldots, u_p \in L\}$;
- *l'itéré* : $L^* = \bigcup_{p \geq 0} L^p$ c'est-à-dire $L^* = \{u_1 \cdots u_p \mid p \geq 0, u_1, \ldots, u_p \in L\}$;
- *l'itéré strict* : $L^+ = \bigcup_{p > 0} L^p$ c'est-à-dire $L^+ = \{u_1 \cdots u_p \mid p \geq 1, u_1, \ldots, u_p \in L\}$.

L'opération d'itération est encore appelée *étoile de Kleene* du nom de son inventeur. Il est facile de voir que l'itéré de V est l'ensemble de tous les mots, ce dernier sera donc naturellement noté par V^*.

Les langages étant des ensembles, ils en héritent également les opérations booléennes habituelles :

- l'union : $L \cup L' = \{u \mid u \in L \text{ ou } u \in L'\}$;
- l'intersection : $L \cap L' = \{u \mid u \in L \text{ et } u \in L'\}$;
- le complémentaire : $\overline{L} = \{u \mid u \in V^* \text{ et } u \notin L\}$.

La question fondamentale qui va nous préoccuper dans ce cours, directement ou indirectement, est celle de la description d'un langage donné. La difficulté bien entendu, est qu'il faut pouvoir décrire des langages infinis, puisqu'il y une infinité de programmes possible dans un langage de programmation. On distingue 3 grandes méthodes de description :

1. La méthode *générative* a pour principe la donnée d'un algorithme permettant d'engendrer l'ensemble des phrases du langage à décrire à partir de ce que l'on appelle une *grammaire*. Il s'agit donc en fait d'une description en extension adaptée au cas des langages infinis.
2. La seconde méthode consiste à donner un algorithme permettant de décider si une phrases donnée appartient ou pas au langage, à l'aide de ce que l'on appelle un *automate* (plus généralement une machine de Türing).
3. La dernière méthode, de nature algébrique, consiste à donner une notation spécifique, les *expressions rationnelles* , permettant de représenter un langage simple par une expression algébrique.

Nous aurons l'occasion de rencontrer ces trois types de méthodes. Dans la suite de ce chapitre, nous allons brièvement introduire la premiere.

1.3 Notion intuitive de grammaire

Les grammaires sont des outils pour décrire des langages, inspirés des grammaires utilisées pour les phrases du langage naturel, comme par exemple :

> *Une phrase simple est constituée d'un sujet suivi d'un verbe suivi d'une préposition suivie d'un complément. Le sujet est un prénom ou bien un pronom. Les prénoms sont à choisir parmi « Alain », « Béatrice » et « Charles ». Les pronoms sont à choisir parmi « Il » et « Elle ». Les prépositions sont à choisir parmi « à », « de », « au » et « en ». Les verbes sont à choisir parmi « va » et « vient ». Les compléments sont à choisir parmi « l'école » et « marché ».*

« *Charles va à l'école* » est un exemple de phrase simple au sens ci-dessus. Cette description fait ressortir plusieurs phénomènes :

1. Le français est utilisé pour sa propre description, ce qui nécessite l'utilisation de guillemets afin de différencier le français en tant que langage de description du français en temps que langage décrit.
2. Le français n'est pas adapté en temps que langage de description, pas plus que tout autre langue naturelle, car trop imprécis. La première phrase de la description, en particulier, a plusieurs interprétations suivant que « suivi d'une préposition » se rapporte à « verbe » ou à « sujet suivi d'un verbe ».
3. les concepts utilisés sont en tout petit nombre : « est constitué », « suivi de », « est », « ou », « à choisir », et la possibilité de nommer.

Tout cela milite en faveur d'une notation spécialisée simple dotée d'une notion de variable. Cette notation consiste en fait à décrire un langage par des règles de grammaire. Pour notre exemple, on obtient :

$$
\begin{array}{rcl}
\textit{Phrase-simple} & \rightarrow & \textit{Sujet}\ \ \textit{Verbe}\ \ \textit{Préposition}\ \ \textit{Complément} \\
\textit{Sujet} & \rightarrow & \textit{Prénom} \mid \textit{Pronom} \\
\textit{Prénom} & \rightarrow & \text{Alain} \mid \text{Béatrice} \mid \text{Charles} \\
\textit{Pronom} & \rightarrow & \text{Il} \mid \text{Elle} \\
\textit{Verbe} & \rightarrow & \text{va} \mid \text{vient} \\
\textit{Préposition} & \rightarrow & \text{à} \mid \text{de} \mid \text{au} \mid \text{en} \\
\textit{Complément} & \rightarrow & \text{l'école} \mid \text{marché}
\end{array}
$$

On peut tout aussi bien décrire des langages informatiques, puisque tel est l'un de nos buts. Voici par exemple une grammaire décrivant les expressions arithmétiques telles qu'on les trouve dans la plupart des langages de programmation (on se limite aux opérations d'addition et de multiplication) :

$$
\begin{array}{rcl}
\textit{Expression} & \rightarrow & \textit{Expression} + \textit{Expression} \mid \textit{Expression} \times \textit{Expression} \mid \textit{Entier} \\
\textit{Entier} & \rightarrow & \textit{Chiffre} \mid \textit{Chiffre-non-nul}\ \ \textit{Chiffres} \\
\textit{Chiffres} & \rightarrow & \textit{Chiffre} \mid \textit{Chiffre}\ \ \textit{Chiffres} \\
\textit{Chiffre} & \rightarrow & 0 \mid \textit{Chiffre-non-nul} \\
\textit{Chiffre-non-nul} & \rightarrow & 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

On verra par la suite qu'une grammaire peut présenter des ambiguités qu'il faudra alors éliminer. Dans la suite de ce chapitre, nous nous contenterons en fait de donner la définition de la notion de grammaire, tout en rejetant leur étude et leur emploi aux chapitres ultérieurs.

1.4 Définition formelle des grammaires

Comme on vient de le voir, toute suite de lettres de l'alphabet latin n'est pas une phrase en français. Il en est de même des langages informatiques : une suite de lettres est un programme si elle respecte certaines règles bien précises. La description de ces règles se fait à l'aide d'une *grammaire*. De la discussion précédente, on peut dire qu'une grammaire est donnée par un ensemble de règles associant à un certain composant du langage un choix possible pour ses constituants.

Définition 1.3 *Une* grammaire G *est un quadruplet* (V_t, V_n, S, R) *où*

1. V_t *est un ensemble fini de symboles dits* terminaux, *appelé* vocabulaire terminal ;

$$\begin{array}{lcl} N & \to & 0 \\ N & \to & 1 \\ N & \to & 1M \\ M & \to & 0 \\ M & \to & 1 \\ M & \to & 0M \\ M & \to & 1M \end{array}$$

$$\begin{array}{lcl} N & \to & 0 \mid 1 \mid 1M \\ M & \to & 0 \mid 1 \mid 0M \mid 1M \end{array}$$

$$\begin{array}{lcl} N & \to & 0 \\ N & \to & 1M \\ M & \to & \varepsilon \\ M & \to & 0M \\ M & \to & 1M \end{array}$$

$$\begin{array}{lcl} N & \to & 0 \mid 1M \\ M & \to & \varepsilon \mid 0M \mid 1M \end{array}$$

FIG. 1.1 – *Deux grammaires décrivant les entiers écrits en binaire*

2. *V_n est un ensemble fini (disjoint de V_t) de symboles dits* non-terminaux*, appelé* vocabulaire non terminal *;*

3. *$V = V_n \cup V_t$ est appelé le* vocabulaire *de la grammaire G ;*

4. *S est un non-terminal particulier appelé* source*, ou* axiome*, ou encore* symbole d'entrée *;*

5. *R est un ensemble fini de* règles de grammaires *de la forme $N \to u$, où $N \in V_n$ et $u \in V^*$*

Dans ce cours, on utilisera en général des mots en italique commençant par une majuscule pour les non-terminaux, et des lettres minuscules ou des symboles spéciaux (+, ×, etc.) pour les terminaux.

> *Remarque : la notion de grammaire formelle est en fait plus générale que celle que nous venons de définir qui suffit pour les besoins informatiques, mais pas par exemple pour les besoins de la linguistique : la définition ci-dessus définit en fait les grammaires « hors-contexte », mais comme nous n'en rencontrerons pas d'autres, nous omettrons ce qualificatif.*

À titre d'exemple, nous nous proposons de décrire l'ensemble des entiers naturels en représentation binaire, vus comme des suites non vides de 0 et de 1 ayant la propriété que seule la suite 0 peut commencer par un 0. Pour cela, nous allons utiliser une grammaire pour laquelle l'ensemble des symboles non-terminaux est $\{ N , M \}$, la source est le non-terminal N, l'ensemble des symboles terminaux est $\{0, 1\}$ et les règles sont décrites figure 1.1. Ce tableau présente en fait quatre ensembles de règles, dont le premier et le troisième sont conformes à la définition, alors que les règles de même membre gauche ont été regroupées dans le second et le quatrième en séparant les différents membres droits par le symbole « | » signifiant « ou », conformément à une convention usuelle : $N \to u_1 \mid \cdots \mid u_n$ abrège l'ensemble des règles $N \to u_1, \ldots, N \to u_n$. Les quatre ensembles de règles de la figure 1.1 décrivent le même langage, celui des nombres entiers naturels écrits en binaire, mais le troisième est plus compact que le premier grâce à l'utilisation du symbole ε. Le premier et le second ensemble de règles doivent être considérés comme identiques, de même que le troisième et le quatrième, car ils ne diffèrent que par l'économie d'écriture juste mentionnée.

En bon français, il faudrait dire *récrivant*

Une grammaire est un mécanisme permettant *d'engendrer* les phrases du langage, en *réécrivant* un mot $u \in V^*$ en un nouveau mot $v \in V^*$ par l'opération consistant à remplacer une occurrence (quelconque) d'un non-terminal N (quelconque) de V_n présent dans u par un membre droit de règle dont N est membre gauche. Ce processus est itéré à partir de la source jusqu'à l'élimination complète des non-terminaux. Par exemple, en utilisant la première grammaire définie dans la figure 1.1 :

$$N \to 1M \to 11M \to 110M \to 1101M \to 11010$$

et en utilisant la deuxième 1.1 :

$$N \to 1M \to 11M \to 110M \to 1101M \to 11010M \to 11010$$

Définition 1.4 *Étant donnée une grammaire $G = \{V_t, V_n, S, R\}$, on dit que le mot $u \in V^*$ se* réécrit *en le mot $v \in V^*$ dans G, et l'on note $u \to v$, si*

1. *$u = w_1 N w_2$, où $w_1 \in V^*$, $N \in V_n$ et $w_2 \in V^*$;*
2. *$v = w_1 w w_2$, où $w \in V^*$;*
3. *$N \to w$ est une règle de R.*

On dit que le mot $v \in V^*$ dérive *du mot $u \in V^*$, dans la grammaire G, ce que l'on note par $u \to^* v$, s'il existe une suite finie $w_0, w_1, \ldots, w_n$ de mots de V^* telle que $w_0 = u$, $w_i \to w_{i+1}$ pour tout $i \in [0, n-1]$, et $w_n = v$.*

Nous avons vu ci-dessus que le mot $11010M$ dérive du mot $1M$ dans la troisième grammaire de la figure 1.1. Par convention, nous indiquerons le non-terminal réécrit dans un mot en le soulignant, comme dans

$$1\underline{M} \to 11\underline{M} \to 110\underline{M} \to 1101\underline{M} \to 11010M$$

Définition 1.5 *Le* langage engendré *par la grammaire G est l'ensemble des mots de V_t^* qui dérivent de l'axiome de G, que l'on note par $\mathcal{L}ang(G)$.*

Il est aisé de se convaincre que l'ensemble des mots qui appartiennent aux langages engendrés par les grammaires de la figure 1.1 est dans les deux cas l'ensemble des entiers naturels écrits en binaire, c'est-à-dire les suites de 0 et de 1 ne commençant pas par 0, sauf la suite 0 elle-même.

1.5 Exercices

Exercice 1.1
Soit l'alphabet $V = \{a, b\}$ et les mots $u = ab$ et $v = aba$. Calculer uv, vu, u^2, u^3 et v^2.
Correction : *$uv = ababa$, $vu = abaab$, $u^2 = abab$, $u^3 = ababab$, $v^2 = abaaba$.*

Exercice 1.2
Montrer que pour tout mot $u \in V^*$ et tous entiers $p, q \in \mathbb{N}$ on a $u^p u^q = u^{p+q} = u^q u^p$.

Exercice 1.3
Considérons les deux langages sur l'alphabet $\{a, b\}$ définis par $L_1 = \{a, ba\}$ et $L_2 = \{\varepsilon, b, aa\}$.
Calculer L_1L_2, L_2L_1, L_1^2, L_2^2, L_1^3, L_2^3, L_1^*, L_2^*.
Correction :

$$\begin{aligned}
L_1L_2 &= \{a, ba, ab, bab, aaa, baaa\}\\
L_1L_2 &= \{a, ba, ba, bba, aaa, aaba\}\\
L_1^2 &= \{aa, aba, baa, baba\}\\
L_2^2 &= \{\varepsilon, b, aa, bb, baa, aab, aaaa\}\\
L_1^3 &= \{aaa, abaa, baaa, babaa, aaba, ababa, baaba, bababa\}\\
L_2^3 &= \{\varepsilon, b, aa, bb, baa, aab, aaaa, bbb, baab, aabb, aaaab, bbaa, baaaa, aabaa, aaaaaa\}
\end{aligned}$$

L_1^* est l'ensemble des mots sur $\{a, b\}$ tels qu'un b est toujours suivi d'un a, L_2^* est l'ensemble des mots sur $\{a, b\}$ tels que les a apparaissent toujours par deux.

Exercice 1.4

Correction : *Le but du premier exercice est de familiariser les étudiants avec la manipulation des langages. Les questions 1 et 2 de cet exercice sont étroitement imbriquées et, on peut les traiter en même temps pour chacun des langages.*

On considère les langages définis sur le vocabulaire $V = \{a, b\}$ par $L_1 = \{a\}^*.\{b\}^*$, $L_2 = \{\{a\} \cup \{b\}\}^*$, $L_3 = \{ab\}^*$ et $L_4 = \{a\}^* \cup \{b\}^*$.

1. Trouver la forme générale des mots de chacun de ces langages.

   Correction :

   - *L_1 est l'ensemble des mots de la forme $a^n b^m$, $n, m \geq 0$.*
   - *L_2 est l'ensemble des mots formés a l'aide des lettres de V soit V^*.*
   - *L_3 est l'ensemble des mots de la forme $\{ab\}^n$, $n \geq 0$.*
   - *L_4 est l'union des mots de la forme a^n, $n \geq 0$ et des mots de la forme b^n, $n \geq 0$.*

2. Les mots suivants appartiennent-ils à chacun de ces langages: a, b, aa, ab, ba, $abab$.

   Correction : *La réponse est évidente pour le langages L_2 qui contient tous les mots formé à l'aide des lettres du vocabulaire V.*

   Le tableau suivant résume les dérivations qui permettent de décider lorsque le mot appartient au langages

	a	b	aa	ab	ba	$abab$
L_1	a^1b^0	a^0b^1	a^2b^0	a^1b^1	*Non*	*Non*
L_3	*Non*	*Non*	*Non*	$\{ab\}^1$	*Non*	$\{ab\}^2$
L_4	a^1	b^1	a^2	*Non*	*Non*	*Non*

   Les résultats négatifs sont les plus difficiles justifier. La non appartenance de ba et $abab$ à L_1 est due au fait que dans un mots du langage L_1 un a ne peut apparaitre après un b. Le mots de L_3 sont une alternance stricte de lettre a et de lettre b commencant par un a. Les mots a, b, aa et ba ne répondent pas à cette définition. Enfin Les mots de L_4 ne contiennent que des a ou que des b. Les mots qui contiennent des a et des b n'appartiennent donc pas à L_4.

3. Comparez ces langages (égalités, inclusions).

   Correction :

   - *L_2 et L_*. Le langage L_2 est l'ensemble de tous les mots sur l'alphabet $\{a, b\}$. Il contient donc les langages L_1, L_3 et L_4. Nous avons donc $L_2 \supseteq L_1$, $L_2 \supseteq L_3$, $L_2 \supseteq L_4$.*
   - *L_1 et L_3. Nous avons qu'il existe au moins un mot (aa) qui appartient à L_1 et pas à L_3 et un mot $(abab)$ qui appartient à L_3 et pas à L_1. La comparaison entre L_3 et L_4 est elle aussi vite faite à l'aide des mots a et ab. En fait les seuls mots commun à ces deux langages sont ε et ab.*
   - *L_1 et L_4. Nous avons le mot ab qui appartient à L_1 mais pas à L_4. Par contre tous les mots de L_4 sont aussi des mots de L_1: Pour un mot de L_4 il y a deux possibilités. Soit ce mot est de la forme a^n avec $n \geq 0$ et dans ce cas il est aussi dans le langage L_1 sous la forme $a^n.b^0$. Soit ce mot est de la forme b^n avec $n \geq 0$ et dans ce cas il est aussi dans le langage L_1 sous la forme $a^0.b^n$. Nous avons donc $L_4 \subseteq L_1$.*

Exercice 1.5

Montrer que pour tous langages L, L_1, L_2 et L_3 sur un vocabulaire V on a

1. $(L_1 \cup L_2)L_3 = (L_1L_3) \cup (L_2L_3)$;
2. $L_1(L_2 \cup L_3) = (L_1L_2) \cup (L_1L_3)$;
3. $L^* = L^+ \cup \{\varepsilon\}$.

4. $(L^*)^* = L^*$.

Exercice 1.6
On considère le langage L_0 sur l'alphabet $V_t = \{a, b, c, d\}$ défini par la grammaire suivante :

$$L \quad \rightarrow \quad a \mid b \mid cL \mid dLL$$

1. Montrer que *ca*, *dba*, *cdccab* sont dans le langage L_0.

   Correction : *Pour cette question, il suffit d'appliquer les règles de définition. Par la règle (1), $a \in L_0$, donc par (3), $ca \in L_0$. Par (2), $b \in L_0$ et $a \in L_0$ donc par (4), $dba \in L_0$. On a vu que $ca \in L_0$, donc par (3), $cca \in L_0$, et par (2) $b \in L_0$ donc par (4), $dccab \in L_0$, donc par (3) $cdccab \in L_0$.*

2. Trouver tous les mots de L_0 d'au plus quatre lettres.

   Correction : *1 lettre : a, b. 2 lettres : ca, cb. 3 lettres : cca, ccb, daa, dab, dba, dbb. 4 lettres : ccca, cccb, cdaa, cdab, cdba, cdbb, daca, dacb, dbca, dbcb, dcaa, dcab, dcba, dcbb.*

3. Donner une méthode pour déterminer l'ensemble E_n des mots de L_0 ayant n lettres, à partir des mots de moins de n lettres. En déduire une formule de récurrence entre les E_n.

   Correction : *La méthode générale pour déterminer tous les mots de L_0 de n lettres est la suivante: Si $n = 1$ alors on a a et b. Si $n > 1$ alors de prendre tous les mots de $n-1$ lettres et d'ajouter un c devant, et pour chaque i entre 1 et $n-2$, de chercher tous les mots w_1 à i lettres et les mots w_2 à $n-i-1$ lettres et de construire dw_1w_2.*

   Nous avons donc la formule suivante mettant en relation les langages $E_i = \{u \in L_0 \mid |u| = n\}$:

   $$\begin{aligned} E_1 &= \{a, b\} \\ E_{n+1} &= cE_n \cup \bigcup_{i=1}^{n-2} dE_iE_{n-i-1} \end{aligned}$$

   Ces formules donnent immédiatement une fonction récursive qui étant donné un entier n calcule l'ensemble E_n.

4. En déduire une méthode pour décider si un mot est dans L_0 ou non. Que peut-on dire de la complexité des calculs effectués par cette méthode?

   Correction : *Une méthode pour décider si un mot u appartient à L_0 peut consister en tester l'appartenance de u à $E_{|u|}$. Ceci donne un programme très complexe en occupation mémoire et par conséquent en temps également.*

5. Montrer que *adccba* n'est pas dans le langage L_0. On pourra raisonner sur la première lettre de ce mot.

   Correction : *Montrer qu'un mot n'appartient pas au langage est bien sûr beaucoup plus difficile. Pour montrer que ce mot n'est pas dans le langage, nous proposons de montrer que pour tout mot $w \in L_0$ de longueur au moins 2, la première lettre de w est un c ou un d.*

   En effet, si $w \in L_0$ est de longueur au moins 2, par la forme de la dernière production de la grammaire il doit exister soit un mot $w' \in L_0$ tel que $w = cw'$, soit deux mots w_1 et w_2 tels que $w = dw_1w_2$. Dans les deux cas la première lettre de w est c ou d

6. Montrer que *cdbadc* n'est pas dans le langage L_0. On pourra raisonner sur la dernière lettre de ce mot.

   Correction : *Nous proposons de montrer que pour tout mot $w \in L_0$, la dernière lettre de w est un a ou b. On procède par récurrence sur la longueur de w. (Une alternative est de procéder par récurrence sur la construction du mot.)*

cas $n = 1$: les mots de longueur 1 de L_0 sont a et b et vérifient la propriété voulue.

cas général : si $w \in L_0$ est de longueur au moins 2, par la forme de la dernière production de la grammaire il doit exister soit un mot $w' \in L_0$ tel que $w = cw'$, soit deux mots w_1 et w_2 tels que $w = dw_1w_2$. Par récurrence sur w' ou w_2, la dernière lettre est un a ou un b.

7. Montrer la propriété suivante : si $w_1v_1 = w_2v_2$ avec w_1, w_2 dans L_0, alors $w_1 = w_2$ et $v_1 = v_2$. En déduire que, si $w = w_1v_1$ avec w et w_1 dans L_0, alors $v_1 = \varepsilon$.

   Correction : *Pour la deuxième propriété on prend $w_2 = w$ et $v_2 = \varepsilon$.*

   On montre la première propriété par récurrence sur la longueur n de w_1 :

   cas $n = 0$: n'est pas possible parce que $\varepsilon \notin L_0$.

   cas $n = 1$: alors $w_1 = a$ ou b, et nécessairement $w_1 = a$ resp. $w_2 = b$.

   cas $n > 1$ et w_1 commence avec c : On a $w_1 = cw_1'$ avec $w_1' \in L_0$, et également $w_2 = cw_2'$ avec $w_2' \in L_0$. Donc, $cw_1'v_1 = cw_2'v_2$, alors $w_1'v_1 = w_2'v_2$. Par récurrence, $w_1' = w_2'$, donc $w_1 = w_2$.

   cas $n > 1$ et w_1 commence avec d: On a $w_1 = dw_1'w_1''$ avec $w_1', w_1'' \in L_0$, et également $w_2 = dw_2'w_2''$ avec $w_2', w_2'' \in L_0$. Donc, $dw_1'w_1''v_1 = dw_2'w_2''v_2$, alors $w_1'w_1''v_1 = w_2'w_2''v_2$. Par récurrence, $w_1' = w_2'$, donc $w_1''v_1 = w_2''v_2$. On applique la récurrence à la dernière équation et obtient $w_1'' = w_2''$. En résumé, on a $w_1'w_1'' = w_2'w_2''$, c'est-à-dire $w_1 = w_2$.

   On peut remarquer l'utilisation de l'associativité de la concaténation, et qu'il faut appliquer la récurrence deux fois.

8. Soit x_n le nombre de mots à n lettres de L_0. Déterminer une relation de récurrence satisfaite par la suite $(x_n)_{n\in\mathbb{N}}$.

   Correction : *On remarque que grâce à la propriété précédente, la méthode de la question 3 ne peut pas engendrer un mot de deux manières différentes : si $w_1w_2 \in L_0$ et $w_1'w_2' \in L_0$ alors $w_1 = w_1'$ et $w_2 = w_2'$.*

   Ainsi, on a $x_0 = 0$, $x_1 = 2$ et

$$x_n = x_{n-1} + \sum_{i=1}^{n-2} x_i \times x_{n-i-1}$$

   pour $n \geq 2$.

9. Montrer que si w est dans L_0, alors $|w|_a + |w|_b - |w|_d = 1$. Trouver un mot w, commençant par c ou d et finissant par a ou b, ne vérifiant pas cette propriété. Ce mot est-il dans L_0 ? Existe-il un mot w commençant par c ou d et finissant par a ou b, vérifiant la propriété précédente, mais n'étant pas dans L_0 ?

   Correction : *Encore par récurrence:*

   cas $n = 1$: $|a|_a + |a|_b - |a|_d = |b|_a + |b|_b - |b|_d = 1$.

   cas général : si $w \in L_0$ est de longueur au moins 2, par la forme de la dernière production de la grammaire il doit exister soit un mot $w' \in L_0$ tel que $w = cw'$, soit deux mots w_1 et w_2 tels que $w = dw_1w_2$. Dans le premier cas, $|w|_a + |w|_b - |w|_d = |cw'|_a + |cw'|_b - |cw'|_d = |w'|_a + |w'|_b - |w'|_d = 1$ par récurrence. Dans le second cas, $|w|_a + |w|_b - |w|_d = |dw_1w_2|_a + |dw_1w_2|_b - |dw_1w_2|_d = |w_1|_a + |w_2|_a + |w_1|_b + |w_2|_b - 1 - |w_1|_d - |w_2|_d = 1 + 1 - 1 = 1$.

   $w = caa$ commence bien par c ou d et finit bien par a ou b, et $|w|_a + |w|_b - |w|_d = 2$, il répond donc à la première question. Il n'est bien sûr pas dans L_0 puisqu'il ne vérifie la propriété.

   cadb répond à la deuxième question (on a vu a la question 2 qu'il n'était pas dans L_0).

   Cette question et les précédentes doivent montrer aux étudiants la difficulté de trouver une méthode pour décider si un mot est dans le langage ou pas.

Correction : *Attention pour le groupe S3, ceux-ci ne font pas l'étude de cas en parallèle et donc ne bénéficient pas des compléments de CAML. C'est pourquoi quelques compléments sont présentés au cours du TP. Avantage néanmoins pour les S3 : ils ont fait du CAML au semestre précédent donc on peut supposer qu'ils s'en souviennent mieux. Attention, dans le cours de CAML de M1, les étudiants ont appris a ecire les fonctions uniquement sous la forme*

```
let f = function(x1,..,xn) -> ...
```

Il faudra les habituer tres progressivement a ecrire

```
let f = function x1 .. xn -> ...
```

ou même

```
let f x1 .. xn =
```

Exercice 1.7
D'une manière générale, un analyseur syntaxique pour un langage L est une fonction dont la spécification est la suivante :

`analyse`$_L$: chaîne de caractères $\rightarrow$ booléen
{ `analyse`$_L(w)$ retourne `vrai` si le mot w appartient au langage L, `faux` sinon }

On reprend l'exercice 1.6. Nous allons écrire une fonction `analyse`$_{L_0}$ par trois méthodes différentes.

Méthode 1 : par énumération des mots du langage.

1. Écrire une fonction `prefixe` qui, étant donnés un mot w et une liste de mots $[w_1; \ldots; w_n]$, retourne la liste $[ww_1; \ldots; ww_n]$. Rappels de CAML : `[]` représente la liste vide, `::` l'opération d'ajout d'un élément en tête d'une liste, `^` l'opération de concaténation des chaînes de caractères. On définira `prefixe` par pattern-matching sur la liste en argument.

   Correction :

   ```
   let rec prefixe = function
      (w,[])   -> []
   |  (w,x::r) -> (w^x)::(prefixe r)
   ;;
   ```

2. Écrire une fonction `prefixe_c` qui, étant donnée une liste de mots $[w_1; \ldots; w_n]$, retourne la liste $[cw_1; \ldots; cw_n]$. On réutilisera la fonction `prefixe`.

   Correction :

   ```
   let prefixe_c = function(l) => prefixe("c",l);;
   ```

3. Écrire une fonction `prefixe_d` qui, étant donnée deux listes $[v_1; \ldots; v_n]$ et $[w_1; \ldots; w_m]$, retourne la liste de tous les mots dv_iw_j avec $1 \leq i \leq n$ et $1 \leq j \leq m$. On définira cette function par récurrence sur la première liste. Rappel de CAML : `@` est la concaténation des listes. On réutilisera la fonction `prefixe`.

   Correction :

   ```
   let rec prefixe_d = function
     ([],l2) -> []
   | (x::l1,l2) -> prefixe("d"^x,l2) @ prefixe_d(l1,l2);;
   ```

4. En utilisant la formule de récurrence trouvée à la question 3 de l'exercice 1.6, implanter une fonction `engendre` qui, étant donné un entier n, retourne l'ensemble E_n de tous les mots de L_0 de longueur n. On définira une fonction auxiliaire `union_partielle` locale à `engendre`, qui calcule

$$\texttt{union\_partielle}(k) = \bigcup_{i=1}^{k} d.E_i.E_{n-i-1}$$

par récurrence sur k. On réutilisera également les fonctions `prefixe_c` et `prefixe_d`.

Correction :

```
let rec engendre = function
  0 -> []
| 1 -> ["a";"b"]
| n ->
   let union_partielle = function(k) ->
     if k = 0
       then []
       else prefixe_d(engendre k,engendre(n-k-1)) @ union_partielle(k-1)
   in
     prefixe_c(engendre(n-1)) @ union_partielle(n-2);;
```

5. Écrire une fonction `analyse_L0` utilisant `engendre`. Complément de CAML : on utilisera les fonctions CAML `string_length` qui donne la longueur d'une chaîne de caractères, et `mem` telle que (`mem` x l) retourne `true` si l'élément x appartient à la liste l. Attention, pour utiliser cette fonction on écrira bien (`mem` x l) et pas `mem`(x, l). Cette façon d'écrire l'appel à une fonction est commun à toutes les fonctions prédéfinies de CAML, et la différence entre les deux façons d'écrire sera expliquée plus tard en cours.

Correction :

```
let analyse_L0 = function(w) -> (mem w (engendre (string_length w));;
```

Remarque : ne pas oublier les parenthèses autour de l'appel à engendre.

6. Tester `analyse_L0` sur des exemples de plus en plus long. Que peut-on dire sur l'efficacité de cette méthode? Comment augmente (approximativement) le temps de calcul en fonction de la longueur du mot à reconnaître?

Correction : *Impraticable, si la longueur du mot est ≥ 7. La complexité est exponentielle (au moins !).*

Méthodes 2 : analyse par cas en fonction de la première lettre du mot à reconnaître. L'idée générale des deux méthodes que nous allons mettre en œuvre maintenant est la suivante : soit w un mot sur l'alphabet $\{a, b, c, d\}$. Si $w = \varepsilon$ alors $w \notin L_0$, sinon $w = \alpha w'$, et suivant le cas sur α :

- si $\alpha = a$ ou $\alpha = b$, alors $w \in L_0$ si et seulement si $w' = \varepsilon$.
- si $\alpha = b$, alors $w \in L_0$ si et seulement si $w' \in L_0$.
- si $\alpha = d$, alors $w \in L_0$ si et seulement si il existe deux mots w_1 et w_2 tels que $w' = w_1 w_2$ avec $w_1 \in L_0$ et $w_2 \in L_0$.

Méthode 2a ***Correction :*** *En cas de manque de temps, on pourra passer directement à la méthode 2b*

1. Écrire deux fonctions prenant en argument une chaîne de caractères : `premier`(w) retourne la première lettre de w, et `reste`(w) retourne le reste de w. On utilisera la fonction CAML `sub_string` telle que (`sub_string` w n m) retourne la sous-chaîne de w commençant au n-ième caractère et de longueur m. Attention, les caractères sont numérotés à partir de 0.

   Correction :

```
let premier = function(s) -> sub_string s 0 1;;
let reste = function(s) -> sub_string s 1 ((string_length s)-1) ;;
```

2. Implanter la fonction `analyse_L0` par la méthode proposée. On écrira une fonction auxiliaire `cherche_un_decoupage` locale à `analyse_L0`, telle que `cherche_un_decoupage`(w, n) retourne `true` s'il existe deux mots w_1 et w_2 tels que $w = w_1.w_2$, $w_1 \in L_0$, $w_2 \in L_0$ et $|w_1| \leq n$. On raisonnera par récurrence sur n.

   Correction :

```
let rec analyse_L0 = function
  "" -> false
| s ->
    let c = premier(s)
    and r = reste(s)
    in
      match c with
       "a" -> r=""
     | "b" -> r=""
     | "c" -> analyse_L0(r)
     | "d" ->
         let rec cherche_un_decoupage = function(w,n) ->
            if n=0
              then false
              else
                 if analyse_L0(sub_string w 0 n) &
                    analyse_L0(sub_string w n ((string_length w)-n))
                 then true
                 else cherche_un_decoupage(w,n-1)


         in cherche_un_decoupage(r,(string_length r))

     | _ -> false
;;
```

   en PASCAL:

3. Que peut-on dire sur l'efficacité de cette deuxième méthode? Tester sur le mot $w_0 = dddaadaaddaadaa$, puis sur dw_0w_0 et enfin $ddw_0w_0dw_0w_0$.

   Correction : *Beaucoup mieux mais encore exponentiel en fonction de la longueur du mot.*

Méthode 2b On cherche à éviter la recherche d'un découpage de w en w_1w_2 dans le dernier cas, en exploitant la propriété montrée à la question 7 de l'exercice I.

1. Montrer la propriété suivante : pour tout mot w, il existe au plus un préfixe w_1 (c'est-à-dire $w = w_1w_2$) tel que $w_1 \in L_0$.

2. Réaliser une fonction `recherche_prefixe`, qui recherche un tel préfixe, dont la spécification exacte est

   `recherche_prefixe` : chaîne de caractères → chaîne de caractères
   { `recherche_prefixe`(w) retourne l'unique mot w' tel que $w = w_1w'$ où $w_1 \in L_0$,
   déclenche l'erreur `failwith("pas de prefixe")` si aucun préfixe n'est dans L_0 }

   Correction :

```
let rec recherche_prefixe = function
  "" -> failwith("pas de prefixe")
| s ->
    let c = premier(s)
    and r = reste(s)
    in
      match c with
       "a" -> r
     | "b" -> r
     | "c" -> recherche_prefixe(r)
     | "d" -> recherche_prefixe(recherche_prefixe(r))
     | _ -> failwith("pas sur le bon alphabet")
;;
```

3. En déduire une nouvelle réalisation de `analyse_L0`.

   Correction :

```
let analyse_L0(w) = recherche_prefixe(w)="";;
```

   Cette fonction lève une exception quand le mot n'est pas dans le langage, on verra deux questions plus loin comment arranger cela.

4. Que peut-on dire sur l'efficacité de cette dernière méthode? Tester à nouveau sur les mots exemples de la méthode 2a.

   Correction : *Encore mieux. C'est cette fois linéaire.*

Complément de CAML La fonction `analyse_L0` réalisée ici déclenche une erreur au lieu de retourner `false`. On dit qu'une *exception a été levée*. Nous allons maintenant apprendre comment *traiter* ces cas d'exception. Pour cela, CAML attend que le programmeur déclare des noms d'exception pour qualifier les cas d'erreur qui l'intéresse. Ceci s'effectue par l'instruction

`exception` *nom d'exception*`;;`

Ensuite, le programmeur peut lever ces exceptions par la fonction `raise` appliquée au nom de l'exception.

Par exemple, voici une fonction `deuxieme` qui retourne le deuxième élément d'une liste, ou bien lève l'exception `Liste_trop_courte` si la liste est trop courte :

```
exception Liste_trop_courte;;
let deuxieme = function
  x::(y::l) -> y
| _ -> raise(Liste_trop_courte);;
```

5. Refaire la réalisation de la fonction `recherche_prefixe` en levant l'exception `Erreur_lexicale` quand une lettre autre que a, b, c ou d est rencontrée ; et `Erreur_syntaxique` quand aucun préfixe dans L_0 n'a pu être trouvé.

   Correction :

   ```
   exception Erreur_lexicale;;
   exception Erreur_syntaxique;;

   let rec recherche_prefixe = function
     "" -> raise(Erreur_syntaxique)
   | s ->
       let c = premier(s)
       and r = reste(s)
       in
         match c with
          "a" -> r
        | "b" -> r
        | "t" -> recherche_prefixe(r)
        | "c" -> recherche_prefixe(recherche_prefixe(r))
        | _ -> raise(Erreur_lexicale)
   ;;
   ```

Lorsqu'une fonction f peut lever une certaine exception e, il est possible de *capturer* cette exception à l'aide de la construction suivante :

`try` *expression contenant un appel à* f
`with` e `->` *valeur de remplacement*

Cette construction permet de dire que lorsque l'appel à la fonction f ne retourne pas un résultat normal mais l'exception e, alors la valeur de remplacement est à prendre à la place de l'expression contenant l'appel à f. Par exemple, voici une fonction qui utilise `deuxieme`, qui étant donnée une liste d'entiers donne la valeur de son deuxième élément plus 1, ou bien 0 si cette liste est trop courte :

```
#let f = function(l)->
  try deuxieme(l)+1
  with Liste_trop_courte -> 0;;
```

L'utilisation de `f` sous CAML donnera alors :

```
#f([1;2]);;
- : int = 3
#f([1]);;
- : int = 0
```

6. Sur ce principe, refaire la réalisation de `analyse_L0` en capturant l'exception `Erreur_syntaxique`. Ainsi, cette fonction retournera `true` pour un mot de L_0, `false` pour un mot sur le bon alphabet mais pas dans L_0

   Correction :

```
let analyse_L0 = function(w) ->
  try recherche_prefixe(w)=""
  with Erreur_syntaxique -> false
;;
```

Exercice 1.8
Décrire les langages engendrés par les grammaires sur $V_t = \{0, 1\}$, $V_n = \{N, M\}$, de source N et dont les règles sont les suivantes.

1. $N \to 0 \mid 1M \qquad M \to 0 \mid 0M \mid 1M$

   Correction : *C'est l'ensemble des mots non vides se terminant par 0, et commençant par 1 excepté 0 lui-même. Il s'agit donc de l'ensemble des entiers pairs écrits en binaire.*

2. $N \to 1M \qquad M \to \varepsilon \mid 0N \mid 1M$

   Correction : *Cette fois c'est les impairs.*

Exercice 1.9
Donner une grammaire décrivant les langages suivants :

1. les entiers relatifs, le signe étant optionnel pour les entiers naturels. Cette grammaire devra par exemple reconnaître `0`, `33`, `+20` et `-13`.
2. les nombres réels en virgule flottante. Cette grammaire devra par exemple reconnaître `0.0`, `-1.33` et `+20.13`.
3. les identificateurs des langages de programmation classiques, formés par des mots non vides sur l'alphabet des lettres majuscules ou minuscules, des chiffres et du caractère souligné `_` ; et commençant par une lettre. Cette grammaire devra par exemple reconnaître `x`, `Y1`, `ptr_v` et `Date`.
4. les nombres réels en virgule flottante en acceptant les exposants. Cette grammaire devra reconnaître les réels de la question 2 ainsi que, par exemple, `1.0e3`, `-1.33E-3` et `+20.13e+4`.

Exercice 1.10
Donner des grammaires pour les langages suivants :

- Les mots de longueur paire sur l'alphabet $\{a, b\}$.
- Les mots sur l'alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -, :\}$ affichés sur l'écran d'une montre digitale, dans le format année-mois-jour-heures:minutes:secondes.
- Les suites de a et de b ne contenant pas deux a consécutifs.
- Les suites de a et de b ne contenant pas deux a consécutifs ni deux b consécutifs.
- $\{a^n b^n \mid n \in \mathbb{N}\}$, où a^n indique la répétition n fois de la lettre a.
- $\{a^n b^m \mid n, m \in \mathbb{N}, n \leq m\}$

- l'ensemble des *palindromes* sur {a,b}, c'est-à-dire les mots qui se lisent indifféremment à l'envers ou à l'endroit (comme « radar » en français).

Correction :

1. $S \to \varepsilon \mid aaS \mid abS \mid baS \mid bbS$
2.
3. $S \to \varepsilon \mid bS \mid aB\ ,\ B \to \varepsilon \mid bS$
4. $S \to \varepsilon \mid A \mid B\ ,\ A \to \varepsilon \mid aB\ ,\ B \to \varepsilon \mid bA$
5. $S \to \varepsilon \mid aSb$
6. $S \to aSb \mid T\ ,\ T \to \varepsilon \mid bT$
7. $S \to \varepsilon \mid a \mid b \mid aSa \mid bSb$

Exercice 1.11
On note L le langage engendré par la grammaire sur l'alphabet $\{0,1\}$, d'axiome N, définie par les règles

$$N \to 11 \mid 1001 \mid N0 \mid NN$$

1. Trouver tous les mots de L de longueur inférieure ou égale à 5.

   Correction : *11, 110, 1001, 1100, 1111, 10010, 11000, 11110, 11011*

2. Si w est un mot sur $\{0,1\}$, on note $b(w)$ l'entier dont w est l'écriture binaire. Montrer que pour tout mot w de L, $b(w)$ est divisible par 3.

   Correction : *Par récurrence sur la longueur du mot. Si $|w| = 2$: OK. Si $|w| = n > 2$ et si l'on suppose que tout mot de L de longueur inférieure à n a un $b()$ divisible par 3, alors suivant la règle qui engendre w:*

   - *si $N \to 11$: OK.*
   - *si $N \to 1001$: OK.*
   - *si $N \to N0$: alors $w = w_1 0$ avec $w_1 \in L$ donc par hypothèse de récurrence $b(w_1)$ est divisible par 3, or $b(w) = 2.b(w_1)$.*
   - *si $N \to NN$: $w = w_1 w_2$, $b(w) = b(w_1).2^k + b(w_2)$ est divisible par 3.*

3. Est-ce que tous les nombres divisibles par 3 sont représentés dans L?

   Correction : *Non, 21 s'écrit 10101 et n'est pas dans L.*

Exercice 1.12
Quel sont les langages sur l'alphabet $V_t = \{(,)\}$ engendrés par les grammaires qui suivent :

1. $S \to \varepsilon \mid (\ \mid\ ) \mid SS$;
2. $S \to \varepsilon \mid (T \qquad T \to S)$;
3. $S \to \varepsilon \mid (S) \mid SS$;
4. $S \to \varepsilon \mid (S)S$

Correction :

1. VT^* ;
2. $\{(^n)^n \mid n \in \mathbb{N}\}$;
3. *ensemble des mots bien parenthésés sur V_t ;*
4. *idem (cf. exercice 1.14).*

Exercice 1.13
On considère la grammaire $G = (\{a, +, *\}, \{S\}, S, \{S \to SS* | SS+ | a\})$. (C'est une version incomplète de la grammaire qui régit le fonctionnement d'une calculatrice HP.)

1. Trouvez les mots de L_G de longueur inférieure ou égale à 5 et donner une des dérivations qui vous permettent de les engendrer.

   Correction :

   - $\underline{S} \to a$,
   - $\underline{S} \to \underline{S}S+ \to a\underline{S}+ \to aa+$,
   - $\underline{S} \to \underline{S}S* \to a\underline{S}* \to aa*$,
   - $\underline{S} \to \underline{S}S+ \to \underline{S}S + S+ \to a\underline{S} + S+ \to aa + \underline{S}+ \to aa + a+$,
   - $\underline{S} \to \underline{S}S* \to \underline{S}S + S* \to a\underline{S} + S* \to aa + \underline{S}* \to aa + a*$,
   - $\underline{S} \to \underline{S}S* \to \underline{S}S * S* \to a\underline{S} * S* \to aa * \underline{S}* \to aa * a*$,
   - $\underline{S} \to \underline{S}S+ \to \underline{S}S * S+ \to a\underline{S} * S+ \to aa * \underline{S}+ \to aa * a+$,
   - $\underline{S} \to \underline{S}S+ \to a\underline{S}+ \to a\underline{S}SS + + \to aa\underline{S} + + \to aaa + +$,
   - $\underline{S} \to \underline{S}S* \to a\underline{S}* \to a\underline{S}S + * \to aa\underline{S} + * \to aaa + *$,
   - $\underline{S} \to \underline{S}S* \to a\underline{S}* \to a\underline{S}S * * \to aa\underline{S} * * \to aaa * *$,
   - $\underline{S} \to \underline{S}S+ \to a\underline{S}+ \to a\underline{S}SS * + \to aa\underline{S} * + \to aaa * +$

2. Dans la suite on note $|w|_\alpha$ le nombre d'occurences de la lettre α dans le mot w.

   Montrer par récurrence sur la longueur de la dérivation du mot que si $w \in L_G$ alors $|w|_+ + |w|_* + 1 = |w|_a$.

   Correction : *Pour le mots de longueur que l'on peut dériver en une étape, (ie a) c'est vrai.* $(0 + 0 + 1 = 1)$

   Les deux cas d'induction sont similaires. Faisons le cas $S \to SS+$.

   Supposons que cela est pour tous les mots que l'on peut dériver en moins de n étapes. Soit w_1 *et* w_2 *deux mots quelconques engendrés respectivement en* $n-k$ *et* k $(n, k \geq 1)$ *étapes. On peut en n étapes engendrer les mots de la forme* $w_1 w_2+$. *Par hypothèse d'induction, nous avons* $|w_1|_+ + |w_1|_* + 1 = |w_1|_a$ *et* $|w_2|_+ + |w_2|_* + 1 = |w_2|_a$. *Nous avons* $|w_1 w_2 +|_+ = |w_1|_+ + |w_2|_+ + 1$, $|w_1 w_2 +|_* = |w_1|_* + |w_2|_*$ *et* $|w_1 w_2 +|_a = |w_1|_a + |w_1|_a$. *Donc* $|w_1 w_2 +|_+ + |w_1 w_2 +|_* + 1 = (|w_1|_+ + |w_2|_+ + 1) + (|w_1|_* + |w_2|_*) + 1 = (|w_1|_+ + |w_1|_* + 1) + (|w_2|_+ |w_2|_* + 1) = |w_1|_a + |w_2|_a = |w_1 w_2 +|_a$.

3. Trouver une procédure qui permet de décider si un mot appartient au langage. On pourra utiliser une procédure auxiliaire qui maintient à jour une variable représentant le nombre d'expressions disponibles pour de futures opérations.

   Correction : *L'idée est de garder un trace à l'aide de la première variable du nombre d'arguments disponibles. En fin de mot, ce nombre d'argument doit être égal à un. Une opération "consomme" deux arguments et en rend un, le résultat de l'opération. De plus à aucun moment le nombre d'arguments disponibles ne doit être négatif. En CAML cela donne:*

```
let decision =
  let rec est_dans_lg nb_arg = function
     "a"::tl -> est_dans_lg (nb_arg+1) tl
    |_  ::tl -> if nb_arg >=2
                then est_dans_lg (nb_arg - 2 + 1) tl
                else false
    |_       -> nb_arg = 1
  in est_dans_lg 0;;
```

Exercice 1.14
Soit L_1 le langage engendré par la grammaire

$$S \to (S) \mid SS \mid \varepsilon$$

et L_2 le langage engendré par la grammaire

$$T \to (T)T \mid \varepsilon$$

1. Montrer que $L_2 \subseteq L_1$.

   Correction : *Si $w \in L_2$, alors soit $w = \varepsilon$ auquel cas $w \in L_1$; soit $w = (w_1)w_2$ avec $w_1, w_2 \in L_2$ par récurrence, on a $w_1, w_2 \in L_1$, mais alors w se dérive par la première grammaire: $S \to SS \to (S)S \to^* (w_1)w_2 = w$.*

2. Montrer que L_2 est stable par concaténation, c'est-à-dire si $w_1 \in L_2$ et $w_2 \in L_2$ alors $w_1 w_2 \in L_2$.

   Correction : *Si $w_1 = \varepsilon$, c'est clair, sinon $w_1 = (w_3)w_4$. Par récurrence on a $w_4 w_2 \in L_2$ mais alors $T \to (T)T \to^* (w_3)w_4 w_2$.*

3. En déduire que $L_1 \subseteq L_2$.

   Correction : *Soit $w \in L_1$. Si $w = \varepsilon$: OK. Si $w = (w_1)$, $w_1 \in L_2$ par hypothèse de récurrence, $T \to (T)T \to (T) \to^* (w_1)$. Si $w = w_1 w_2$: on conclut par l'hypothèse de récurrence et la propriété précédente.*

Exercice 1.15
On considère la grammaire $G = (\{a, b\}, \{S, A, B\}, S, R)$ où R contient les règles

$$\begin{array}{lcl} S & \to & \varepsilon \mid aB \mid bA \\ A & \to & aS \mid bAA \\ B & \to & bS \mid aBB \end{array}$$

On note L_G le langage engendré par G.

1. Montrer que les mots $abab$ et $baaabb$ sont dans L_G.

   Correction : *$S \to aB \to abS \to abaB \to ababS \to abab$*

   $S \to bA \to baS \to baaB \to baaaBB \to baaabSB \to baaabB \to baaabbS \to baaabb$

2. Montrer que chaque $w \in L_G$ vérifie $|w|_a = |w|_b$. Indication : soit L_A l'ensemble de mots qui dérivent de A, et soit L_B l'ensemble de mots qui dérivent de B. Montrer que pour chaque mot w sur l'alphabet $\{a, b\}$, par récurrence sur la longueur de w, les trois assertions

   (a) si $w \in L_G$ alors $|w|_a = |w|_b$,
   (b) si $w \in L_A$ alors $|w|_a = |w|_b + 1$,
   (c) si $w \in L_B$ alors $|w|_a + 1 = |w|_b$.

   Correction : *Il faut faire une preuve par récurrence simultanée.*

   $w = \varepsilon$: $w \in L_G$, $w \notin L_A$, $w \notin L_B$.

   $w = aw'$ et $w \in L_G$: on a $w' \in L_B$, alors par récurrence

   $$|w|_a = |aw'|_a = 1 + |w'|_a = |w'|_b = |aw'|_b = |w|_b$$

$w = aw'$ et $w \in L_A$: on a $w' \in L_G$, alors par récurrence

$$|w|_a = |aw'|_a = 1 + |w'|_a = 1 + |w'|_b = 1 + |aw'| = 1 + |w|_b$$

$w = aw'$ et $w \in L_B$: on a $w' = vv'$ avec $v, v' \in L_B$, alors par récurrence

$$|w|_a + 1 = |avv'|_a + 1 = 1 + |v|_a + |v'|_a + 1 = |v|_b + |v'|_b = |avv'|_b = |w|_b$$

Les cas où w commence par b sont symétriques.

3. Montrer que pour chaque mot w sur l'alphabet $\{a, b\}$: si $|w|_a = |w|_b$ alors $w \in L_G$. Indication: Montrer que pour chaque mot w sur l'alphabet $\{a, b\}$, par récurrence sur la longueur de w, les trois assertions
   (a) si $|w|_a = |w|_b$ alors $w \in L_G$,
   (b) si $|w|_a = |w|_b + 1$ alors $w \in L_A$,
   (c) si $|w|_a + 1 = |w|_b$ alors $w \in L_B$.

   Correction : *similaire*

Exercice 1.16
Considérons le langage L des mots *non carrés* sur l'alphabet $\{a, b\}$, c'est-à-dire l'ensemble des mots w tels qu'il n'y ait pas de mot v tel que $w = vv$.

1. Donner une grammaire pour le langage des mots de longueur impaire de L.

   Correction : *On remarque que tous les mots de longueur impaire sont dans L.*

$$A \rightarrow a \mid b \mid aaA \mid abA \mid baA \mid bbA$$

2. Montrer que si $w = v_1av_2bv_3$ avec $|v_2| = |v_1| + |v_3|$ alors $w \in L$.

   Correction : *Il faut montrer que w n'est pas un carré. Par l'absurde, si $w = vv$ alors $vv = v_1av_2bv_3$ mais comme $|v_2| = |v_1| + |v_3|$, on peut découper v_2 en v_4v_5 avec $|v_4| = |v_3|$ et $|v_5| = |v_1|$, et alors $|v_1av_4| = |v_5bv_3|$ donc $v = v_1av_4 = v_5bv_3$ mais cette dernière égalité est impossible car a et b sont à la même position.*

3. En exprimant l'ensemble des mots de la forme ci-dessus (en découpant v_2 de manière astucieuse), donner une grammaire engendrant les mots de longueur paire de L.

   Correction : *On remarque que l'on peut également découper v_2 en v_4v_5 avec cette fois $|v_4| = |v_1|$ et $|v_5| = |v_3|$. Autrement l'ensemble des mots de la forme ci-dessus est*

$$\{v_1av_4v_5bv_3 \mid |v_4| = |v_1| \text{ et } |v_5| = |v_3|\}$$

   Ce langage est engendré par la grammaire

$$\begin{aligned} B &\rightarrow B_1B_2 \\ B_1 &\rightarrow a \mid aB_1a \mid aB_1b \mid bB_1a \mid bB_1b \\ B_2 &\rightarrow b \mid aB_2a \mid aB_2b \mid bB_2a \mid bB_2b \end{aligned}$$

   On sait que tous les mots de ce langage sont dans L, réciproquement on remarque que les mots de L de longueur paire sont soit de la forme ci-dessus, soit de la forme obtenue en échangeant a et b. Par conséquent, une grammaire engendrant les mots de longueur paire de L est

$$B \rightarrow B_1B_2 \mid B_2B_1$$

4. En déduire une grammaire pour L.

   Correction :

$$S \rightarrow A \mid B_1B_2 \mid B_2B_1$$

Chapitre 2

Grammaires régulières et automates

Ce chapitre a pour but de donner une méthode permettant de reconnaître les mots d'un langage. On s'intéresse à une classe de langages simples à reconnaître, les langages réguliers, reconnaissables par des automates.

2.1 Grammaires régulières

Définition 2.1 *Une grammaire $G = (V_t, V_n, S, R)$ est dite* régulière *si toutes ses règles sont de la forme $N \to w$, avec $w \in V_t^*$ ou bien $w = uM$, $u \in V_t^*$ et $M \in V_n$. Un langage engendré par une grammaire régulière est dit* régulier.

Les grammaires régulières sont beaucoup moins expressives que les grammaires hors-contexte. En particulier, elles ne peuvent pas engendrer le langage $\{a^n b^n \mid n \in \mathbb{N}\}$ (cf. exercice 2.3). Elles peuvent en fait engendrer des langages finis, ou des langages infinis dont la structure est simple.

Nous allons maintenant simplifier la forme de nos règles régulières en plusieurs étapes, sans en changer le pouvoir d'expression, c'est-à-dire que nous pourrons décrire les mêmes langages, avec bien sûr des règles différentes. Dans un premier temps, nous imposons que le membre droit contienne au plus un symbole terminal. Il reste quatre sortes de règles possibles :

$$N \to \varepsilon \qquad N \to \alpha \qquad N \to M \qquad N \to \alpha M$$

où $N \in V_n$, $\alpha \in V_t$ et $M \in V_n$. On peut en effet toujours transformer une règle de la forme $N \to \alpha_1 \cdots \alpha_n$ (resp. $N \to \alpha_1 \cdots \alpha_n M$) en n règles $N \to \alpha_1 N_1$, $N_1 \to \alpha_2 N_2$, $\ldots N_{n-1} \to \alpha_n$ (resp. $N_{n-1} \to \alpha_n M$) sans changer le langage engendré, au prix de l'ajout de $n-1$ nouveaux non-terminaux à la grammaire.

Par exemple, partant de la grammaire

$$\begin{array}{lcl} S & \to & ab \mid T \\ T & \to & cdT \mid U \\ U & \to & e \mid S \end{array}$$

on construit la grammaire

$$S \to aS_1 \mid T$$

$$\begin{aligned} T &\to cT_1 \mid U \\ U &\to e \mid S \\ S_1 &\to b \\ T_1 &\to dT \end{aligned}$$

On va maintenant éliminer les règles de la forme $N \to M$, en prenant soin d'ajouter les règles $N \to w$ pour toute règle $M \to w$ telle que $w \notin V_n$. La transformation à effectuer est en fait un peu plus complexe, il faut ajouter les règles en question pour tout non-terminal $M \neq N$ tel que $N \to^* M$, c'est-à-dire tel qu'il y ait une dérivation de N vers M.

Ainsi si on reprend la grammaire obtenue précédemment, on obtient

$$\begin{aligned} S &\to aS_1 \mid cT_1 \mid e \\ T &\to cT_1 \mid aS_1 \mid e \\ U &\to e \mid aS_1 \mid cT_1 \\ S_1 &\to b \\ T_1 &\to dT \end{aligned}$$

La grammaire obtenue à l'issue de ces deux phases ne nous satisfait pas encore entièrement car elle peut contenir des règles dont le membre droit est un seul terminal α. Les règles $N \to \alpha$ peuvent en fait être facilement éliminées à condition d'ajouter une règle $E \to \varepsilon$ (où E est un nouveau non-terminal) et de remplacer toute règle $N \to \alpha$ par $N \to \alpha E$.

Sur l'exemple précédent, cela donne

$$\begin{aligned} S &\to aS_1 \mid cT_1 \mid eE \\ T &\to cT_1 \mid aS_1 \mid eE \\ U &\to eE \mid aS_1 \mid cT_1 \\ S_1 &\to bE \\ T_1 &\to dT \\ E &\to \varepsilon \end{aligned}$$

Définition 2.2 *Une grammaire régulière $G = (V_t, V_n, S, R)$ est dite* réduite *si toutes ses règles sont de la forme $N \to \alpha M$ ou $N \to \varepsilon$, où $N \in V_n$, $\alpha \in V_t$ et $M \in V_n$.*

La discussion précédente montre que pour toute grammaire régulière, il existe une grammaire régulière réduite qui engendre le même langage.

Considérons la deuxième grammaire de la figure 1.1 engendrant les entiers en numération binaire, qui est régulière et dont la variante réduite est :

$$\begin{aligned} N &\to 0E \mid 1M \\ M &\to 0M \mid 1M \mid \varepsilon \\ E &\to \varepsilon \end{aligned}$$

Les dérivations de cette grammaire sont extrêmement simples, puisqu'il n'y a pas le choix du non-terminal à remplacer. Par exemple

$$N \to 1M \to 10M \to 100M \to 1001M \to 10011M \to 10011$$

De plus, il est facile de voir que cette grammaire est non-ambiguë, car la règle à utiliser est entièrement déterminée à chaque étape par le prochain symbole du mot à engendrer : il y a donc une unique dérivation pour chaque mot, et donc un unique arbre syntaxique (qui est filiforme). Cela n'est pas le cas de toutes les grammaires régulières réduites. Par exemple, la

première grammaire de la figure 1.1 engendrant les entiers en numération binaire, dont la variante réduite est :

$$\begin{aligned} N &\rightarrow 0E \mid 1E \mid 1M \\ M &\rightarrow \varepsilon \mid 0M \mid 1M \\ E &\rightarrow \varepsilon \end{aligned}$$

est ambiguë puisque le mot 1 a deux arbres syntaxiques différents. Nous verrons dans les paragraphes suivants un cas particulier de grammaire régulière réduite pour lequel la non-ambiguïté est garantie.

2.2 Automates déterministes

Nous allons maintenant nous préoccuper de reconnaître les mots engendrés par une grammaire régulière réduite. Le mécanisme est celui des automates.

Définition 2.3 *Un* automate déterministe A *est un quintuplet* (V_t, Q, q_0, F, T) *où*

1. V_t *est le* vocabulaire *de* A *;*
2. Q *est l'ensemble des* états *de* A *;*
3. $q_0 \in Q$ *est* l'état initial *de* A *;*
4. $F \subseteq Q$ *est l'ensemble des* états acceptants *de* A *;*
5. T *est la* fonction de transition, *une fonction de* $Q \times V_t$ *dans* Q.

Lorsque $T(q, \alpha) = q'$, on notera $q \xrightarrow{\alpha} q'$ ce qui se lit « dans l'état q, en lisant la lettre α on passe dans l'état q' ». Lorsque T est une application, autrement dit s'il y a dans chaque état exactement une transition pour toute lettre de l'alphabet, l'automate est dit *complet*.

On a l'habitude de dessiner les automates, en figurant les états par des cercles, en indiquant l'état initial par une flèche entrante, les états acceptants par un double cercle ou une flèche sortant d'un seul cercle, et la transition de l'état q à l'état q' en lisant la lettre α par une flèche allant de q vers q' et étiquetée par α. Les automates sont donc des graphes, dont les nœuds et les états sont étiquetés.

Reconnaître un mot par un automate fonctionne comme suit : les lettres du mot $\alpha_1 \dots \alpha_n$ sont entrées dans l'automate à partir de la gauche, c'est-à-dire dans l'ordre $\alpha_1, \dots, \alpha_n$. La lettre α_1 provoque une transition de l'état initial q_0 vers un état q_1 (qui peut être le même état que l'état initial), puis la lettre α_2 provoque une transition de l'état q_1 vers un état q_2, etc. jusqu'à arriver à un état q_n. Il suffit alors de regarder si $q_n \in F$ auquel cas le mot est reconnu, alors qu'il est rejeté dans le cas contraire. Notons qu'il n'est pas possible d'être stoppé en chemin lorsque l'automate est complet. Si au contraire l'automate est incomplet, on convient que le mot n'est pas reconnu lorsque l'exécution est stoppée en chemin.

Définition 2.4 *Étant donné un automate* $A = (V_t, Q, q_0, F, T)$, *on appelle* exécution *toute suite (éventuellement vide) de transitions* $q_0 \xrightarrow{\alpha_1} q_1 \cdots q_{n-1} \xrightarrow{\alpha_n} q_n$, *aussi notée* $q_0 \xrightarrow{w}{}^* q_n$, *qui* lit *le mot* $w = \alpha_1 \cdots \alpha_n$. *On appelle alors* q_n l'état atteint *lors de l'exécution de* A *sur la donnée* w.

Un mot w *est* reconnu *par l'automate* A *s'il existe une exécution de l'automate* A *qui se termine en ayant lu le mot* w, *et tel que l'état atteint soit acceptant. On note par* $\mathcal{L}\text{ang}(A)$ *le langage des mots reconnus par l'automate* A. *Un langage reconnu par un automate est dit* reconnaissable.

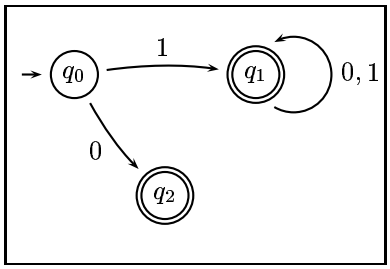


FIG. 2.1 – *Automate déterministe reconnaissant les entiers naturels en numération binaire.*

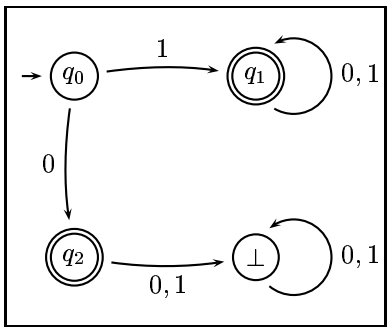


FIG. 2.2 – *Automate déterministe complet reconnaissant les entiers naturels en numération binaire.*

La figure 2.1 représente un automate reconnaissant le langage des entiers naturels en représentation binaire. Cet automate n'est pas complet (par exemple $T(q_2, 0)$ n'est pas défini), aussi il est susceptible de se bloquer lors d'une exécution, ce qui arrive si l'on veut exécuter cet automate sur le mot 00.

En pratique, il peut être utile de rendre un automate complet en ajoutant un nouvel état appelé *poubelle*, étiqueté par $\perp$, vers lequel vont toutes les transitions manquantes. L'automate obtenu reconnaît exactement le même langage si l'on convient que le nouvel état poubelle n'est pas acceptant. Ainsi, la figure 2.2 présente un automate complet reconnaissant le langage des entiers naturels en représentation binaire.

Notons que la reconnaissance d'un mot w par un automate est automatique : la lecture des lettres composant le mot provoque des transitions bien définies jusqu'à être bloqué (en cas de transitions manquantes) ou bien jusqu'à atteindre un état qui peut être ou ne pas être un état de satisfaction. Un automate déterministe est donc exempt d'ambiguïté.

2.3 Programmation des automates

2.3.1 Spécifications

La spécification du type abstrait « automate sur (V_t, Q) » est la suivante. Elle suppose donnée un alphabet V_t et un ensemble d'états Q.

etat_initial : automate sur $(V_t, Q) \to Q$

{ *etat_initial*(A) est l'état initial de A }

est_acceptant : automate sur $(V_t, Q) \times Q \to$ booléen

{ *est_acceptant*(A, q) est vrai si q est acceptant dans A }

transition : automate sur $(V_t, Q) \times Q \times V_t \to Q$

{ *transition*(A, q, α) retourne l'état $T(q, \alpha)$ donné par A }

Cette spécification exprime donc le fait qu'un automate sur un alphabet donné V_t et un ensemble d'états Q donné est caractérisé par les trois fonctions précédentes. Une implantation concrète de ce type de données dans un certain langage de programmation pourra être quelconque à condition de fournir au minimum ces trois fonctionnalités.

Les fonctions importantes sur les automates sont alors les suivantes.

fonction *exécute* : automate sur $(V_t, Q) \times Q \times V_t^* \to Q$

{ *exécute*(A, q, w) retourne l'état final atteint par lecture du mot w sur l'automate A à partir de l'état q }

fonction *reconnaît* : automate sur $(V_t, Q) \times V_t^* \to$ booléen

{ *reconnaît*(A, w) retourne vrai si $w \in \mathcal{L}ang(A)$ }

2.3.2 Implantations

Il y a plusieurs façons d'implémenter ces spécifications dans un langage évolué, d'abord suivant le choix de ce langage, et ensuite suivant que l'on veuille implanter un automate particulier, ou bien implanter des fonctions génériques fonctionnant sur n'importe quel automate représenté par une structure de données.

2.3.3 Fonctions *exécute* et *reconnaît*

Mais pour commencer, regardons comment on peut implanter les fonctions *exécute* et *reconnaît*. Cette dernière se définit très simplement à partir des autres fonctions par

$$reconnaît(A, w) = est\_acceptant(A, exécute(A, etat\_initial(A), w))$$

se qui se traduit très facilement dans n'importe quel langage évolué.

La fonction *exécute* se traduit soit en une boucle (langage impératif), soit une fonction récursive (langage fonctionnel). Le principe est de toute façon le même : on parcourt les transitions jusqu'à ce que le mot d'entrée soit entièrement lu. Voici une version impérative de cette fonction :

```
fonction exécute(A:automate,q:etat,m:mot):etat
  tant que m n'est pas vide faire
    q = transition(A,q,tete(m))
    m = reste(m)
  fin tant que
  retourner q
fin fonction
```

et en voici une version fonctionnelle :

```
fonction exécute(A:automate,q:etat,m:mot):etat
  si m est vide
    q
  sinon
    soit q' = transition(A,q,tete(m))
    dans exécute(A,q',reste(m))
  fin si
fin fonction
```

2.3.4 Implantation du type automate

Il nous reste maintenant à montrer comment implanter les fonctionnalités des automates, c'est-à-dire principalement la fonction *transition*. Comme dit précédemment, cela dépend si l'on désire simplement coder un automate particulier, ou si l'on veut coder les automates en général.

Commençons par le premier cas qui est plus simple : il nous suffit d'écrire une fonction définie à l'aide de cas. Sur l'exemple de l'automate de la figure 2.2, cela donne la fonction suivante :

type etat = $q_0, q_1, q_2, \perp$

fonction *etat_initial*():etat
  retourner q_0
fin fonction

fonction *est_acceptant*(q:etat):etat
  si q=q_1 ou q=q_2 **alors** retourner vrai **sinon** retourner faux
fin fonction

fonction *transition*(q:etat, a:lettre):etat
  cas q=q_0 :
    cas a=0 : retourner q_2
    cas a=1 : retourner q_1
  cas q=q_1 : retourner q_1
  cas q=q_2 : retourner $\perp$
  cas q=$\perp$: retourner $\perp$
fin fonction

Dans le cas où l'on désire coder les automates en général, il s'agit de coder, une fois pour toutes, les fonctions *transition* et *est_acceptant* en fonction de la représentation choisie de l'automate. Comme d'habitude, ce choix de représentation est guidé par deux objectifs : l'efficacité des fonctions d'accès, ici *transition* et *est_acceptant*, et l'économie d'utilisation de la mémoire. Comme souvent, ces deux objectifs sont contradictoires, et la bonne représentation nécessite un compromis qui dépend de l'application.

Nous allons voir successivement deux représentations de la fonction de transition des automates. La première, la plus simple, consiste à représenter la fonction de transition sous la forme d'un tableau à deux dimensions. Sur notre exemple des entiers en binaire, cela donne :

	0	1
q_0	q_2	q_1
q_1	q_1	q_1
q_2	$\perp$	$\perp$
$\perp$	$\perp$	$\perp$

Pour faciliter la lecture du tableau, la première ligne indique les noms des lettres successives de l'alphabet de l'automate, et la première colonne les noms des états. La représentation mémoire omet bien sûr cette première ligne et cette première colonne, sachant que, *par convention*, les états sont numérotés de 1 à n, et les lettres de 1 à p. La transition de l'état de nom q_i en lisant la lettre de nom α_j est donc indiquée à la case (i, j) du tableau. Si le tableau a pour nom T, l'état obtenu est donc donné par $T(i, j)$. La fonction *transition* générique s'écrit donc

fonction *transition*(A:automate,q:etat,a:lettre):etat
  retourner T(q,a)
fin fonction

On peut améliorer légèrement cette représentation en représentant un automate incomplet, ce qui se comprend car la dernière ligne est toujours remplie de $\perp$. Ceci donne sur notre exemple des entiers en numération binaire :

	0	1
q_0	q_2	q_1
q_1	q_1	q_1
q_2	$\perp$	$\perp$

Cela est facile à mettre en œuvre dans le calcul de la fonction *transition*:

```
fonction transition(A:automate,q:etat,a:lettre):etat
 si q=⊥
  retourner ⊥
 sinon
  retourner T(q,a)
 fin si
fin fonction
```

On peut aussi remarquer qu'une fois l'état poubelle atteint, le mot à lire ne peut pas être reconnu, et on peut donc accélérer la non-reconnaissance en retournant la valeur faux immédiatement. Cela nécessite bien sûr une modification de la fonction *exécute*.

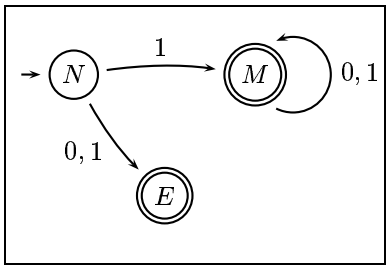
Cette solution reste gourmande en mémoire, puisqu'elle occupe $n \times p$ cases mémoires. Cela est rédhibitoire pour les applications, par exemple à la preuve de circuits intégrés, qui nécessitent la manipulation d'automates pouvant avoir des millions d'états. On est alors amené à utiliser des techniques de représentation spécifiques (de type matrices creuses), en éliminant comme précédemment les transitions vers la poubelle, et en tirant parti de la structure de l'automate de manière à partager des parties communes. C'est ce qui est fait dans l'outil LEX que nous verrons au chapitre 3, les automates d'analyse lexicale pouvant être assez volumineux eux aussi (des centaines d'états et de lettres, donc des dizaines de milliers de cases dans le tableau).

La représentation permettant d'utiliser beaucoup moins de mémoire est les *listes de succession*. Cela consiste à associer à chaque état q la liste des couples (a, q') tels que $T(q, a) = q'$ pour lesquels q' n'est pas l'état poubelle. Cette représentation est proche du dessin de l'automate, et bien sûr beaucoup plus économe que la précédente lorsqu'il y a peu de transitions vers des états qui ne soient pas l'état poubelle. Sur l'exemple, cela donne :

q_0	$[(0, q_2); (1, q_1)]$
q_1	$[(0, q_1); (1, q_1)]$
q_2	[ ]

Cette nouvelle représentation ne nécessite pas de numéroter les états, ni les lettres de l'alphabet, l'un et l'autre apparaissant dans la représentation. Elle est beaucoup plus économe en place puisqu'elle occupe $n \times 2t$ cases mémoires où t est le nombre moyen de transitions par états, en général petit devant p. Programmer la fonction *transition* est un peu plus complexe puisqu'il s'agit de rechercher la transition voulue dans la liste concernée :

```
fonction transition(A:automate,q:etat,a:lettre):etat
 l = liste_succession(A,q)
 retourner chercher(l,a)
fin fonction
```



$$\begin{array}{lcl} N & \to & 0E \mid 1E \mid 1M \\ M & \to & \varepsilon \mid 0M \mid 1M \\ E & \to & \varepsilon \end{array}$$

FIG. 2.3 – *Automate et grammaires correspondantes pour les entiers naturels en notation binaire.*

où la fonction *chercher* se code, en impératif, par

```
fonction chercher(l:liste,a:lettre):etat
  tant que l n'est pas vide
    (a',q')=tete(l)
    si a=a' retourner q'
    l = reste(l)
  fin tant que
  retourner ⊥
fin fonction
```

et en récursif par

```
fonction chercher(l:liste,a:lettre):etat
  si l est vide
    retourner ⊥
  sinon
    soit (a',q')=tete(l)
    si a=a'
      retourner q'
    sinon
      retourner chercher(reste(l),a)
    fin si
  fin si
fin fonction
```

2.4 Automates non-déterministes et grammaires régulières

L'ensemble des états d'un automate joue un rôle analogue aux non-terminaux d'une grammaire. Mais le vocabulaire des états n'a nul besoin d'être disjoint du vocabulaire terminal car ils ne sont jamais mélangés lors des exécutions. Pour comprendre la relation entre les deux notions, il suffit d'examiner un exemple de mot, par exemple le mot 10110, engendré par la grammaire et reconnu par l'automate de la figure 2.3.

On voit que la dérivation dans la première grammaire de la figure 2.3, et l'exécution de l'automate sont presque identiques :

$$\begin{array}{lccccccccccc} \text{dérivation :} & N & \to & 1M & \to & 10M & \to & 101M & \to & 1011M & \to & 10110E \to 10110 \\ \text{exécution :} & N & \xrightarrow{1} & M & \xrightarrow{0} & M & \xrightarrow{1} & M & \xrightarrow{1} & M & \xrightarrow{0} & E \\ \text{mot reconnu :} & & 1 & & 10 & & 101 & & 1011 & & 10110 & \end{array}$$

De manière générale,

- à tout automate $A = (V_t, Q, q_0, F, T)$, on associe une grammaire régulière réduite $G = (V_t, Q, q_0, R)$ dont les non-terminaux sont les états de l'automate, l'état initial étant identifié avec la source, et dont les règles sont obtenues en ajoutant aux transitions de l'automate une règle produisant le mot vide pour chaque état de satisfaction :

$$R = \{q \to aq' \mid T(q,a) = q'\} \cup \{q \to \varepsilon \mid q \in F\}$$

- à toute grammaire régulière réduite $G = (V_t, V_n, S, R)$, on associe un automate $A = (V_t, V_n, S, F, T)$, où $F = \{M \mid M \to \varepsilon \in R\}$, et dont les transitions sont obtenues comme suit :

$$M \xrightarrow{a} N \text{ ssi } M \to aN \in R$$

La première transformation produit une grammaire régulière réduite qui est nécessairement non-ambiguë, puisqu'il n'y aura qu'une seule dérivation possible. Au contraire, si la grammaire de départ est ambiguë, la seconde transformation n'engendre pas nécessairement un automate déterministe. Partant de la grammaire

$$(\{0,1\}, \{S,T\}, S, \{S \to 1S \mid 1T \mid \varepsilon, T \to 1T \mid \varepsilon\})$$

qui engendre le langage $\{1\}^*$, on obtient l'automate

$$(\{0,1\}, \{S,T\}, S, \{S,T\}, \{S \xrightarrow{1} S, S \xrightarrow{1} T, T \xrightarrow{1} T)$$

qui n'est pas un automate déterministe. En effet, sa fonction de transition n'est pas une application puisqu'elle associe à la donnée S le résultat S ou le résultat T. Cela montre que notre notation $q \xrightarrow{a} q'$ est plus générale que nécessaire, puisqu'elle permet de décrire des automates *non-déterministes* : ce sont des automates qui permettent plusieurs transitions depuis un état donné étiquetés par la même lettre, ce que l'on va formaliser en disant que la fonction de transition associe à chaque état et à chaque lettre un ensemble d'état.

Définition 2.5 *Un* automate non-déterministe *A est un quintuplet (V_t, Q, q_0, F, T) où*

1. *V_t est le* vocabulaire *de A,*
2. *Q est l'ensemble des* états *de A,*
3. *$q_0 \in Q$ est l'*état initial *de A,*
4. *$F \subseteq Q$ est l'ensemble des* états acceptants *de A,*
5. *T, la* fonction de transition *de A, est une application de $Q \times V_t \to \mathcal{P}(Q)$.*

La fonction de transition est maintenant à valeurs dans l'ensemble des parties de l'ensemble des états, $T(q, \alpha)$ est donc l'ensemble des états que l'on peut atteindre à partir de l'état q en lisant la lettre α. On continuera à noter $q \xrightarrow{\alpha} q'$ lorsque $q' \in T(q, \alpha)$.

Un automate déterministe est alors un cas particulier d'automate non-déterministe qui associe à tout élément de $Q \times V_t$ une partie de Q possédant zéro ou un élément (exactement un si c'est un automate complet). La reconnaissance d'un mot par un automate non-déterministe implique la *recherche* d'une exécution qui réussisse parmi toutes les exécutions possibles :

Définition 2.6 *Étant donné un automate non-déterministe $A = (V_t, Q, q_0, F, T)$, on appelle* exécution *associée au mot $\alpha_1 \ldots \alpha_n \in V_t^*$ toute suite (éventuellement vide) de transitions $q_0 \xrightarrow{\alpha_1} q_1 \ldots q_{n-1} \xrightarrow{\alpha_n} q_n$, qui* lit *le mot $\alpha_1 \times \cdots \times \alpha_n$. Un mot w est reconnu par l'automate A s'il existe une exécution de l'automate A qui se termine en état acceptant en ayant lu le mot w. On note par $\mathcal{L}ang(A)$ le langage des mots reconnus par l'automate A.*

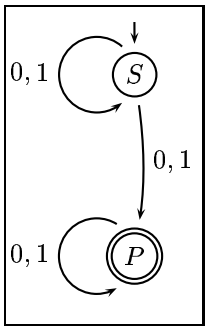


FIG. 2.4 – *Un automate non-déterministe.*

Par exemple, considérons l'automate de la figure 2.4 qui reconnaît le langage $\{0,1\}^+$, et le mot 10110 de ce langage. Des deux exécutions qui suivent, seule la seconde est un succès

$$S \xrightarrow{1} S \xrightarrow{0} S \xrightarrow{1} S \xrightarrow{1} S \xrightarrow{0} S, \qquad S \xrightarrow{1} S \xrightarrow{0} S \xrightarrow{1} S \xrightarrow{1} S \xrightarrow{0} P$$

mais cela suffit pour que le mot 10110 soit reconnu.

2.5 Déterminisation des automates

On pourrait croire que la classe des automates non-déterministes est strictement plus expressive que la classe des automates déterministes. En fait, il n'en est rien : il est toujours possible de *déterminiser* un automate en ajoutant de nouveaux états : si Q est l'ensemble des états d'un automate non-déterministe, l'ensemble des états de l'automate déterministe associé sera $\mathcal{P}(Q)$, l'ensemble des parties de Q. L'idée est que s'il est possible dans l'automate non-déterministe d'atteindre les états $q_1, \ldots, q_n$ depuis l'état q en lisant la lettre a, alors il sera possible dans l'automate déterministe d'atteindre l'état $\{q_1, \ldots, q_n\}$ en partant de tout état contenant q et en lisant la lettre a.

Définition 2.7 *Soit $A = (V_t, Q, q_0, F, T)$ un automate non-déterministe. On définit $\mathcal{D}et(A)$ comme l'automate $(V_t, \mathcal{P}(Q), \{q_0\}, F_{det}, T_{det})$ où*

$$\begin{aligned} F_{det} &= \{K \in \mathcal{P}(Q) \mid K \cap F \neq \emptyset\} \\ T_{det}(K,a) &= \bigcup_{q \in K} T(q,a) \end{aligned}$$

Théorème 2.8 *Soit A un automate non-déterministe. Alors $\mathcal{D}et(A)$ est déterministe et reconnaît le même langage que A.*

Par exemple, l'automate de la figure 2.5 est le déterminisé de celui de la figure 2.4. Notons que l'automate déterminisé obtenu à partir d'un automate non-déterministe possède en général un grand nombre d'états inutiles car inaccessibles :

Définition 2.9 *L'état $q \in Q$ est* accessible *dans l'automate $A = (V_t, Q, q_0, F, T)$ s'il existe un mot $w \in V_t^*$ tel que $q_0 \xrightarrow{w}{}^* q$.*

Les états inaccessibles peuvent être facilement éliminés en calculant les états accessibles depuis l'état initial. Dans l'automate déterminisé de la figure 2.5, les états $\{S\}$ et $\{S, P\}$ sont accessibles, et $\emptyset$ et $\{P\}$ sont inaccessibles.

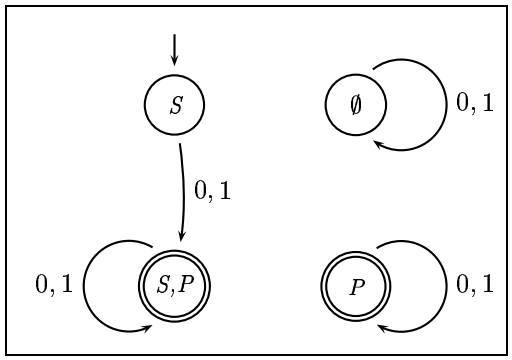


FIG. 2.5 – *Automate déterminisé.*

2.6 Minimisation des automates

Dans tout ce paragraphe, il est fondamental de supposer les automates COMPLETS, même si leur représentation en graphes correspond parfois à un automate incomplet par souci de clarté des dessins. On supposera également que tous les états des automates manipulés sonmt accessibles.

On voit que la déterminisation fait exploser le nombre d'états, mais que tous les états de $\mathcal{P}(Q)$ ne sont pas utiles à la construction du déterminisé d'un automate dont Q est l'ensemble des états. De plus, il se trouve que l'automate construit est en fait souvent plus compliqué que nécessaire. Un premier exemple est l'automate déterminisé de la figure 2.5 où l'on peut obtenir un automate plus simple en supprimant les états inaccessibles. L'élimination des états inaccessibles n'est pas toujours suffisant pour obtenir l'automate le plus simple possible. En haut de la figure **??** sont représentés trois automates équivalents reconnaisent le même langage $\{0(01)^n00 \mid n \geq 0\} \cup \{1(01)^n00 \mid n \geq 0\}$ (la vérification, facile au demeurant, que ces trois automates reconnaissent bien ce langage est laissée au lecteur). Le premier de ces trois automates comporte moins d'états, il est en fait minimal. Les deux automates (non-minimaux) appelés `AutomateG1` et `AutomateG2` ont en fait été obtenus par une déformation d'`Automate` initial. Les autres automates apparaissant sur la figure correspondent à des transformations réduisant le nombre d'états que nous allons maintenant expliquer.

Il est clair que les états finaux de l'automate `AutomateG1`, ainsi que ceux de l'automate `AutomateD1` sont intercheangeables. Étant à l'un quelconque de ces états, on est en état final, donc on reconnait le mot vide, et toute transition nous envoie dans la poubelle (non représentée sur le dessin). Plus précisément, ils reconnaissent le même langage $\{\varepsilon\}$. On va donc les confondre, c'est ce qui est fait à cette première étape, qui nous rend donc les automates `AutomateG2` et `AutomateD2`.

D'une manière générale, si deux états d'un automate reconnaissent le même langage, on les dira *intercheangeables* et on peut alors *réduire* l'automate. Notons par $\mathcal{L}^q_{\mathcal{A}}$ le langage des mots reconnus par l'automate obtenu à partir de $\mathcal{A}$ en adoptant l'état q comme état initial. Dans l'exemple précédent, $\mathcal{L}^4_{\texttt{A}utomateG1} = \mathcal{L}^7_{\texttt{A}utomateG1} = \{\varepsilon\}$. Le problème, bien sûr, va être de déterminer si deux états q et q' sont intercheangeables. La réponse à cette question est difficile, et nous nous contenterons d'approximer la notion d'intercheangeabilité par la relation d'équivalence $\equiv$ ainsi définie:

$$q \equiv q' \text{ ssi } (q \in F\ mboxssiq' \in F) \text{ and } \forall a \in V_t\ T(q,a) = T(q',a)$$

Il est aisé de vérifier que la relation ainsi définie est réflexive ($q \equiv q$), symmétrique ($q \equiv q'$ implique $q' \equiv q$) et transitive ($q \equiv q'$ et $q' \equiv q$" implique $q \equiv q$"), et que deux états équivalents sont intercheangeables, c'est-à-dire reconnaissent le même langage.

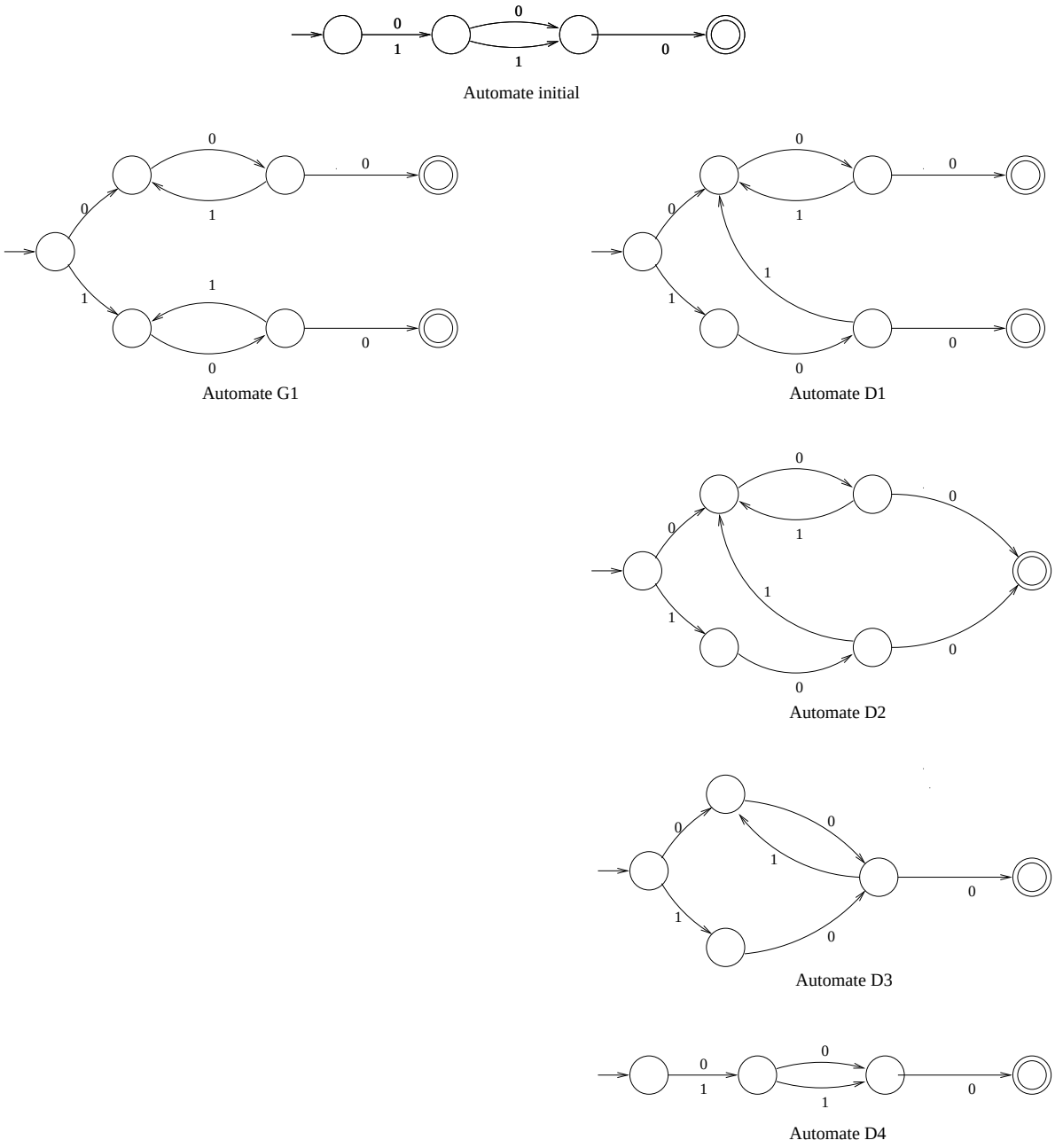


FIG. 2.6 – *Réduction d'un automate non-minimal.*

Supposons que les états $q_1, \ldots, q_p$ de l'automate $\mathcal{A}$ soient équivalents. On va construire l'automate réduit $\mathcal{A}_{rduit} = (V_t, Q \setminus \{q_2, \ldots, q_p\}, q_0, T_{rduit})$ ainsi défini :

(i) q_1 est initial si l'un des états $q_1, \ldots, q_p$ est initial,

(ii) $\begin{array}{ll} T_{rduit}(q,a) = T(q,a) & \text{si} \quad q \notin \{q_1, \ldots, q_p\} \text{ et } T(q,a) \notin \{q_1, \ldots, q_p\} \\ T_{rduit}(q,a) = q_1 & \text{si} \quad T(q,a) \in \{q_1, \ldots, q_p\} \end{array}$ Notons que les arcs sortant d'un état q_i étant redondants dans l'automate de départ avec ceux sortant de l'état q_1, il n'est nul besoin de les recopier dans l'automate résultat.

Il est facile de vérifier que cet automate est lui-aussi déterministe (aucun arc sortant n'ayant été ajouté) et qu'il reconnait le même langage que l'automate de départ. En effet, tout mot reconnu dans l'automate $\mathcal{A}$ sans passer par un état de $q_1, \ldots, q_n$ sera encore reconnu de la même manière. Un mot reconnu par un chemin empruntant un arc $q \xrightarrow{a} q_i$, avec $q \notin \{q_1, \ldots, q_p\}$ sera maintenant reconnu avec le changement de cet arc en l'arc $q \xrightarrow{a} q_1$, la suite de la reconnaissance se poursuivant normalement à partir de l'état q_1 puisque les arcs sortants de q_1 et de q_i sont les mêmes par hypothèse.

Dans notre cas, il est facile de réduire l'automate `AutomateD1` en l'automate initial, grâce à 3 réductions successives. Notons que chaque réduction (sauf la dernière) fait apparaitre de nouvelles réductions. Par contre, une seule réduction s'applique à l'automate `AutomateG1`, et l'automate minimal ne peut donc pas être atteint. Cela montre les limites de la méthode, lorsque nous détectons des états interchangeables grâce à l'équivalence $\equiv$.

Le but de la réduction précédente est de partitionner l'ensemble des états en états équivalents, c'est-à-dire reconnaissant le même langage. La partition obtenue a la propriété clé de *compatibilité* :

Définition 2.10 *Étant donné un automate déterministe complet $\mathcal{A} = (V_t, Q, q_0, F, T)$, une partition $\mathcal{P}$ de Q est* compatible *si toute classe de la partition est* compatible. *Une classe C est* compatible *si elle vérifie la propriété suivante :*

$$\forall q, q' \in C \ \forall a \in V_t \ \exists C', \ T(q,a) \in C' \text{ ssi } T(q',a) \in C'$$

Ëtant donné une partition compatible des états Q de l'automate $\mathcal{A}$, on peut aisément construire un nouvel automate qui reconnait le même langage que $\mathcal{A}$:

Proposition 2.11 *Étant donnée une partition compatible $\mathcal{P}$ des états de l'automate $\mathcal{A} = (V_t, Q, q_0, F, T)$, l'*automate quotient *$\mathcal{A}_q = (V_t, \mathcal{P}, C_0, \{C \in \mathcal{P} \mid C \subseteq F\}, T_q$ tel que $T_q(C,a) = C'$ ssi $\exists q \in C \ \exists q' \in C'$ tel que $T(q,a) = q'$ reconnait le $\mathcal{L}(\mathcal{A})$.*

Notons que la compatibilité de la partition assure que la classe C' ne dépend pas du choix de l'état q de C, et que de plus, C' existe toujours puisque $\mathcal{A}$ est complet.

Nous sommes donc ramenés à trouver la partition compatible de Q dont le nombre de classes est minimum. Le problème est que la détermination a-priori d'une telle partition n'est pas simple, et peut difficillement être obtenue par un algorithme de réduction comme le précédent. Nous allons au contraire procéder *à l'envers*, c'est ce que nous expliquons maintenant.

L'idée est de partir d'une partition des états en deux classes, les états acceptants d'une part, et les autres, puisqu'une classe compatible ne peut contenir à la fois des acceptants et des non-acceptants. On va ensuite affiner cette partition chaque fois que nécessaire, c'est-à-dire lorsque une classe C ne sera pas compatible. Ce sera le cas lorsqu'une même lettre a provoque la transition de $p \in C$ vers p' et de $q \in C$ vers q', p' et q' n'appartenant pas à une même classe, donc ne reconnaissant pas le même langage. On doit alors partitionner C en sous classes compatibles, ce qui fera apparaître au moins deux nouvelles classes, C' contenant p et C'' contenant q. Ce partitionnement va rendre incompatibles des classes auparavent compatibles, et il va donc falloir tester toutes les classes à l'exception de celles

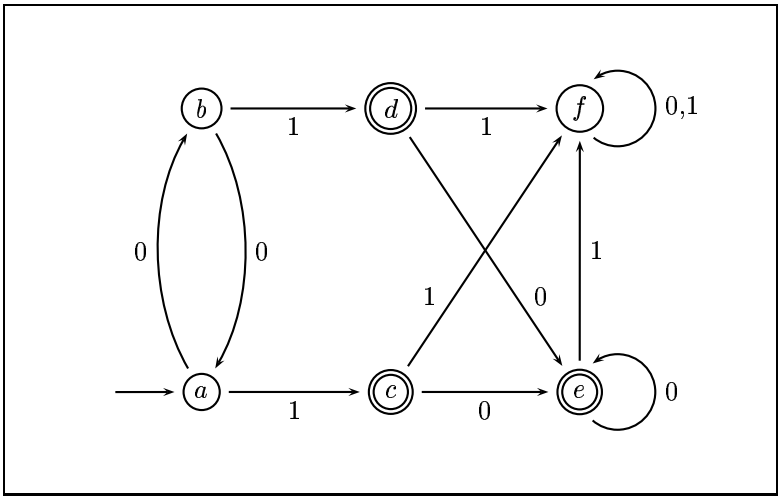


FIG. 2.7 – *Un automate non minimal.*

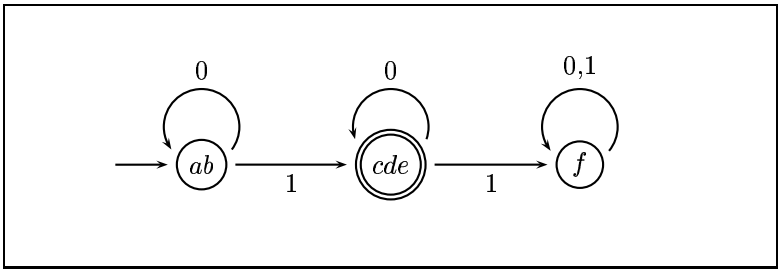


FIG. 2.8 – *Automate minimisé.*

issues du partitionnement de C. L'algorithme s'arrête (nécessairement) lorsque toutes les classes sont compatibles.

Étant donnée une partition P des états d'un automate, la fonction suivante recherche une classe de P qui n'est pas compatible. Dans le cas où toutes les classes sont compatibles la fonction retourne $\emptyset$. Puisque tous les classes dans les partitions construites par l'algorithme sont non-vides, le résultat $\emptyset$ signale le cas d'échec de la recherche.

```
type classe sur Q = ensemble sur Q
type partition sur Q = ensemble sur classe sur Q
fonction non_compatible(A: automate sur (V_t, Q),
                        P: partition sur Q) : classe sur Q
% Recherche d'une classe non compatible de P
 si il existe C ∈ P, p ∈ C, q ∈ C et a ∈ V_t tels que
  transition(A, p, a) ∈ C' et transition(A, q, a) ∈ C'' avec C' ≠ C''
  retourner C
 sinon
  retourner ∅
 fin si
fin fonction
```

La fonction suivante calcule alors, à partir d'une partition P donnée, le *raffinement compatible le plus grossier* de P, c'est-à-dire la sous-partition compatible de P ayant le plus petit nombre possible de classes (on peut montrer qu'elle est unique) :

```
fonction partition(A: automate sur (V_t,Q),
                   P: partition sur Q): partition sur Q
% Calcul d'un raffinement compatible de la partition P
 soit C = non_compatible(A, P)
 si C = ∅
  retourner P
 sinon
  soit P' la partition la plus grossière de C telle que
   pour tout C' ∈ P', pour tout C'' ∈ P, pour tous p, q ∈ C'
   pour tout a ∈ V_t :
    transition(A, p, a) ∈ C'' ssi transition(A, q, a) ∈ C''
  retourner partition(A, (P − {C}) ∪ P')
 fin si
fin fonction
```

La fonction suivante calcule l'automate minimisé.

```
fonction minimisation(A: automate sur (V_t, Q)) : automate sur (V_t, classe sur Q)
% Calcul du minimisé de A
 % Calcul de la plus grossière partition compatible
 soit F = {q ∈ Q | est_acceptant(A, q)}
 P = partition(A, {Q − F, F})
 % Calcul de l'état initial de l'automate minimal
 soit I_min la classe de P telle que etat_initial(A) ∈ I_min
 % Calcul des états acceptants de l'automate minimal
 pour chaque C ∈ P :
  soit F_min(C) = true ssi C ∩ F ≠ ∅
 fin pour
 % Calcul de la fonction de transition de l'automate minimal
```

pour chaque $(C, a) \in P \times V_t$ **faire**
soit $q \in C$ (un représentant quelconque)
soit C' la classe de P telle que $\mathit{transition}(A, q, a) \in C'$
$T_{min}(C, a) = C'$
fin pour
retourner $(I_{min}, F_{min}, T_{min})$
fin fonction

Appliqué à l'exemple précédent, cet algorithme calcule la séquence de partitions

$$\frac{\{a,b,f\},\ \{c,d,e\}}{\{a,b\},\ \{f\},\ \{c,d,e\}}$$

et retourne l'automate de la figure 2.8.

Théorème 2.12 *Soit $A = (V_T, Q, q_0, F, T)$ un automate déterministe et complet. L'automate envoyé par la fonction minimisation est l'automate déterministe complet avec un nombre minimal d'états acceptant le même langage que A.*

2.7 Exercices

Exercice 2.1

1. Donner une grammaire régulière décrivant les réels en virgule flottante, avec un signe éventuel.

   Correction : *Pour une grammaire régulière, les membres droits doivent être w ou wN où $w \in V_t^*$ et $N \in V_n$. On peut prendre :*

$$\begin{array}{rcl} \mathit{Réel} & \to & +\ \mathit{Réel\_sans\_signe} \mid -\ \mathit{Réel\_sans\_signe} \mid \mathit{Réel\_sans\_signe} \\ \mathit{Réel\_sans\_signe} & \to & 0\ \mathit{Réel\_sans\_signe} \mid \cdots \mid 9\ \mathit{Réel\_sans\_signe} \mid 0.\ \mathit{Entier} \mid \cdots \mid 9.\ \mathit{Entier} \\ \mathit{Entier} & \to & 0\ \mathit{Entier} \mid \cdots \mid 9\ \mathit{Entier} \mid 0 \mid \cdots \mid 9 \end{array}$$

2. Donner une grammaire régulière réduite équivalente.

   Correction : *Les membres droits doivent être ε ou bien αN avec $\alpha \in V_t$.*

$$\begin{array}{rcl} \mathit{Réel} & \to & +\ \mathit{Réel\_sans\_signe} \mid -\ \mathit{Réel\_sans\_signe} \mid \\ & & 0\ \mathit{Suite\_réel\_sans\_signe} \mid \cdots \mid 9\ \mathit{Suite\_réel\_sans\_signe} \\ \mathit{Réel\_sans\_signe} & \to & 0\ \mathit{Suite\_réel\_sans\_signe} \mid \cdots \mid 9\ \mathit{Suite\_réel\_sans\_signe} \\ \mathit{Suite\_réel\_sans\_signe} & \to & 0\ \mathit{Suite\_réel\_sans\_signe} \mid \cdots \mid 9\ \mathit{Suite\_réel\_sans\_signe} \mid .\ \mathit{Entier} \\ \mathit{Entier} & \to & 0\ \mathit{Suite\_entier} \mid \cdots \mid 9\ \mathit{Suite\_entier} \\ \mathit{Suite\_entier} & \to & 0\ \mathit{Suite\_entier} \mid \cdots \mid 9\ \mathit{Suite\_entier} \mid \varepsilon \end{array}$$

3. Dessiner un automate reconnaissant les réels.

Exercice 2.2
Transformer la grammaire suivante en une grammaire régulière réduite:

$$S \to a \mid T \mid \varepsilon \qquad T \to bcT \mid U \qquad U \to e \mid S \mid \varepsilon$$

Exercice 2.3
On veut montrer que le langage $L = \{a^n b^n \mid n \in \mathbb{N}\}$ ne peut pas être engendré par une grammaire régulière.

1. Soit $A = (V_t, Q, q_0, F, T)$ un automate déterministe quelconque. Trouver une borne $N \in \mathbb{N}$ associée à A qui assure que pour tout mot $w \in \mathcal{L}ang(A)$ de longueur supérieure ou égale à N, l'exécution de A sur w passe nécessairement au moins deux fois par un certain état.

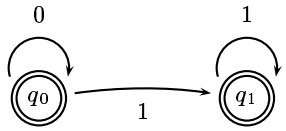
2. Raisonnons par l'absurde en supposant que le langage L est reconnaissable par un automate A. Par un raisonnement analogue à la question précédente, trouver un entier N tel que lors de l'exécution de A sur le mot $a^N b^N$, un état est parcouru au moins deux fois lors de la lecture des lettres a.

3. En déduire qu'il existe un mot de la forme $a^{N+k}b^N$ avec $k > 0$ reconnu par l'automate, et conclure.

Exercice 2.4
Donner un automate qui reconnaisse l'ensemble des entiers naturels en représentation octale.

Exercice 2.5
Pour chacun des langages suivants sur l'alphabet $\{0, 1\}$, trouver un automate qui le reconnaisse.

1. $\{0^n 1^m \mid n, m \geq 0\}$
   Correction :



2. l'ensemble des mots contenant 11,

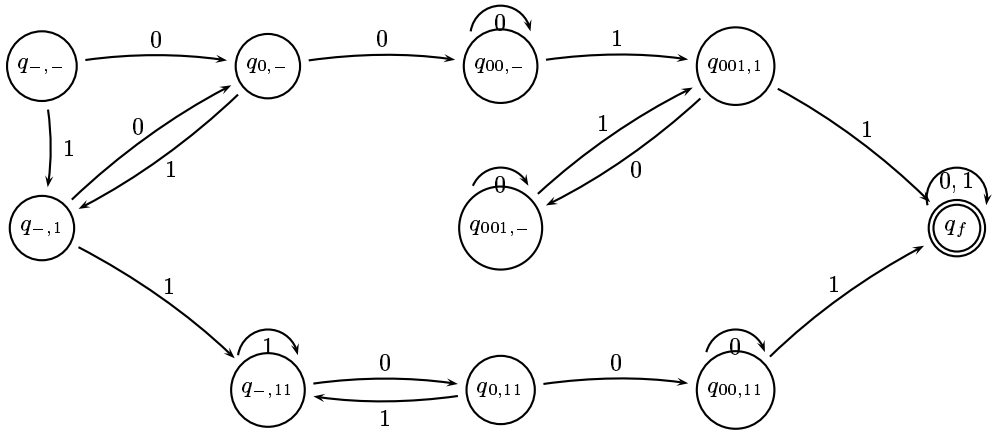
3. l'ensemble des mots contenant 001,

4. l'ensemble des mots contenant 001 et 11,

   Correction :

   On peut indicer les états par 2 composantes qui disent si on a déjà rencontré l'une des séquences, et qui se rappellent les 2 ou 3 dernières lettres. Remarquer que le mot 0011 convient.

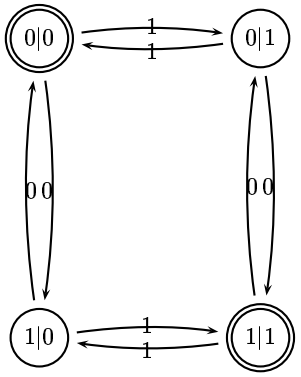
A verifier



5. l'ensemble des mots contenant 001 ou 11.

6. L'ensemble des mots tel que la parité du nombre de 0 est égale à celle de 1.

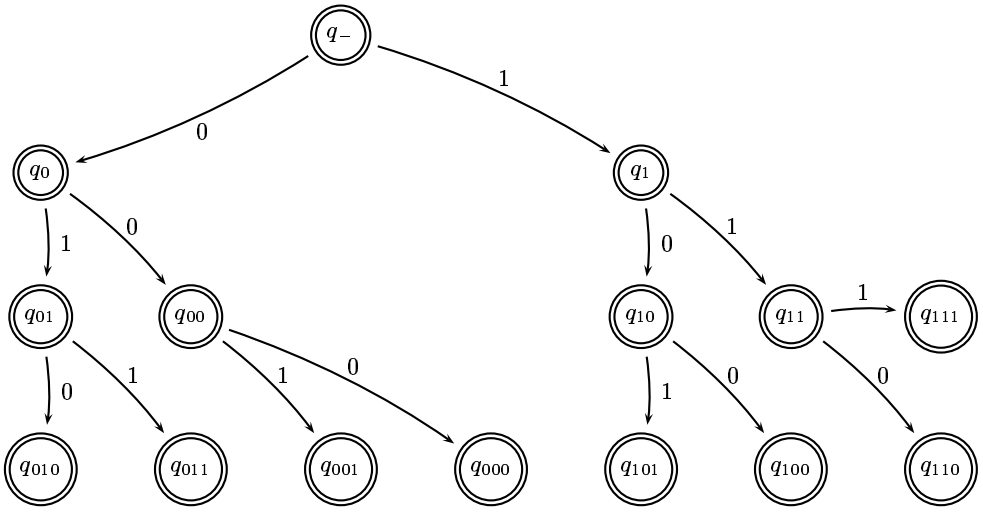
Correction :



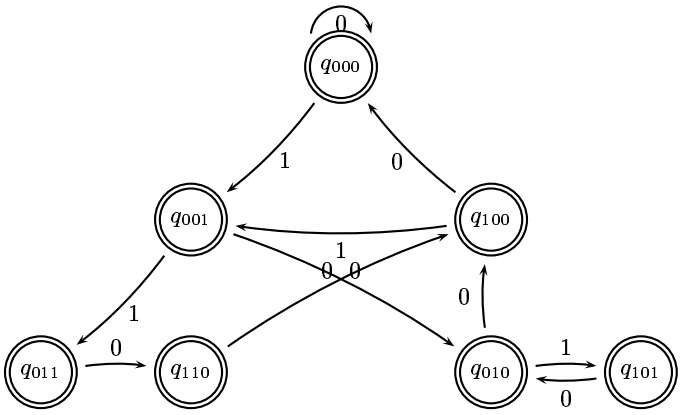
7. l'ensemble des mots où chaque bloc de 4 caractères successifs contient au moins deux fois 0,

Correction :

En particulier il faut accepter tout les mots de moins de 4 lettres. L'automate est en deux parties, une qui atteint tous les etats de trois lettres, l'autre est un cycle reliant ces etats a trois lettres. Il suffit de se rapeller de trois lettres, car la quatrieme est lue.

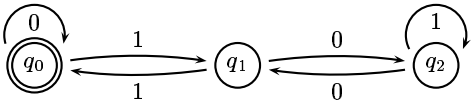


Avec :



8. l'ensemble des nombres naturels en représentation binaire qui sont divisibles par 3.

   Correction : *On mémorise le reste de la division par 3 dans un état, sachant que la lecture d'un 0 fait multiplier par 2, et la lecture d'un 1 fait $*2+1$. Remarque: pour simplifier 000... est un nombre binaire accepté. On pourra montrer la transformation pour avoir la solution stricte.*



Exercice 2.6
On appelle *chien de garde* un systeme qui recoit les signaux $\{e, f, r, s\}$ et qui est spécifié ainsi :

- Initialement le système est en état de repos et ignore les messages recus.
- À l'arrivée du signal e (éveil), le système passe à l'état de veille.
- Quand le système est en veille, tout signal s le fait passer dans un état d'alerte.
- Le signal f (fin) le fait toujours passer dans l'état de repos.
- Le système reste en état d'alerte tant que le signal r (repos) n'est pas émis.

Donner un automat réalisant un tel chien de garde.
Correction : *Une introduction. Implicitement l'état final est celui de repos.*

Exercice 2.7

1. Donner un automate déterministe qui reconnaît une date de la forme "jour/mois" où le jour est noté par un entier entre 1 et 31 et le mois est noté par un entier entre 1 et 12. Attention aux mois avec moins de 31 jours! Par exemple, votre automate doit accepter "7/4" (pour le 7 avril) et "31/12" (pour le 31 décembre), mais pas "30/2".

2. Étendre l'exercice aux dates de la forme "jour/mois/année", où l'année est notée par un entier entre 1000 et 1999. On suppose que les années bissextiles sont les années qui sont divisibles par 4.

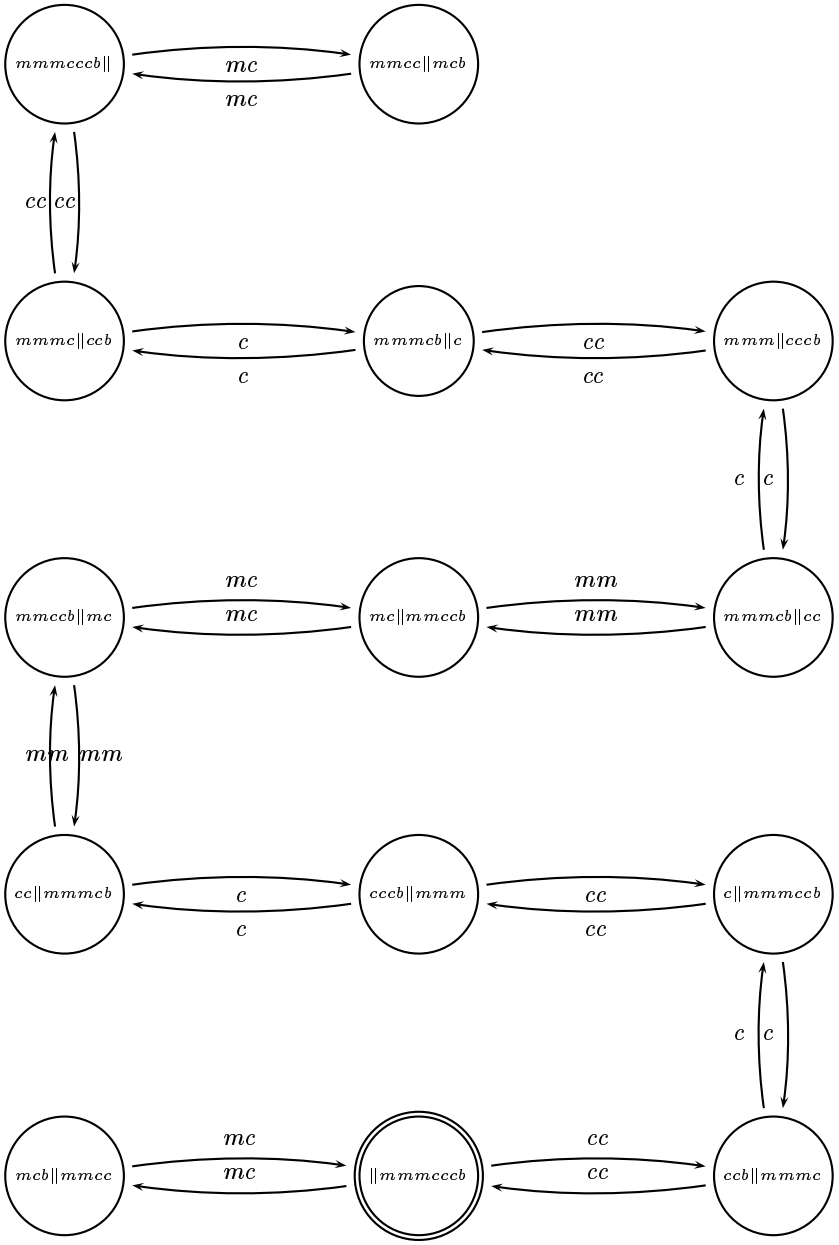
Exercice 2.8
Trois missionnaires et trois cannibales veulent traverser une rivière pleine de piranhas. Il y a une seule pirogue qui peut transporter deux personnes au maximum. Un missionnaire ne

peut pas utiliser la pirogue tout seul (mais deux le peuvent). Dès qu'il y a, à quelque endroit, plus de cannibales que de missionnaires, les cannibales vont manger les missionnaires.

- Est-ce qu'il y a une possibilité pour les six personnes de traverser la rivière sans être mangées ?

  Indication : utiliser un automate où les états sont les différentes répartitions possibles des missionnaires et des cannibales d'un côté et de l'autre de la rivière, et les transitions représentent les traversées de la pirogue.

  Correction :

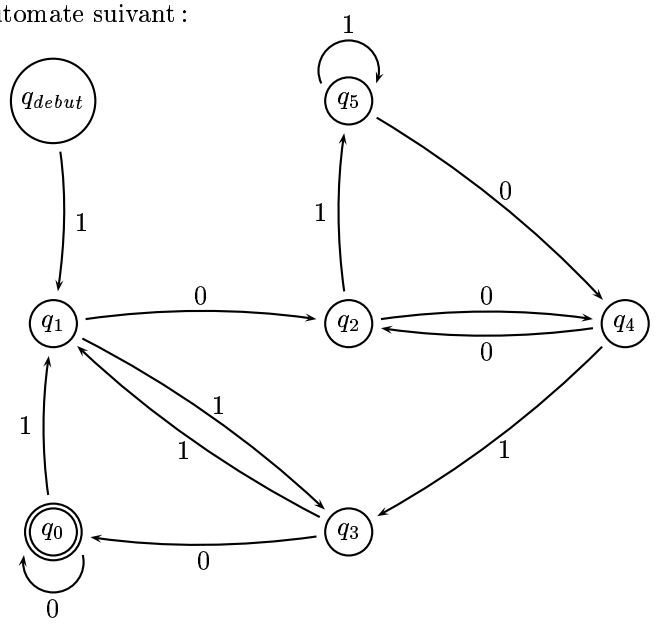


Un missionnaire est figuré par un M, un cannibale par un C et le bateau par un B.

- Si l'on considère le mot des actions, que penser d'une solution miroir ? Justifier.

Correction : *C'est encore une solution, traverser d'un coté pour aller à l'autre c'est la même chose que le contraire. Reste a le justifier.*
Pour avoir l'ensemble des solutions, on a un automate complet, et localement réversible ($q \xrightarrow{a} q' \Rightarrow q' \xrightarrow{a} q$) Soit $\tilde{q}_{b\|a} = q_{a\|b}$ pour chaque $q_{a\|b}$ de notre automate.
On montre par récurence sur $\|w\|$, mot quelconque, si $q_i \xrightarrow{w} q_j \Rightarrow \tilde{q}_j \xrightarrow{mir(w)} \tilde{q}_i$
Dans le cas où $w = w_1.a$, si $q_i \xrightarrow{w_1.a} q_j$ alors $q_i \xrightarrow{w_1} q_k \xrightarrow{a} q_j$ et par récurence $\tilde{q}_j \xrightarrow{a} \tilde{q}_k \xrightarrow{mir(w_1)} \tilde{q}_i$

Exercice 2.9
On considère l'automate suivant :



1. Verifiez que les mots 110, 1100, 10010 sont reconnus par cet automate.

2. Caractériser le language L_0 reconnu par cet automate?

   Correction : *C'est l'ensemble des mots non nuls dont la valeur en binaire est multiple de 6. Si $b(w)$ est la valeur de w en binaire, on montre que si $q_{debut} \xrightarrow{w} q_1$ alors le reste de la division de $val(w)$ par 6 vaut i. Le cas d'une lettre est facile, si $w = w_1.a$, alors puisque l'automate est déterministe et complet, il existe q_i et q_j tels que $q_{debut} \xrightarrow{w_1} q_i \xrightarrow{a} q_j$. On utilise alors l'hypothèse de recurence sur w_1, puis on vérifie que toutes les transitions suivent la loi : $q_j \xrightarrow{0} q_i$ ssi $j * 2 = i$ et $q_j \xrightarrow{1} q_i$ ssi $j * 2 + 1 = i$ (modulo 6)*

3. Si w est un mot, on note $mir(w)$ le mot miroir de w : si $w = a_1.a_2.\ldots.a_n$ alors $mir(w) = a_n.\ldots.a_2.a_1$.
   Et on definit le langage miroir comme : $mir(L) = \{mir(w), w \in L\}$.
   Donner un automate pour le langage $mir(L_0)$.

   Correction : *Il suffit d'inverser les transitions, de prendre q_0 pour état initial et q_{debut} pour état final. On obtient un automate non-déterministe*
   En effet, si $w \in L$, alors il existe un chemin allant de q_{debut} à q_0 tel que $q^{i-1} \xrightarrow{a_i} q^i$. Alors en inversant les transitions on a un chemin qui reconnait $mir(w)$ dans l'automate B. De même, si on a $w \in L(B)$, alors il existe un calcul qui, transposé dans A dit que $w' = mir(w) \in L(A)$. Donc $\exists w' \in L(A), w = mir(w')$

Exercice 2.10
Donner un algorithme pour calculer l'ensemble des états accessibles d'un automate.

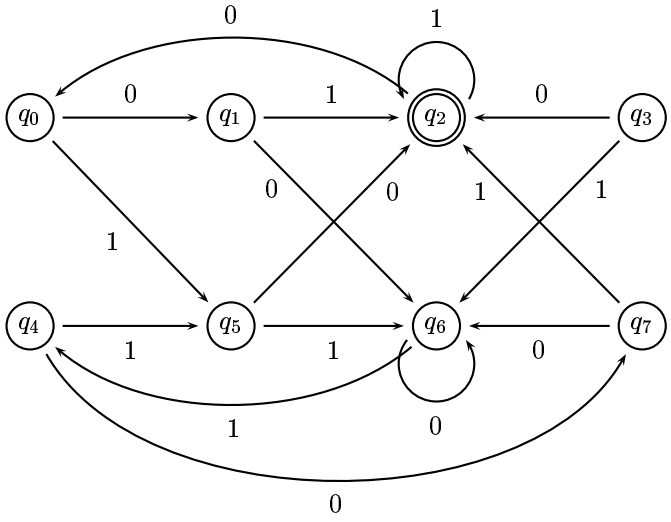
Exercice 2.11
En pratique, le calcul de la partition compatible la plus grossière possible se fait à l'aide d'un tableau ayant autant de lignes et de colonnes que d'états. Il est facile, à l'aide d'un tel tableau, de marquer les états incompatibles entre eux. À l'initialisation, les états non-acceptants seront marqués comme incompatibles avec les états acceptants. Puis on marque récursivement tout couple d'états (p, q) tels que le couple $(transition(p, a), transition(q, a))$ soit marqué pour une certaine lettre $a \in V_t$. Le marquage une fois terminé, la ligne p du tableau de marquage indique les états qui sont dans la même classe de la partition que p: ce sont les états q tels que les couples (p, q) ne sont pas marqués. Formaliser cet algorithme.

Correction :

```
pour chaque q ∈ Q faire
  pour chaque q ∈ Q faire
    si acceptant(p) et non acceptant(q)
      marquer(p,q)
    fin si
  fin pour
fin pour

répéter
  stable = true
  pour chaque q ∈ Q faire
    pour chaque p ∈ Q faire
      pour chaque a ∈ Vt faire
        si marqué(p,q) et non marqué(transition(q,a), transition(p,a))
          marquer(p,q)
          stable = false
        fin si
      fin pour
    fin pour
  fin pour
jusqu'à stable
```

Exercice 2.12
Donner l'automate minimal équivalent à l'automate suivant:



Chapitre 3

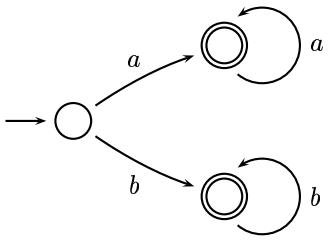
Expressions rationnelles et analyse lexicale

Ce chapitre a pour objectif de décrire la première phase de l'interprétation d'un langage : l'analyse lexicale. Pour ce faire, la classe des langages réguliers étudiée au chapitre précédent va nous servir à définir et à reconnaître les différentes unités lexicales d'un programme, car celles-ci constituent justement des langages réguliers. La définition de ces unités lexicales va nécessiter l'introduction d'un nouveau formalisme de description des langages réguliers : les expressions rationnelles.

Les grammaires régulières ne permettant pas une écriture simple ni concise des langages réguliers (par exemple pour les constantes réelles), nous allons donc en introduire une nouvelle. L'idée que l'on va exploiter pour décrire les langages réguliers de manière simple est d'utiliser les opérations de base des langages : concaténation, itération, union, etc. Nous allons donc étudier les propriétés de clôture des langages réguliers : étant donnés deux automates A_1 et A_2 reconnaissant respectivement les langages L_1 et L_2, comment peut-on construire un automate reconnaissant $L_1 \times L_2$, L_1^*, $L_1 \cup L_2$, etc?

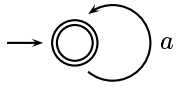
Conformément à notre habitude, nous allons d'abord essayer de comprendre ces mécanismes à l'aide d'un exemple simple : supposons $L_1 = \{a\}^+$ et $L_2 = \{b\}^+$ reconnus par les automates suivants :

Pour construire l'automate reconnaissant $L_1 \cup L_2$, il suffit alors de réunir les deux états initiaux en un seul ce qui donne l'automate

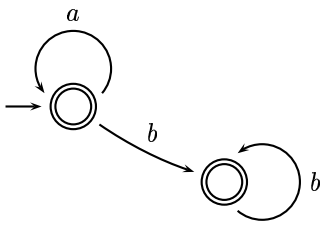


Malheureusement, cette idée de réunir en un seul les deux états initiaux ne fonctionne pas en général: si l'on remplace dans la construction précédente le langage L_1 en le langage

$\{a\}^*$ reconnu par l'automate

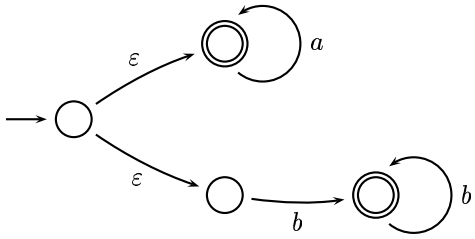


l'automate obtenu par réunion des états initiaux est alors



qui accepte le mot ab alors que celui-ci n'est pas dans la réunion des deux langages. La construction est donc erronée.

Ce phénomène se produit en fait dès l'un des automates possède un circuit passant par son état initial. Pour résoudre ce problème, il nous faut des automates munis d'une opération de choix, permettant de passer d'un état à d'autres états *sans lire une lettre du mot à reconnaître*. Nous allons donc tout d'abord généraliser un peu plus notre définition d'automate en introduisant les automates avec *transitions vides*. La construction correcte de l'union des deux langages précédents sera alors



3.1 Automates non-déterministes avec transitions vides

Un automate avec transitions vides est un automate où certaines transitions peuvent être étiquetées par ε au lieu d'une lettre.

Définition 3.1 *Un* automate non-déterministe A avec transitions vides *est un quintuplet* (V_t, Q, q_0, F, T) *où*

1. V_t *est le* vocabulaire *de* A,
2. Q *est l'ensemble des* états *de* A,
3. $q_0 \in Q$ *est l'*état initial *de* A,
4. $F \subseteq Q$ *est l'ensemble des* états acceptants *de* A,
5. T, *la* fonction de transition *de* A, *est une application de* $Q \times (V_t \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$. *On notera* $q \xrightarrow{\alpha} q'$ *pour* $q' \in T(q, \alpha) = q'$, *avec* $\alpha \in V_t$ *ou* $\alpha = \varepsilon$.

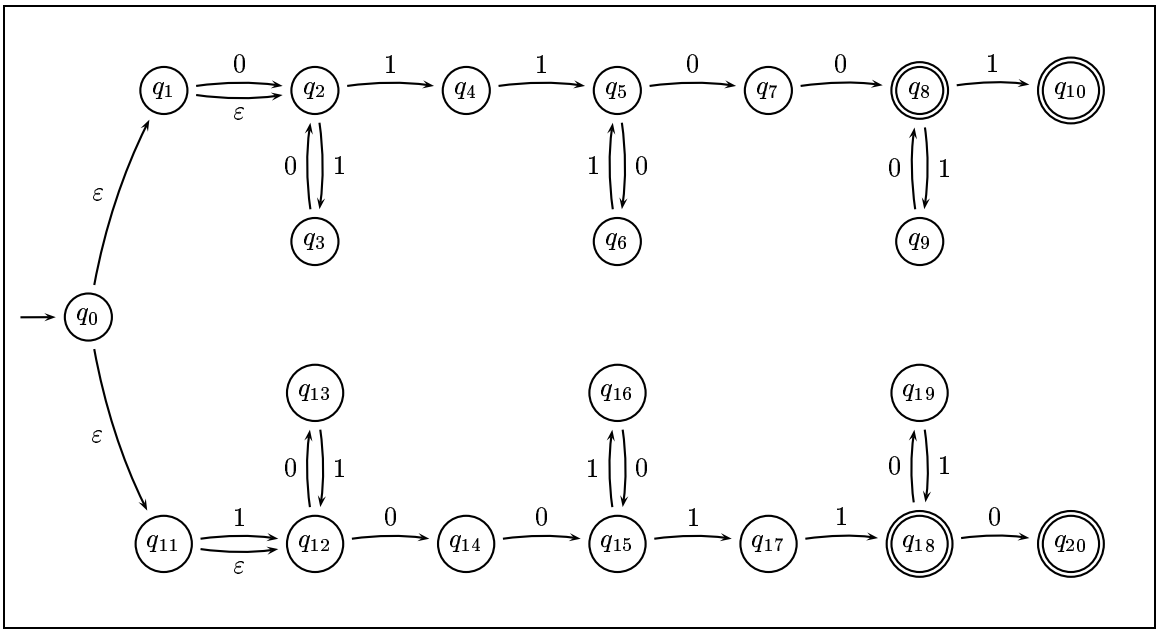


FIG. 3.1 – *Un automate non-déterministe avec transitions vides.*

L'exécution d'un automate avec transitions vides autorise le passage par un nombre quelconque de transitions vides au cours de l'exécution. Le nombre d'états parcourus n'est bien sûr plus égal au nombre de lettres du mot d'entrée mais peut être plus grand.

Définition 3.2 *Étant donné un automate non-déterministe avec transitions vides* $A = (V_t, Q, q_0, F, T)$*, on appelle* exécution *toute suite (éventuellement vide) de transitions* $q_0 \xrightarrow{\alpha_1} q_1 \dots q_{n-1} \xrightarrow{\alpha_n} q_n$ *avec* $\alpha_i \in V_t \cup \{\varepsilon\}$*, qui* lit *le mot* $w = \alpha_1 \cdots \alpha_n$ *(donc de longueur inférieure ou égale à* n*). Un mot* w *est reconnu par l'automate* A *s'il existe une exécution de l'automate* A *qui se termine en état acceptant en ayant lu le mot* w*. Comme toujours, on note par* $\mathcal{L}ang(A)$ *le langage des mots reconnus par l'automate* A*.*

Nous voulons maintenant montrer que le langage reconnu par un automate avec transitions vides peut également l'être par un automate déterministe. Pour cela, nous allons d'abord éliminer les transitions vides et donc se ramener à un automate non-déterministe, que nous pouvons alors déterminiser selon la méthode du chapitre précédent.

Pour éliminer les transitions vides on calcule pour chaque état l'ensemble des états accessibles par le mot vide :

Définition 3.3 *Soit* $A = (V_t, Q, q_0, F, T)$ *un automate avec transitions vides. La* clôture par ε *ou en abrégé l'*ε*-clôture d'un état* $q \in Q$ *est*

$$\varepsilon\text{-}clôture(q) = \{p \in Q \mid q \xrightarrow{\varepsilon}{}^{*} p\}$$

Par exemple, la figure 3.1 représente un automate non-déterministe avec transitions vides reconnaissant le langage sur l'alphabet $\{0, 1\}$ contenant exactement une occurrence de chacun des mots 00 et 11.

On obtient pour les ε-clôtures des états:

état	q_0	q_1	q_2	q_{11}	q_{12}
ε-*clôture*	$\{q_0, q_1, q_2, q_{11}, q_{12}\}$	$\{q_1, q_2\}$	$\{q_2\}$	$\{q_{11}, q_{12}\}$	$\{q_{12}\}$

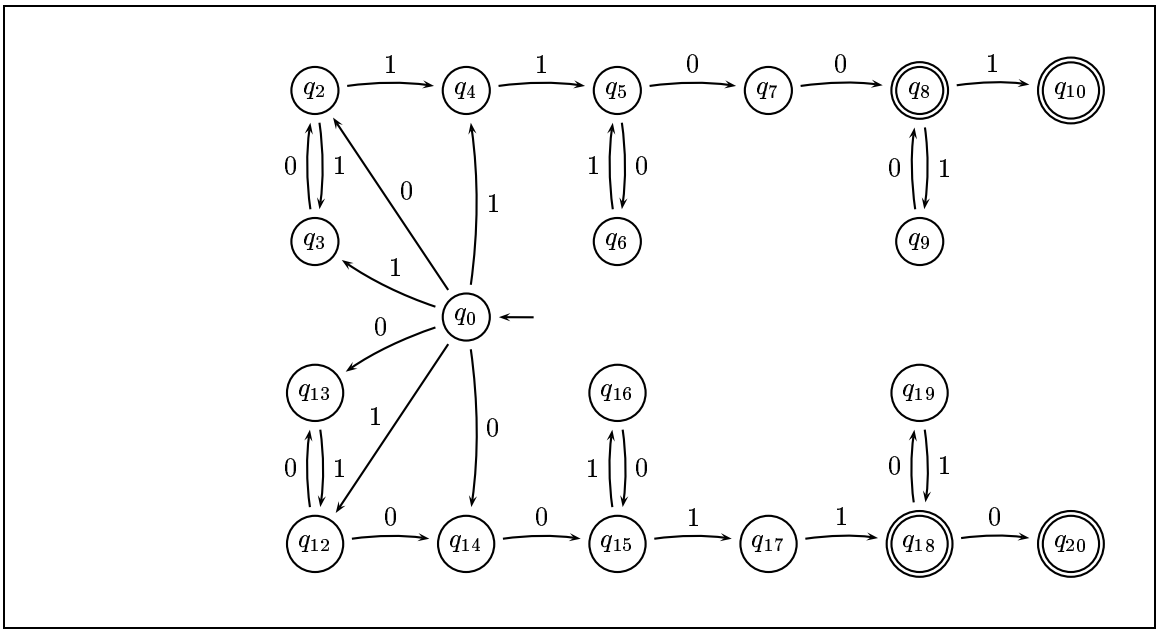


FIG. 3.2 – *Automate non-déterministe sans transitions vides.*

et ε-*clôture*$(q) = \{q\}$ pour les autres états.

Définition 3.4 *Soit $A = (V_t, Q, q_0, F, T)$ un automate non-déterministe avec transitions vides. On définit $E(A)$ comme l'automate $(V_t, Q, q_0, E(F), E(T))$ tel que*

$$\begin{aligned} E(T)(q,\alpha) &= \bigcup_{p \in \varepsilon\text{-clôture}(q)} T(p,\alpha) \text{ pour chaque } q \in Q \text{ et } \alpha \in V_t \\ E(F) &= \{q \in Q \mid \varepsilon\text{-clôture}(q) \cap F \neq \emptyset\} \end{aligned}$$

Théorème 3.5 *Pour tout automate A non-déterministe avec transitions vides, l'automate $E(A)$ n'a pas de transitions vides et reconnaît le même langage que A.*

Ainsi, pour déterminiser l'automate de la figure 3.1, on construit d'abord l'automate sans transitions vides mais toujours non-déterministe de la figure 3.2. Nous avons supprimé les états q_1 et q_{11} parce qu'ils ne sont plus accessibles.

L'automate déterminisé correspondant est représenté sur la figure 3.3, où seulement les états accessibles sont montrés. Les états sont étiquetés par l'ensemble des noms des états de l'automate non-déterministe qui le constituent.

Enfin, le minimisé de l'automate de la figure 3.3 est représenté sur la figure 3.4.

3.2 Propriétés de clôture des langages réguliers

Grâce aux automates avec transitions vides, nous sommes maintenant armés pour construire les automates reconnaissant des langages obtenus par concaténation, itération, et union de langages reconnus par des automates.

Pour ces constructions nous allons supposer que tous les automates mis en jeu possèdent un unique état acceptant : on peut toujours se ramener à ce cas au prix de transitions vides additionnelles. Dans les figures qui suivent, un automate arbitraire est figuré par une ellipse

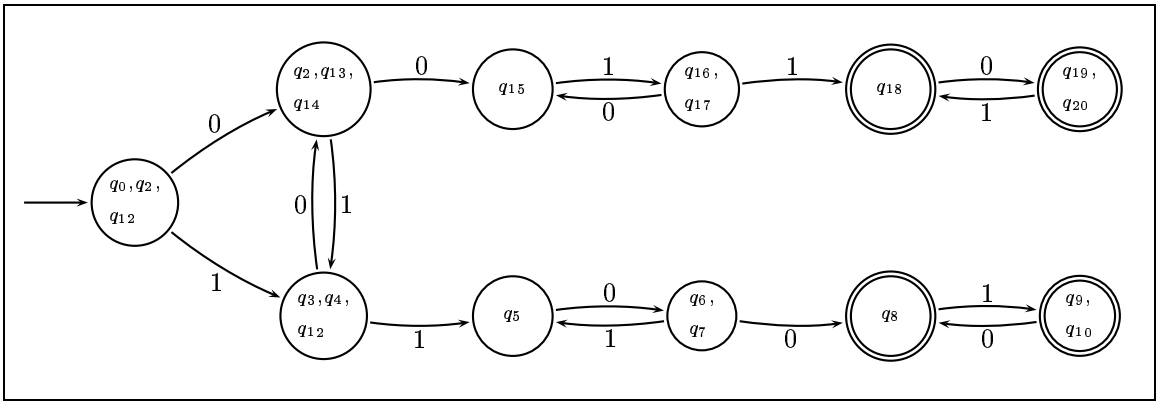


FIG. 3.3 – *Automate déterminisé.*

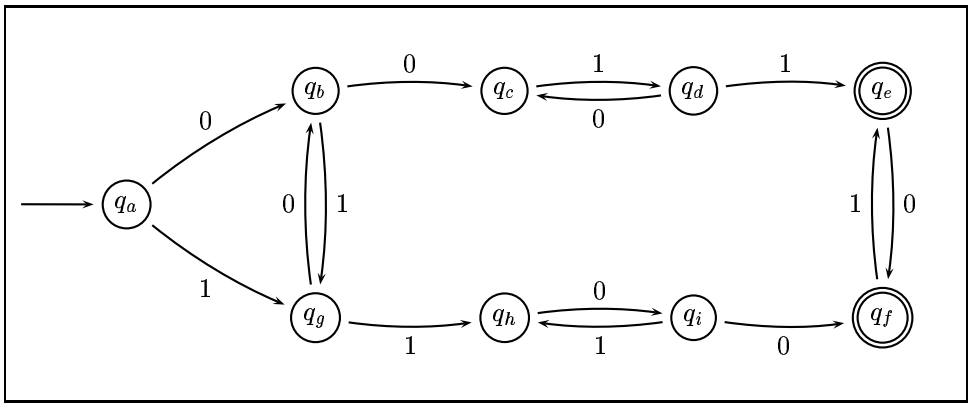
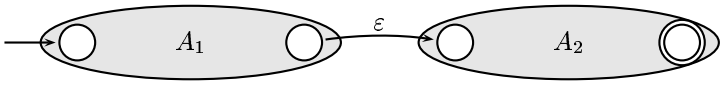


FIG. 3.4 – *Automate déterminisé et minimisé.*

dans laquelle nous avons fait figurer l'état initial à gauche et l'unique état acceptant à droite. Les constructions conserveront la propriété que l'automate construit possède un unique état acceptant.

3.2.1 Concaténation et Puissance

L'automate reconnaissant la concaténation de deux langages peut être obtenu est ajoutant une transition vide depuis l'état acceptant du premier automate vers l'état initial du deuxième :

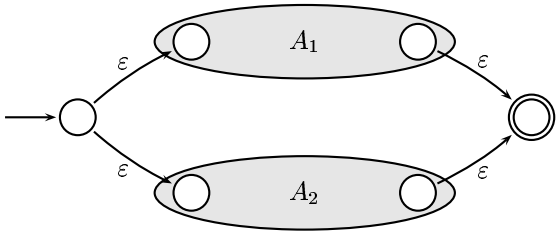


Il faut faire attention à ne pas identifier directement l'état acceptant du premier automate et l'état initial du deuxième, car cela peut permettre de reconnaître plus de mots que nécessaire, dans le cas où il est possible de boucler sur ces états (cf. exercice 3.11).

Il est ensuite facile de construire l'automate reconnaissant la puissance n-ième d'un langage régulier en appliquant $n-1$ fois la construction précédente.

3.2.2 Union

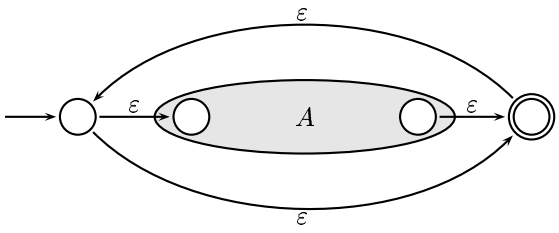
L'automate reconnaissant l'union de deux langages peut être obtenu en « branchant » vers les deux états initiaux. De plus pour conserver la propriété d'unicité de l'état acceptant, nous rajoutons deux transitions vers un nouvel état acceptant :



Toute construction plus simple serait erronée (cf. exercice 3.11).

3.2.3 Itération et itération stricte

L'automate reconnaissant l'itération d'un langage peut être obtenu de la façon suivante :



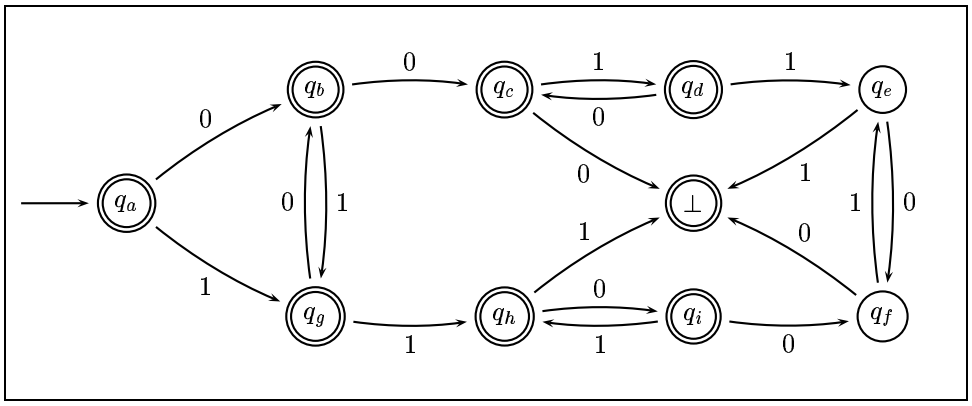


FIG. 3.5 – *Complémentation de l'automate de la figure 3.4*

Là encore, toute construction plus simple serait erronée (cf. exercice 3.11).

L'automate reconnaissant l'itération stricte d'un langage régulier peut être obtenu par une légère variation de la construction précédente, consistant à supprimer la transition vide allant de l'état initial à l'état final de l'automate construit.

3.2.4 Complémentation et intersection

La complémentation obéit à un principe de construction différent en ce qu'il ne fait pas intervenir de transitions vides. Cette construction suppose donné un automate déterministe complet :

Théorème 3.6 *Soit $A = (V_t, Q, q_0, F, \delta)$ un automate déterministe et complet. Le langage complémentaire $\overline{\mathcal{L}ang(A)}$ de $\mathcal{L}ang(A)$, est reconnaissable et reconnu par l'automate $A = (V_T, Q, q_0, Q - F, \delta)$.*

En effet, un mot de V_t^* est dans le complémentaire de $\mathcal{L}ang(A)$ si et seulement si l'exécution correspondante n'aboutit pas en un état acceptant de F. La complémentation est donc une opération particulièrement simple, et l'automate obtenu est lui-même déterministe. Par contre, pour calculer l'automate qui reconnaît le complémentaire d'un langage reconnu par un automate quelconque, il faut d'abord déterminiser et compléter cet automate.

Il faut bien faire attention que l'opération de complémentation n'est correcte que si l'automate est complet. Par exemple, le complémentaire du langage reconnu par l'automate de la figure 3.4 est l'automate de la figure 3.5, ou nous avons rajouté explicitement un état poubelle $\perp$ qui est devenu acceptant par complémentation. Sans cet ajout, l'automate obtenu ne reconnaît pas le complémentaire du langage, contenant en particulier le mot 000.

La clôture par intersection peut maintenant être traitée sans effort par combinaison des clôtures par union et complémentaire (cf. exercice 3.12).

3.3 Expressions rationnelles

En pratique, il n'est pas toujours commode de décrire un langage par un automate, déterministe ou pas. C'est en particulier le cas des automates que l'on manipule en analyse lexicale, qui doivent reconnaître un grand nombre de mots de petite taille. Ces automates

possèdent un grand nombre d'états et sont fastidieux à construire. Il est beaucoup plus facile de décrire un tel langage en énumérant les mots qu'il contient.

Définition 3.7 *L'ensemble des* expressions rationnelles $\mathcal{R}at$ *sur l'alphabet* V_t *est formé comme suit :*

1. *tout mot de* V_t^* *appartient à* $\mathcal{R}at$ *;*
2. *si* $e \in \mathcal{R}at$ *et* $e' \in \mathcal{R}at$*, alors* $e \mid e' \in \mathcal{R}at$ *;*
3. *si* $e \in \mathcal{R}at$ *et* $e' \in \mathcal{R}at$*, alors* $e \times e' \in \mathcal{R}at$ *;*
4. *si* $e \in \mathcal{R}at$*, alors* $e^* \in \mathcal{R}at$ *;*
5. *si* $e \in \mathcal{R}at$*, alors* $(e) \in \mathcal{R}at$*.*

Les expressions rationnelles permettent de dénoter de manière succincte des langages pas trop complexes : il s'agit donc d'un langage permettant de dénoter des langages, de même que le formalisme des grammaires est un langage permettant d'engendrer des langages.

Définition 3.8 *Le langage* dénoté *par une expression rationnelle* e*, noté* $\mathcal{L}ang(e)$*, est défini par :*

1. *le mot* $w \in V_t^*$ *dénote le langage à un mot* $\{w\}$ *;*
2. *l'expression* $e \mid e'$ *dénote le langage* $\mathcal{L}ang(e) \cup \mathcal{L}ang(e')$ *;*
3. *l'expression* $e \times e'$ *dénote le langage* $\mathcal{L}ang(e) \times \mathcal{L}ang(e')\}$ *;*
4. *l'expression* e^* *dénote le langage* $(\mathcal{L}ang(e))^*$ *;*
5. *l'expression* (e) *dénote le même langage que* e*.*

Un langage est dit rationnel *s'il peut être dénoté par une expression rationnelle.*

Par exemple, le langage des entiers naturels en représentation binaire est dénoté par l'expression rationnelle $0 + 1 \times (0 + 1)^*$. Se pose à nouveau un problème d'ambiguïté de cette notation, très semblable au problème que nous avons rencontré avec les expressions arithmétiques. Le parenthésage attendu est $0 + (1 \times ((0 + 1)^*))$. Il résulte là aussi de règles de priorité, dans l'ordre décroissant : $*$, $\times$, $+$, et de règles d'association à gauche pour $\times$ et $+$.

Nous avons vu que les langages réguliers et les langages reconnaissables étaient identiques. Il nous reste à les comparer aux langages rationnels. Le résultat fondamental de ce chapitre est le suivant :

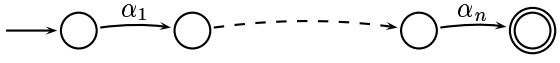
Théorème 3.9 *Soit* L *un langage. Les trois propriétés suivantes sont équivalentes :*

(i) L *est régulier ;*

(ii) L *est reconnaissable ;*

(iii) L *est rationnel ou bien vide.*

La construction d'un automate reconnaissant le langage dénoté par une expression rationnelle se fait grâce aux propriétés de clôtures vues précédemment. Il nous faut tout de même amorcer la construction, en donnant l'automate associé à un mot $\alpha_1 \cdots \alpha_n$:



Le passage d'un automate à une expression rationnelle dénotant le même langage est beaucoup plus complexe et ne sera pas abordé dans ce cours.

3.4 Analyse lexicale avec l'outil LEX

L'analyse lexicale aura pour but de préparer l'étape suivante, l'analyse syntaxique, en engendrant une séquence de *tokens* qui sera ultérieurement analysée par l'analyseur syntaxique. Un analyseur lexical est donc une sorte de gros automate dont les états acceptants reconnaissent les différents tokens (c'est-à-dire les différentes sortes d'unités lexicales) du langage.

Les différents tokens une fois spécifiés sous forme d'expressions rationnelles, un automate déterministe minimal peut être construit pour chacun d'eux. L'outil LEX est un outil disponible dans différents langages de programmation, qui automatise cette transformation.

3.4.1 Notion de token

Lors de l'analyse syntaxique d'un langage, nous avons vu qu'il était pratique de définir la grammaire du langage non pas directement sur le vocabulaire du langage, mais plutôt en supposant déjà définies des grammaires simples (en l'occurrence des grammaires régulières) pour les composants simples du langage appelés *unités lexicales* : les constantes numériques, les identificateurs, les mots-clés, etc. Chaque non-terminal source de l'une de ces grammaires régulières est appelé un *token*. Un token représente donc le langage régulier associé à une unité lexicale donnée.

3.4.2 Fichier de description LEX

En pratique, il est agréable d'utiliser des expressions rationnelles auxiliaires, et la définition de langages rationnels se présente donc sous la forme suivante dans le langage LEX :

1. Une suite de définitions de la forme $D_i = R_i$, où D_i est un nom, et R_i est une expression régulière sur l'alphabet $V \cup \{D_1, \ldots, D_{i-1}\}$.

2. Une suite de règles de la forme **Si** P_i **Alors** A_i, où P_i est une expression rationnelle sur l'alphabet $V \cup \{D_i\}_i$ appelée *filtre*, et A_i est un programme caractéristique du type d'unité lexicale reconnu par le filtre P_i.

Par exemple, pour le langage des expressions arithmétiques, on aura :

lettre = $a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
chiffre = $0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Si chiffre(chiffre)* **Alors** %traitement des constantes
Si lettre(lettre)* **Alors** %traitement des identificateurs
Si + **Alors** %traitement de l'addition
Si $\times$ **Alors** %traitement de la multiplication

Les traitements peuvent être en général n'importe quelle expression ou instruction du langage de programmation auquel l'outil LEX est associé. Nous l'utiliserons uniquement pour retourner des *tokens* (voir paragraphe suivant).

Les outils LEX autorisent en général d'autres constructions pour les expressions rationnelles, comme les intervalles de lettres, qui permettent d'abréger l'écriture. L'outil LEX le plus utilisé est l'original et produit du langage C ; mais il existe également pour d'autres langages, avec quelques variantes. Noter qu'en CAML, l'outil CAMLLEX ne permet pas les définitions.

3.4.3 Description des tokens

LEX va nous servir à reconnaitre les différentes unités lexicales sous forme de définitions auxiliaires et de règles. Nous pourrons par exemple avoir un fichier LEX de la forme :

lettre = $a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
chiffre = $0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Si *if* **Alors** If
Si *then* **Alors** Then
Si *else* **Alors** Else
Si chiffre(chiffre)* **Alors** *Id*
Si lettre(lettre)* **Alors** Cte

Notons que puisque toutes les constantes sont remplacées par le token Cte, on a perdu de l'information. Pour la suite de l'interprétation du langage considéré, il faut garder un lien entre le token et la valeur de la constante correspondante. Le même problème se pose pour les identificateurs.

Il y a plusieurs façons de conserver ce lien, suivant le type de langage utilisé. En programmation fonctionnelle, on peut considérer le token comme étant une fonction dont l'argument est l'unité lexicale qui l'a produit : c'est ce qui est fait dans CAMLLEX. En programmation impérative, il est d'usage de conserver ce lien dans une table, la *table des symboles*. Le token est alors une paire, formée du token lui-même d'une part, et d'un pointeur sur la table des symboles d'autre part. La table des symboles pourra bien sûr contenir un grand nombre d'informations pour chaque occurrence différente d'un même token.

Décider d'une solution ou d'une autre est une question d'implémentation. En ce qui nous concerne, nous supposerons que l'unité lexicale représentée par le token t est obtenue par un appel à une fonction val, donc est donnée par l'expression val(t).

3.4.4 Règles de priorité

Étant donné un programme à analyser, c'est-à-dire un mot w sur l'alphabet V du langage de programmation considéré, il y aura en général de multiples façons de découper w en unités lexicales. Un principe de sélection est donc ajouté qui garantit l'unicité du découpage :

(i) Le filtre choisi est celui qui reconnaît le plus grand préfixe du mot w.

(ii) Au cas où plusieurs filtres $P_{i_1}, \ldots, P_{i_q}$, avec $i_1 < \ldots < i_q$, reconnaissent le même (plus grand) préfixe u, le filtre de plus petit indice P_{i_1} est choisi.

Soit par exemple un fragment de langage contenant deux types de tokens, le mot clé « end », et les identificateurs. On aura :

lettre = $a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
Si *end* **Alors** End
Si lettre(lettre)* **Alors** Id

Les automates correspondants à ces filtres sont représentés sur la figure 3.6 sous la forme d'un automate avec transitions vides dont la partie supérieure traite le mot clé « end » et la partie inférieure traite les identificateurs. À chaque état acceptant peut être associé le token correspondant : End pour q_4 et Id pour q_6.

Soit « endormi » le mot à reconnaître. Les deux filtres reconnaissent le même préfixe, mais le deuxième reconnaît aussi un préfixe plus long, le second filtre est donc choisi et l'état final de l'exécution de l'automate doit donc être l'état q_6. Soit maintenant « end » le mot à

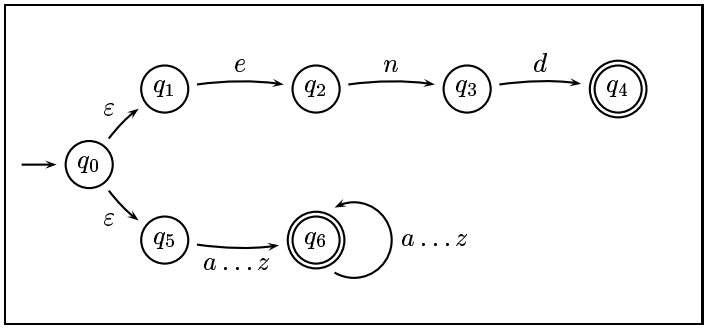


FIG. 3.6 – *Automate d'analyse lexicale*

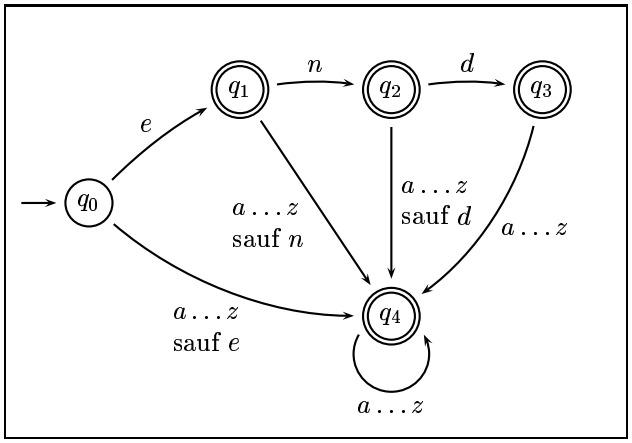


FIG. 3.7 – *Automate déterminisé d'analyse lexicale*

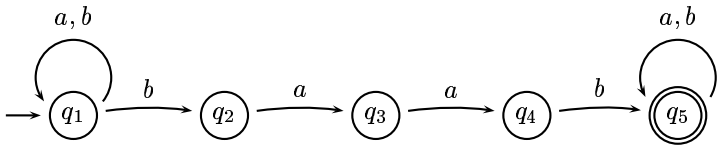
reconnaître. Les deux filtres reconnaissent le même préfixe de longueur maximale, « end ». Le premier filtre est donc choisi, et l'état final doit donc être l'état q_4.

Cet automate est déterminisé (et se trouve être minimal) sur la figure 3.7. L'état acceptant q_3 devrait normalement correspondre aux deux états acceptants de la figure 3.6. Le choix du premier filtre, c'est-à-dire du mot clé, fait que cet état acceptant est en fait associé au mot clé uniquement. Les tokens correspondants aux états acceptants sont donc End pour q_3 et Id pour q_1, q_2 et q_4.

En pratique, ces règles de priorité imposent que dans LEX, les filtres correspondants aux différents mots clés devront apparaître avant le filtre pour les identificateurs. De plus, on remarque qu'à chaque état acceptant est associé un unique token, décrit par un filtre P_i, et le code à exécuter lorsque cet état acceptant est atteint est donc le code A_i.

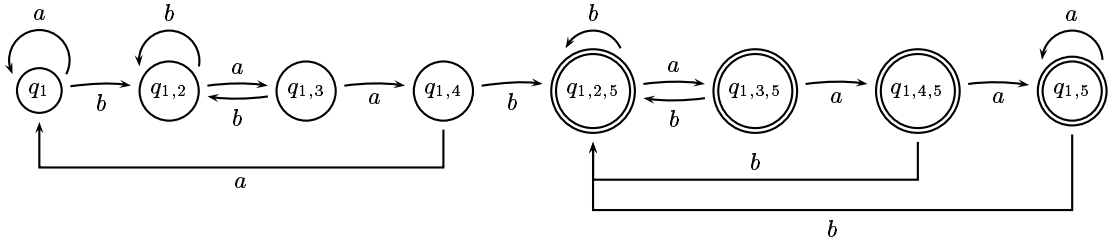
3.5 Exercices

Exercice 3.1
Soit l'automate :

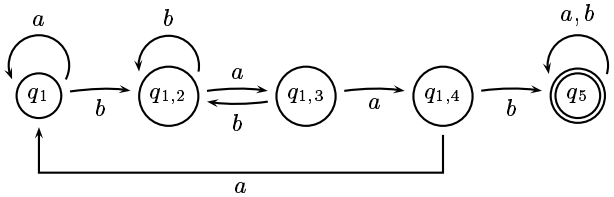


Quel est langage accepté par cet automate ? Le déterminiser et minimiser.
Correction : *Bien sûr, l'automate accepte le langage des mots qui contiennent baab comme sous-mot.*

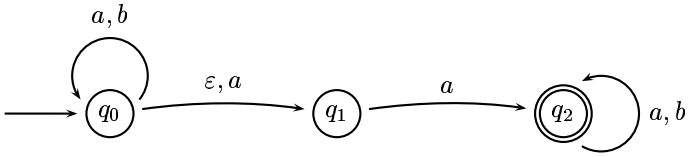
L'automate déterminisé correspondant est :



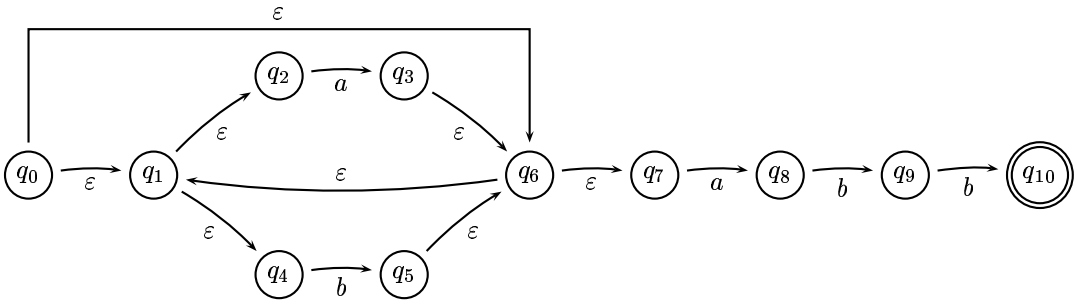
L'automate minimal est :



Exercice 3.2
Construire l'automate déterministe équivalent à l'automate suivant :



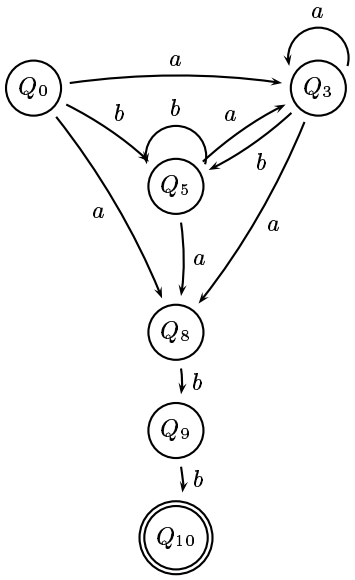
Exercice 3.3
Soit l'automate :



1. supprimer les transitions vides

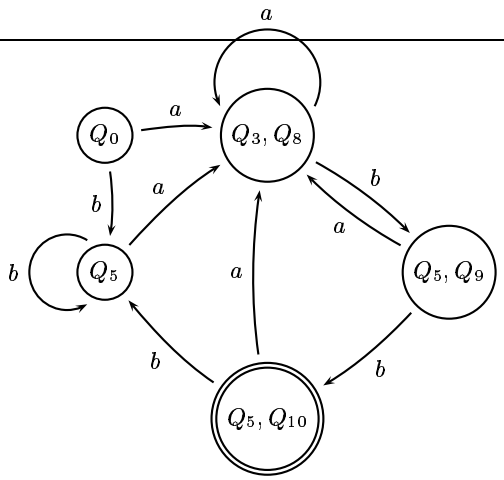
Correction :
On supprime d'abord les ε-transitions, pour cela on défini les clotures :
$cloture(q_0) = \{q_0, q_1, q_2, q_4, q_6, q_7\} = Q_0$
$cloture(q_1) = \{q_1, q_2, q_4\} = Q_1$
$cloture(q_3) = \{q_1, q_2, q_3, q_4, q_6, q_7\} = Q_3$
$cloture(q_5) = \{, q_1, q_2, q_4, q_5, q_6, q_7\} = Q_5$
$cloture(q_6) = \{q_1, q_2, q_4, q_6, q_7\} = Q_6$

et pour les autres etats, on a $cloture(q_i) = \{q_i\} = Q_i$
L'automate non déterministe sans ε-transition est :



2. Le déterminiser

Correction : *L'automate déterminisé est :*



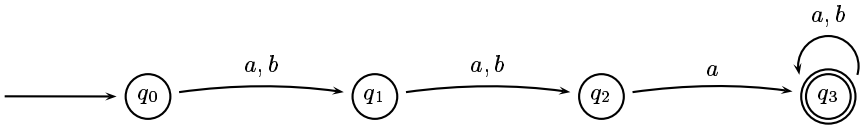
3. Est-il minimal ?

   Correction : *Non, les états Q_0 et Q_5 sont équivalent. L'automate minimal est l'automate sans Q_0, et avec Q_5 comme état initial*

Exercice 3.4

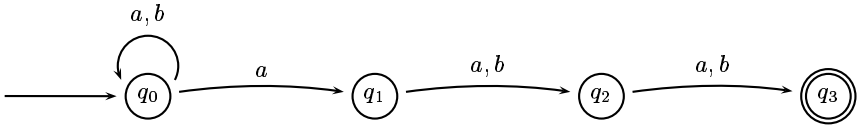
1. Donner un automate déterministe qui accepte le langage sur l'alphabet $\{a, b\}$ dont tous les mots admettent un a comme troisième lettre.

   Correction : *quatre états suffisent.*



2. Donner un automate non-déterministe qui accepte le langage sur l'alphabet $\{a, b\}$ dont tous les mots admettent un a comme troisième lettre *à partir de la droite.*

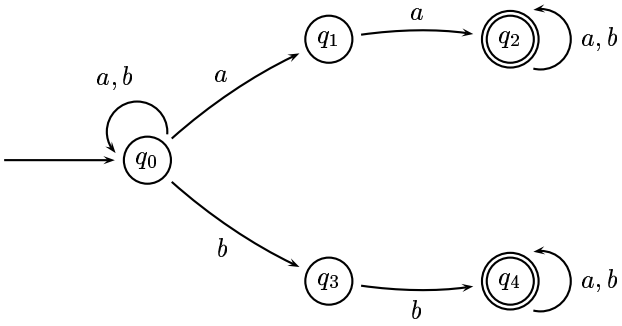
   Correction : *là encore, quatre états suffisent : c'est le miroir du précédent. Mais en faisant le miroir, cela devient non-déterministe.*



3. Donner un automate déterministe pour le même langage.

   Correction : *Il a huit états.*

Exercice 3.5
Considérons l'automate non-déterministe suivant :



1. Construire un automate déterministe équivalent.

   Correction : *L'etat initial du déterminisé est* $\{q_0\}$ *On calcule successivement*

   (a) $T(\{q_0\}, a) = \{q_0, q_1\}$

   (b) $T(\{q_0\}, b) = \{q_0, q_2\}$

   (c) $T(\{q_0, q_1\}, a) = \{q_0, q_1, q_2\}$

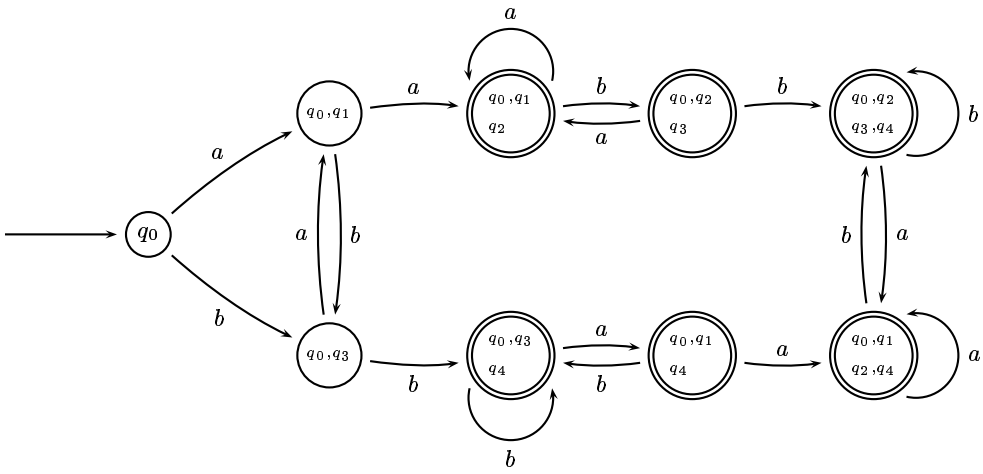
   (d) $T(\{q_0, q_1\}, b) = \{q_0, q_2\}$

   (e) $T(\{q_0, q_1, q_2\}, a) = \{q_0, q_1, q_2\}$

   (f) $T(\{q_0, q_1, q_2\}, b) = \{q_0, q_2, q_3\}$

   (g) *etc.*

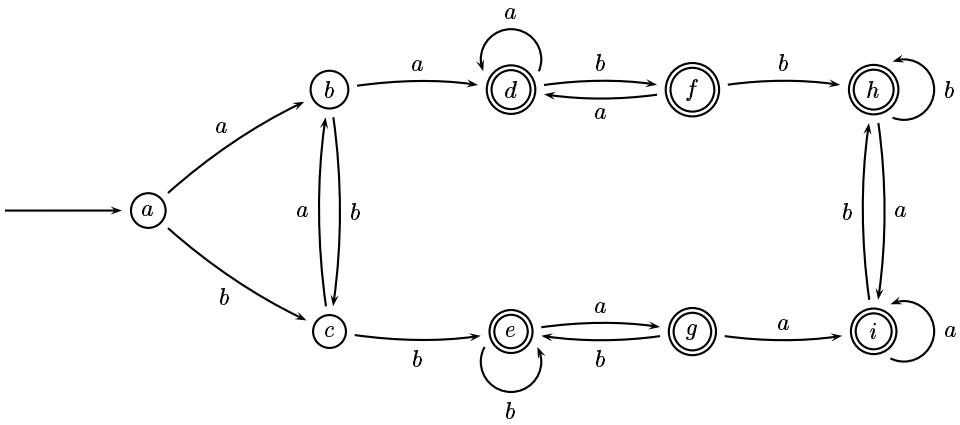
   Les états acceptants sont ceux qui contiennent au moins un des états acceptants de l'automate de départ. Ce qui donne



2. Minimiser l'automate obtenu.

   Correction :

On renomme les états d'abord pour éviter de se trimballer des noms à rallonge :



Il faut leur montrer comment on minimise à l'aide d'un tableau : on met une colonne par états, et sur la première ligne on écrire la partition de départ : classe numéro 0 pour les états non acceptants et classe numéro 1 pour les états acceptants. Ensuite, sur chaque ligne successivement :

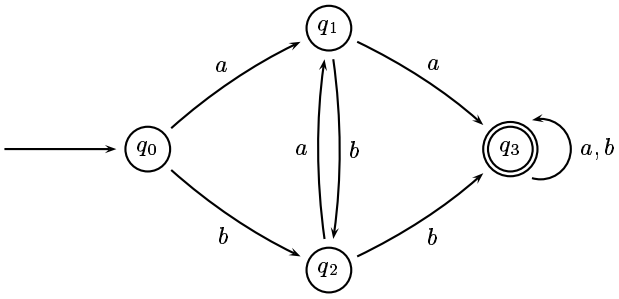
(a) on calcule la classe atteinte par lecture de a et de b ;

(b) on raffine la partition en créant un nouveau numéro de partition à chaque nouveau triplet (numéro de partition, transition par a, transition par b).

Sur l'exercice cela donne :

états	a	b	c	d	e	f	g	h	i
partition	0	0	0	1	1	1	1	1	1
classes de $T(q,a)$, $T(q,b)$	0, 0	1, 0	0, 1	1, 1	1, 1	1, 1	1, 1	1, 1	1, 1
partition	0	1	2	3	3	3	3	3	3
classes de $T(q,a)$, $T(q,b)$	1, 2	3, 2	1, 3	3, 3	3, 3	3, 3	3, 3	3, 3	3, 3
partition	0	1	2	3	3	3	3	3	3

et on s'arrête car c'est stable. D'où l'automate minimal :



Exercice 3.6
Soit L un langage reconnaissable, et L' le langage des mots obtenus à partir des mots de L en supprimant dans les mots de L les lettres de rang impair. Montrer que L' est reconnaissable.

Exercice 3.7
Au café des automates à Paris, quatre consommateurs ayant commandé quatre bières jouent avec un serveur aux yeux bandés et dont les mains sont munies de gants de boxe au jeu

suivant : ils commencent par retourner certains des quatre verres disposés en carré sur la table, puis, à chaque fois que le serveur a lui-même retourné soit un verre, soit deux verres, ils font tourner la table d'un nombre arbitraire de quart de tour. Le serveur étant les yeux bandés et muni de gants de boxe, ne peut savoir si les verres qu'il retourne étaient à l'endroit ou à l'envers : la seule information à sa disposition est que les verres qu'il retourne sont contigus ou opposés. Le jeu s'arrête (avec gain du serveur) s'il arrive à remettre tous les verres dans le même sens, soit à l'endroit, soit à l'envers. Montrer que le serveur possède une stratégie gagnante.

Exercice 3.8
Donner des expressions rationnelles dénotant les langages suivants :

1. les entiers en décimal ;

   Correction : *si l'on autorise les nombres commençant par 0, cela donne* $(0 \mid \cdots \mid 9)(0 \mid \cdots \mid 9)^*$ *ou si l'on est plus exigeant* $0 \mid (1 \mid \cdots \mid 9)(0 \mid \cdots \mid 9)^*$

2. les entiers relatifs en décimal, c'est-à-dire pouvant optionnellement commencer par un signe + ou un signe − ;

   Correction : $(+ \mid - \mid \varepsilon)(0 \mid (1 \mid \cdots \mid 9)(0 \mid \cdots \mid 9)^*)$

3. les identificateurs de PASCAL ;

   Correction : $(a \mid \cdots \mid z \mid A \mid \cdots \mid Z)(a \mid \cdots \mid z \mid A \mid \cdots \mid Z \mid 0 \mid \cdots \mid 9 \mid \_)^*$

4. les nombres réels en virgule flottante ;

   Correction : $(+ \mid - \mid \varepsilon)(0 \mid (1 \mid \cdots \mid 9)(0 \mid \cdots \mid 9)^*).(0 \mid \cdots \mid 9)^*(0 \mid \cdots \mid 9)$

5. les nombres réels en virgule flottante avec exposants.

   Correction : $(+ \mid - \mid \varepsilon)(0 \mid (1 \mid \cdots \mid 9)(0 \mid \cdots \mid 9)^*).(0 \mid \cdots \mid 9)^*(0 \mid \cdots \mid 9)(\varepsilon \mid (e \mid E)(+ \mid - \mid \varepsilon)(0 \mid (1 \mid \cdots \mid 9)(0 \mid \cdots \mid 9)^*)$

Exercice 3.9
On dit que deux expressions rationnelles r et s sont *équivalentes* si elles engendrent le même langage. Prouver les équivalences suivantes:

1. $r + s = s + r$
2. $(r + s) + t = r + (s + t)$
3. $(rs)t = r(st)$
4. $r(s + t) = rs + rt$
5. $(r + s)t = rt + st$
6. $\emptyset^* = \varepsilon$
7. $(r^*)^* = r^*$
8. $(\varepsilon + r)^* = r^*$
9. $(r^*s^*)^* = (r + s)^*$

Exercice 3.10
Est-ce que les expressions suivantes sont équivalentes pour n'importe quelles expressions rationnelles r et s:

1. $(rs + r)^*r$ et $r(sr + r)^*$

2. $s(rs+s)^*r$ et $rr^*s(rr^*s)^*$
3. $(r+s)^*$ et r^*+s^*

Exercice 3.11
Pour chaque cas de constructions des automates pour la concaténation, l'union et l'itération de langages, donner des exemples d'automates A_1 et A_2 qui la justifie, en ce sens que toute construction plus simple est incorrecte.

Exercice 3.12
1. Exprimer l'intersection de deux langages L_1 et L_2 à l'aide des opérations union et complémentaire de langages.
2. En déduire une méthode de construction de l'automate reconnaissant $L_1 \cap L_2$ en supposant connus les automates reconnaissant L_1 et L_2.
3. Donner une méthode directe de construction de l'automate reconnaissant $L_1 \cap L_2$ en supposant connus les automates reconnaissant L_1 et L_2, telle que l'automate obtenu soit déterministe si ces derniers le sont.

Exercice 3.13
Donner un automate reconnaissant le langage des mots formés d'un nombre pair de a et d'un nombre impair de b. En déduire une expression rationelle pour ce langage.

Exercice 3.14
Considérons le langage L des mots *non carrés* sur l'alphabet $\{a,b\}$, c'est-à-dire l'ensemble des mots w tels qu'il n'y ait pas de mot v tel que $w = vv$.

1. Donner une grammaire pour le langage des mots de longueur impaire de L.

   Correction : *On remarque que tous les mots de longueur impaire sont dans L.*

   $$A \rightarrow a \mid b \mid aaA \mid abA \mid baA \mid bbA$$

2. Montrer que si $w = v_1av_2bv_3$ avec $|v_2| = |v_1| + |v_3|$ alors $w \in L$.

   Correction : *Il faut montrer que w n'est pas un carré. Par l'absurde, si $w = vv$ alors $vv = v_1av_2bv_3$ mais comme $|v_2| = |v_1| + |v_3|$, on peut découper v_2 en v_4v_5 avec $|v_4| = |v_3|$ et $|v_5| = |v_1|$, et alors $|v_1av_4| = |v_5bv_3|$ donc $v = v_1av_4 = v_5bv_3$ mais cette dernière égalité est impossible car a et b sont à la même position.*

3. En exprimant l'ensemble des mots de la forme ci-dessus (en découpant v_2 de manière astucieuse), donner une grammaire engendrant les mots de longueur paire de L.

   Correction : *On remarque que l'on peut également découper v_2 en v_4v_5 avec cette fois $|v_4| = |v_1|$ et $|v_5| = |v_3|$. Autrement l'ensemble des mots de la forme ci-dessus est*

   $$\{v_1av_4v_5bv_3 \mid |v_4| = |v_1| \text{ et } |v_5| = |v_3|\}$$

   Ce langage est engendré par la grammaire

   $$\begin{aligned} B &\rightarrow B_1B_2 \\ B_1 &\rightarrow a \mid aB_1a \mid aB_1b \mid bB_1a \mid bB_1b \\ B_2 &\rightarrow b \mid aB_2a \mid aB_2b \mid bB_2a \mid bB_2b \end{aligned}$$

   On sait que tous les mots de ce langage sont dans L, réciproquement on remarque que les mots de L de longueur paire sont soit de la forme ci-dessus, soit de la forme obtenue en

échangeant a et b. Par conséquent, une grammaire engendrant les mots de longueur paire de L est

$$B \rightarrow B_1B_2 \mid B_2B_1$$

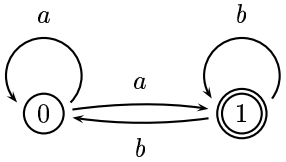
4. En déduire une grammaire pour L.

   Correction :

$$S \rightarrow A \mid B_1B_2 \mid B_2B_1$$

5. Soit L un langage reconnaissable. Montrer que le langage des mots non carrés de L défini par $L' = \{u \mid \text{il n'existe pas de } v \in L \text{ tel que } v = uu\}$ est également reconnaissable.

Exercice 3.15
Soit l'automate $\mathcal{A}$ dessiné ci-dessous:



On demande une expression rationelle permettant de décrire les langages suivants :

1. Le langage $L_{0,1}^{\{0,1\}}$ des mots reconnus par $\mathcal{A}$ en partant de l'état initial 0, en arrivant en état final 1, et en ne passant jamais par 0 ni 1 entre temps,

   Correction : a

2. Le langage $L_{1,0}^{\{0,1\}}$ des mots reconnus par $\mathcal{A}$ en partant de l'état 1 considéré comme état initial, en arrivant en état 0 considéré comme état final, et en ne passant jamais par 0 ni 1 entre temps,

   Correction : b

3. Le langage $L_{0,0}^{\{0,1\}}$ des mots reconnus par $\mathcal{A}$ en partant de l'état initial 0, en arrivant en état 0 considéré comme état final, et en ne passant jamais par 0 ni 1 entre temps,

   Correction : $\varepsilon + a$

4. Le langage $L_{1,1}^{\{0,1\}}$ des mots reconnus par $\mathcal{A}$ en partant de l'état 1 considéré comme état initial, en arrivant en état final 1, et en ne passant jamais par 0 ni 1 entre temps,

   Correction : $\varepsilon + b$

5. Le langage $L_{0,1}^{\{1\}}$ des mots reconnus par $\mathcal{A}$ en partant de l'état initial 0, en arrivant en état 1 considéré comme état final, et en ne passant jamais par 1 entre temps,

   Correction : $L_{0,1}^{\{0,1\}} \cup L_{0,0}^{\{0,1\}} L_{0,0}^{\{0,1\}^*} L_{0,1}^{\{0,1\}} = a + (\varepsilon + a)(\varepsilon + a)^* a = aa^*$

6. Le langage $L_{1,1}^{\{1\}}$ des mots reconnus par $\mathcal{A}$ en partant de l'état 1 considéré comme état initial, en arrivant en état final 1, et en ne passant jamais par 1 entre temps,

   Correction : $L_{1,1}^{\{0,1\}} \cup L_{1,0}^{\{0,1\}} L_{0,0}^{\{0,1\}^*} L_{0,1}^{\{0,1\}} = \varepsilon + b + b(\varepsilon + a)^* a = \varepsilon + ba^*$

7. En déduire une expression rationnelle décrivant le langage reconnu par l'automate $\mathcal{A}$

   Correction : $L_{0,1}^{\{1\}} L_{1,1}^{\{1\}^*} = aa^*(\varepsilon + ba^*)^* = a(a + b)^*$

8. Soit $\mathcal{A}$ un automate fini. On note par $L^I_{p,q}$, avec $I \subseteq Q$, $p \in I$ et $q \in I$, le langage des mots reconnus en allant de l'état p à l'état q dans l'automate $\mathcal{A}$ sans passer par un état intermédiaire appartenant à I. Exprimer $L^I_{p,q}$ en fonction $L^{I\cup\{r\}}_{p,q}$, $L^{I\cup\{r\}}_{p,r}$, $L^{I\cup\{r\}}_{r,r}$ et $L^{I\cup\{r\}}_{r,q}$, où r est un état quelconque n'appartenant pas à I.

   Correction : $L^I_{p,q} = L^{I\cup\{r\}}_{p,q} + L^{I\cup\{r\}}_{p,r} L^{I\cup\{r\}^*}_{r,r} L^{I\cup\{r\}}_{r,q}$

9. En déduire un algorithme prenant en entrée un automate $\mathcal{A}$, et rendant comme résultat une expression rationnelle dénotant le langage reconnu par $\mathcal{A}$.

   Correction : *On utilise la recurrence au-dessus. Il reste le cas $I = Q$:*

$$L^Q_{p,q} = \begin{cases} \{a \mid T(p,a) = q\} & p \neq q \\ \{a \mid T(p,a) = q\} \cup \{\varepsilon\} & p = q \end{cases}$$

   Ça donne:

$$L = \bigcup_{q \in F} \left(L^{\{q\}}_{q_0,q} L^{\{q\}^*}_{q,q} \right)$$

10. Appliquer l'algorithme à l'automate $\mathcal{A}$ de la première question.

Exercice 3.16

1. Donner une expression rationnelle dénotant le langage

$$L_0 = \{a^n b^p \mid \text{ où } n \text{ et } p \text{ sont des entiers naturels pairs(y compris 0)}\}$$

   Correction : $(aa)^*(bb)^*$

2. Donner un automate déterministe minimal reconnaissant le langage L_0.

   Correction : *4 états, 5 si complet.*

3. Donner une expression rationnelle (naïve) dénotant le langage $L = \bigcup_{k\geq 0} L_0^{2k+1}$

   Correction : $L = L_0.(L_0.L_0)^*$ *donc* $(aa)^*(bb)^*((aa)^*(bb)^*(aa)^*(bb)^*)^*$, *avec 34 symboles.*

4. Donner une expression rationnelle d'au plus 13 symboles qui dénote le langage L (il en existe en fait de moins de 10 symboles). Justifiez votre réponse.

   Correction : *On commence par constater que* $L_0^{2k} \subseteq L_0^{2k+1}$ *car* $\varepsilon \in L_0$, *donc* $L = \bigcup_{k>0} L_0^k$, *mais comme* $\varepsilon \in L_0$, $L = \bigcup_{k\geq 0} L_0^k$, *donc L est dénoté par* $((aa)^*(bb)^*)^*$ *avec 13 symboles. On voit facilement que cette expression est équivalente à* $(aa|bb)^*$, *avec 8 symboles.*

5. En déduire un automate déterministe minimal reconnaissant L.

   Correction : *Sans pb, avec 3 états, ou 4 si complet.*

6. Donner une expression rationnelle dénotant le langage L'_0 des mots obtenus à partir des mots de L_0 en supprimant, pour chacun d'eux, les lettres de rang impair (la première, la troisième, etc.), de telle sorte que le mot *aaaaaabbbbbbbb* devient *aaabbbb*).

   Correction : *C'est trivialement* a^*b^*.

7. Donner un automate déterministe minimal pour L'_0.

   Correction : *2 états, ou 3 si complet.*

8. Donner une expression rationnelle dénotant le langage L_0'' des mots obtenus à partir des mots de L_0 en supprimant, pour chacun d'eux, leur seconde moitié, de telle sorte que le mot *aaaaaabbbbbbbb* devient le mot *aaaaaab*.

   Correction : *Si l'on coupe un mot qui a plus de a que de b, on obtient une suite de a, et n'importe quelle suite de a peut être obtenue. Si l'on coupe un mot qui a plus de b que de a, on obtient un nombre pair de a suivi d'un nombre quelconque de b. L_0'' est donc dénoté par $a^*|(aa)^*b^*$.*

9. Donner un automate déterministe minimal pour L_0''.

   Correction : *3 états, 4 si complet.*

10. Exprimer à l'aide de L et L_0'' le langage L'' des mots obtenus à partir des mots de L en supprimant, pour chacun d'eux, leur seconde moitié, de telle sorte que le mot *aabbaaaabbaaaabbbbbbbb* devient *aabbaaaabba*.

    Correction : *$L = L_0^*$ donc les mots de L'' sont obtenus en découpant un mot de L, de la forme $w_1 \cdots w_n$ à un certain endroit, donc de la forme $w_1 \cdots w_k w$ où w est un mot de L_0 coupé à un endroit a priori quelconque, mais un tel mot est de toute façon dans L_0''. Donc $L'' = L_0^*.L_0'' = L.L_0''$.*

11. En déduire un automate déterministe minimal pour L''.

    Correction : *On construit un automate pour L'' en concatenant, avec une transition vide, les automates de L et de L_0''. La déterminisation et minimisation aboutit à un automate à 3 états, ou 4 si complet. Remarque : une expression rationnelle simple pour L'' est alors $(aa|bb)^*(a|b)$.*

Exercice 3.17

1. Donner un automate reconnaissant le langage dénoté par l'expression rationnelle ab^*.

   Correction : *Faire à l'intuition, pas avec la construction du cours.*

2. Donner un automate reconnaissant le langage dénoté par l'expression rationnelle $ab^*(a \mid b)a$.

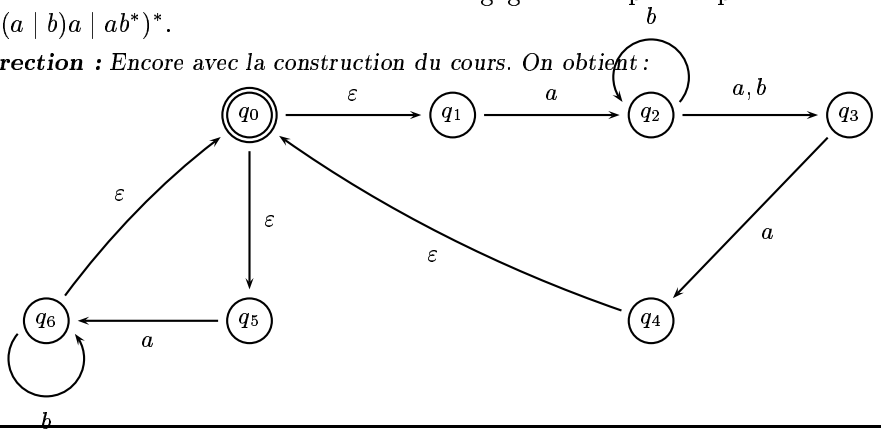
   Correction : *Idem*

3. En déduire un automate reconnaissant le langage dénoté par l'expression rationnelle $ab^*(a \mid b)a \mid ab^*$.

   Correction : *Appliquer la construction du cours. Faire remarquer qu'on ne peut pas se contenter de l'idée naïve consistant à identifier les états initiaux et les états acceptants, qui reconnaîtrait abab.*

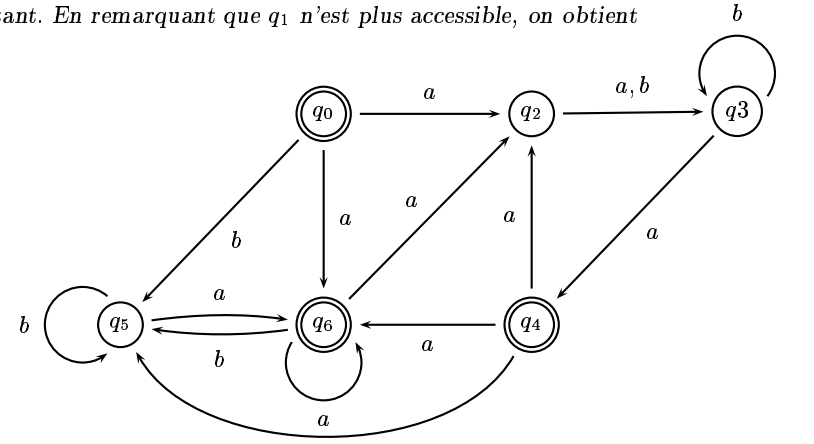
4. En déduire un automate reconnaissant le langage dénoté par l'expression rationnelle $(ab^*(a \mid b)a \mid ab^*)^*$.

   Correction : *Encore avec la construction du cours. On obtient :*



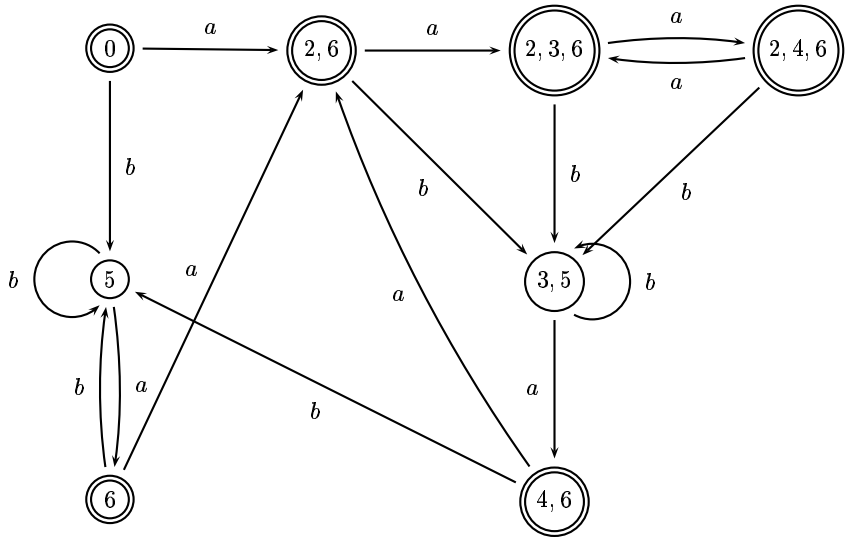
5. Supprimer les transitions vides de l'automate obtenu à la question précédente, puis supprimer les états inaccessibles.

   Correction : *On commence par calculer les différentes clôtures.* $cloture(q_0) = \{q_0, q_1, q_5\}, cloture(q_4) = \{q_0, q_1, q_4, q_5\}, cloture(q_6) = \{q_0, q_1, q_5, q_6\}$ *et pour les autres* $cloture(q) = \{q\}$ *Les nouveaux états acceptants sont ceux dont la clôture contient un état acceptant. En remarquant que* q_1 *n'est plus accessible, on obtient*



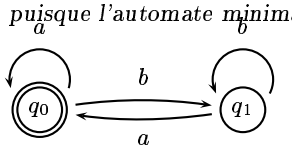
6. Déterminiser l'automate obtenu.

   Correction :



7. Le minimiser.

   Correction : *Là, c'est un plaisir puisque l'automate minimal est :*



8. Déduire de cet automate une nouvelle expression rationnelle dénotant le langage.

   Correction : *On fait ça intuitivement puisque la méthode générale est hors programme. Il faut chercher les différents cycles allant de* q_0 *à* q_0 *: il y a le cycle passant par* a *et celui effectuant* bb^*a*. On obtient donc* $(a \mid b.b^*a)^*$.

9. (facultatif) Vérifier par le calcul que ces expressions rationnelles dénotent le même langage. On procédera par inclusion réciproque, et on utilisera la propriété suivante : si $L_1 \subseteq L_2^*$ alors $L_1^* \subseteq L_2^*$.

   Correction : *On note* $L_1 = (L_1')^* = (a \mid b.b^*a)^*$ *et* $L_2 = (L_2')^* = (a(a \mid b)b^*a \mid b^*a)^*$

   - $a = b^0 a \in L_2'$
   - $L(bb^*a) \subset L(b^*a)$
     Donc $\underline{L_1 \subset L_2}$
   - *soit* $w \in L(b^*a)$, *en regardant sa première lettre, soit* $w = a$, *soit* $w \in L(bb^*a)$ *donc* $w \in L_2$
   - *soit* $w \in L(a(a \mid b)b^*a)$, *alors si* w *ne contient pas de* b *il est multiple de* a, *sinon, il est de la forme* $a^{1 \text{ ou } 2}bb^{j>0}a$ *et se décompose dans* L_1

Exercice 3.18
On veut montrer que le langage $L = \{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas régulier.

1. Soit $A = (V_t, Q, q_0, F, T)$ un automate déterministe complet quelconque. Trouver une borne $N \in \mathbb{N}$ associée à A qui assure que pour tout mot $w \in \mathcal{L}ang(A)$ de longueur supérieure ou égale à N, l'exécution de A sur w passe nécessairement au moins deux fois par un certain état.

   Correction : *Il suffit de prendre* $N = |Q|$. *En effet, pour tout mot* $w = \alpha_1 \cdots \alpha_n$ *de longueur* $n \geq N$, *l'exécution de* A *sur* w *a la forme*

   $$q_0 \to^{\alpha_1} q_1 \cdots \to^{\alpha_n} q_n$$

   la séquence $q_0, \ldots, q_n$ *possède* $n + 1 > |Q|$ *éléments, donc il n'est pas possible que tous les états ne soit parcouru que zéro ou une fois.*

2. Raisonnons par l'absurde en supposant que le langage L est reconnaissable par un automate déterministe complet A. Par un raisonnement analogue à la question précédente, trouver un entier N tel que lors de l'exécution de A sur le mot $a^N b^N$, un état est parcouru au moins deux fois lors de la lecture des lettres a.

   Correction : *Là encore, il suffit de prendre* $N = |Q|$. *En effet, l'exécution de* A *sur* $a^N b^N$ *a la forme*

   $$q_0 \to^a q_1 \cdots \to^a q_N \to^b q_1' \cdots \to^b q_N'$$

   la séquence $q_0, \ldots, q_N$ *possède* $N + 1 > |Q|$ *éléments, donc il n'est pas possible que tous les états ne soit parcouru que zéro ou une fois.*

3. En déduire qu'il existe un mot de la forme $a^{N+k}b^N$ avec $k > 0$ reconnu par l'automate, et conclure.

   Correction : *Dans la séquence précédente, nous savons qu'il existe* i *et* j *avec* $0 \leq i < j \leq N$ *tels que* $q_i = q_j$, *c'est-à-dire que l'exécution est de la forme*

   $$q_0 \to^{a^i} q_i \to^{a^{j-i}} q_j \to^{a^{N-j}} q_N \to^{b^N} q_N'$$

   Mézalor nous avons aussi l'exécution suivante qui est possible

   $$q_0 \to^{a^i} q_i \to^{a^{j-i}} q_j \to^{a^{j-i}} q_j \to^{a^{N-j}} q_N \to^{b^N} q_N'$$

   puisque $q_i = q_j$, *et donc le mot* $a^i a^{j-i} a^{j-i} a^{N-j} b^N = a^{N+j-i} b^N$ *est accepté par l'automate. alors qu'il n'est pas dans le langage* L, *ce qui est une contradiction.*

Chapitre 4

Analyse syntaxique

L'objectif de ce chapitre est d'introduire le lecteur aux techniques d'analyse syntaxique et à l'utilisation du logiciel YACC. L'analyse syntaxique est un processus techniquement plus complexe que l'analyse lexicale. Les objets mathématiques sous-jacents, appelés automates à pile, n'ont guère d'applications importantes en dehors de la compilation. Nous ne les étudierons donc pas en détail, nous contentant au contraire de dégager les principes essentiels de l'analyse syntaxique, de façon à comprendre grossièrement le fonctionnement du logiciel YACC.

4.1 Dérivations gauches et droites

Nous allons maintenant décrire le langage des expressions arithmétiques. Nous supposerons disposer d'un analyseur lexical reconnaissant le token *Id* décrivant le langage des (noms de) variables, et le token *Cte* décrivant le langage des constantes entières ou réelles (sans les exposants pour simplifier). Ces syntaxes seront supposées être celles des langages informatiques classiques, à l'exception près que les constantes et les variables pourront être de taille arbitraire. Pour simplifier nous considérons également que les seules opérations sont l'addition et la multiplication.

Une grammaire des expressions arithmétiques aura alors pour symboles terminaux $V_t = \{+, \times, (,), Id, Cte\}$ et pour non-terminaux l'unique symbole E qui sera donc la source :

$$E \to Id \mid Cte \mid E + E \mid E \times E \mid (E)$$

Cette grammaire permet d'engendrer l'expression arithmétique $x + 2.5 \times 4 + (y + z)$, en supposant que les mots 2.5 et 4 font partie du langage engendré par la grammaire des constantes, et que x, y, z font partie du langage engendré par la grammaire des variables. En effet :

$$\begin{aligned}&\underline{E} \to \underline{E} + E \to Id + \underline{E} \to Id + \underline{E} \times E \to Id + Cte \times \underline{E} \to Id + Cte \times \underline{E} + E \to \\ &Id + Cte \times Cte + \underline{E} \to Id + Cte \times Cte + (\underline{E}) \to Id + Cte \times Cte + (\underline{E} + E) \to \\ &Id + Cte \times Cte + (Id + \underline{E}) \to Id + Cte \times Cte + (Id + Id)\end{aligned}$$

et il suffit de remplacer de manière appropriée les différentes occurrences de *Id* et *Cte* pour obtenir le résultat désiré. La dérivation ci-dessus est appelée *dérivation gauche*, car le non-terminal réécrit à chaque étape est celui qui est situé le plus à gauche dans le mot obtenu à l'étape précédente. La dérivation ci-dessus n'est pas unique, en voici une autre très semblable, qui n'est pas une dérivation gauche, mais est obtenue en permutant les deux dernières étapes

de la dérivation gauche :

$$\underline{E} \to \underline{E} + E \to Id + \underline{E} \to Id + \underline{E} \times E \to Id + Cte \times \underline{E} \to Id + Cte \times \underline{E} + E \to$$
$$Id + Cte \times Cte + \underline{E} \to Id + Cte \times Cte + (\underline{E}) \to Id + Cte \times Cte + (E + \underline{E}) \to$$
$$Id + Cte \times Cte + (\underline{E} + Id) \to Id + Cte \times Cte + (Id + Id)$$

En voici encore une autre, une *dérivation droite* cette fois :

$$\underline{E} \to E + \underline{E} \to E + (\underline{E}) \to E + (E + \underline{E}) \to E + (\underline{E} + Id) \to \underline{E} + (Id + Id) \to$$
$$E \times \underline{E} + (Id + Id) \to \underline{E} \times Cte + (Id + Id) \to E + \underline{E} \times Cte + (Id + Id) \to$$
$$\underline{E} + Cte \times Cte + (Id + Id) \to Id + Cte \times Cte + (Id + Id)$$

Comment être sûr que ces dérivations sont équivalentes, en le sens que les mots engendrés ont la même signification, c'est-à-dire donneront lieu au même calcul ? Il se trouve que le calcul qui est associé à un mot w dépend non seulement du mot lui-même, mais aussi de la dérivation qui a servi à le fabriquer ! Cette dépendance s'exprime par un parenthésage du mot, qui peut être implicite dans le mot w, mais devient explicite dans la dérivation. Par exemple, le parenthésage explicite du mot $x + 2.5 \times 4 + (y + z)$ tel qu'il est défini par la première dérivation est $(x + (2.5 \times (4 + (y + z))))$. La seconde dérivation donne le même parenthésage, mais il n'en est pas de même pour la troisième, qui correspond au parenthésage $(((x + 2.5) \times 4) + (y + z))$. En fait le parenthésage attendu n'est ni l'un ni l'autre. Il doit traduire les conventions de *désambiguïsation* habituelles, à savoir que la multiplication a priorité sur l'addition, et que multiplication et addition associent à gauche, c'est-à-dire que $x + y + z$ doit être interprété comme étant $((x + y) + z)$. Cela ne semble pas très important puisque l'addition et la multiplication sont associatives, mais cela le devient lorsque la soustraction et la division font partie du langage (cf. exercice **??**). Le bon parenthésage est donc $((x + (2.5 \times 4)) + (y + z))$, et une dérivation de notre exemple conforme à ce parenthésage est la suivante :

On dit aussi *désambiguation* en mauvais français.

$$\underline{E} \to E + \underline{E} \to E + (\underline{E}) \to E + (E + \underline{E}) \to E + (\underline{E} + Id) \to \underline{E} + (Id + Id) \to$$
$$E + \underline{E} + (Id + Id) \to E + E \times \underline{E} + (Id + Id) \to E + \underline{E} \times Cte + (Id + Id) \to$$
$$\underline{E} + Cte \times Cte + (Id + Id) \to Id + Cte \times Cte + (Id + Id)$$

Cette dérivation est elle aussi une dérivation droite. Nous pouvons maintenant en donner une variante (à des permutations près) dans laquelle l'utilisation de la règle $E \to E \times E$ a lieu lorsqu'aucune autre règle ne peut être utilisée, ce qui est une manière de traduire la convention de désambiguïsation :

$$\underline{E} \to E + \underline{E} \to E + (\underline{E}) \to E + (E + \underline{E}) \to E + (\underline{E} + Id) \to \underline{E} + (Id + Id) \to$$
$$\underline{E} + E + (Id + Id) \to Id + \underline{E} + (Id + Id) \to Id + E \times \underline{E} + (Id + Id) \to$$
$$Id + \underline{E} \times Cte + (Id + Id) \to Id + Cte \times Cte + (Id + Id)$$

Notre mot possède donc plusieurs dérivations droites (et aussi plusieurs dérivations gauches) dans la grammaire, ce qui est la manifestation visible de l'ambiguïté. Nous posons donc la définition suivante.

Définition 4.1 *Une grammaire G est* ambiguë *s'il existe un mot $w \in \mathcal{L}ang(G)$ qui possède plusieurs dérivations droites dans G.*

Notre grammaire des expressions arithmétiques est donc ambiguë. Nous allons la transformer de manière à refléter les règles de désambiguïsation, ce qui nécessite l'utilisation de nouveaux non-terminaux F et G. L'idée de la transformation est la suivante : un nouveau symbole non-terminal est utilisé pour chaque niveau de priorité, et une règle récursive

gauche est utilisée pour traduire l'associativité à gauche. Par exemple, la règle $E \to E + F$ va permettre d'engendrer une suite d'additions parenthésées à gauche. Le non-terminal F servira à engendrer les expressions arithmétiques qui ne sont pas des additions. Il faudra donc rajouter aussi la règle $E \to F$ de manière à terminer la séquence d'additions. Ainsi, si l'on dispose également de la règle $F \to a$, on engendrera l'expression $a + a + a$ par la dérivation suivante:

$$\underline{E} \to \underline{E} + F \to \underline{E} + F + F \to \underline{F} + F + F \to a + \underline{F} + F \to a + a + \underline{F} \to a + a + a$$

la règle de multiplication doit être changée conformément à ce principe, et les règles de notre nouvelle grammaire des expressions arithmétiques sont alors les suivantes :

$$E \to E + F \mid F \qquad F \to F \times G \mid G \qquad G \to (E) \mid Cte \mid Id$$

Le même mot est maintenant obtenu par une unique dérivation gauche ou droite, nous donnons ci-dessous la dérivation gauche:

$$\begin{aligned}
&\underline{E} \to \underline{E} + F \to \underline{E} + F + F \to \underline{F} + F + F \to \underline{G} + F + F \to Id + \underline{F} + F \to \\
&Id + \underline{F} \times G + F \to Id + \underline{G} \times G + F \to Id + Cte \times \underline{G} + F \to Id + Cte \times Cte + \underline{F} \to \\
&Id + Cte \times Cte + \underline{G} \to Id + Cte \times Cte + (\underline{E}) \to Id + Cte \times Cte + (\underline{E} + F) \to \\
&Id + Cte \times Cte + (\underline{F} + F) \to Id + Cte \times Cte + (\underline{G} + F) \to Id + Cte \times Cte + (Id + \underline{F}) \to \\
&Id + Cte \times Cte + (Id + \underline{G}) \to Id + Cte \times Cte + (Id + Id)
\end{aligned}$$

La transformation opérée sur la grammaire a éliminé l'ambiguïté au profit d'une part d'une perte de simplicité, car l'écriture de la grammaire est plus complexe, et d'autre part d'une augmentation de la longueur de la dérivation des mots du langage. Nous reviendrons sur cette perte de simplicité au chapitre 5.

4.2 Arbres syntaxiques

Nous avons vu que, dans une même grammaire G, un mot pouvait avoir plusieurs dérivations équivalentes, c'est-à-dire s'obtenant les unes des autres par des permutations adéquates. Nous allons donc maintenant exhiber une structure de donnée, les *arbres syntaxiques*, qui est invariante par ces permutations, de telle sorte qu'un mot de $\mathcal{L}ang(G)$ possède un arbre syntaxique unique lorsque G est non-ambiguë.

Définition 4.2 *Étant donnée une grammaire $G = (V_t, V_n, S, R)$, les arbres de syntaxe de G sont des arbres, les nœuds internes étiquetés par des symboles de V_n, et les feuilles étiquetés par des symboles de V, tels que, si les fils pris de gauche à droite d'un nœud interne étiqueté par le non-terminal N sont étiquetés par les symboles respectifs $\alpha_1, \ldots, \alpha_n$, alors $N \to \alpha_1 \cdots \alpha_n$ est une règle de la grammaire G.*

Un arbre de syntaxe dont la racine est étiquetée par l'axiome de la grammaire et dont le mot des feuilles u appartient à V_t^* est appelé arbre de dérivation *de u.*

les arbres de dérivation sont également appelés *arbres syntaxiques.*

Tout arbre de dérivation est donc un arbre de syntaxe, mais certains arbres de syntaxe ne sont pas des arbres de dérivation, soit parce que leur racine n'est pas étiquetée par l'axiome de la grammaire, soit parce que certaines de leur feuilles sont étiquetées par des non-terminaux.

L'arbre de dérivation définit tout un ensemble de dérivations de son mot des feuilles. Une dérivation est en fait obtenue dès que l'on a fixé un mode de parcours de l'arbre dans lequel un nœud ne peut être visité que si son père l'a déjà été. Ainsi, la figure 4.1 montre deux arbres de dérivation du mot $Id + Cte \times Cte + (Id + Id)$ dans la grammaire ambiguë des expressions

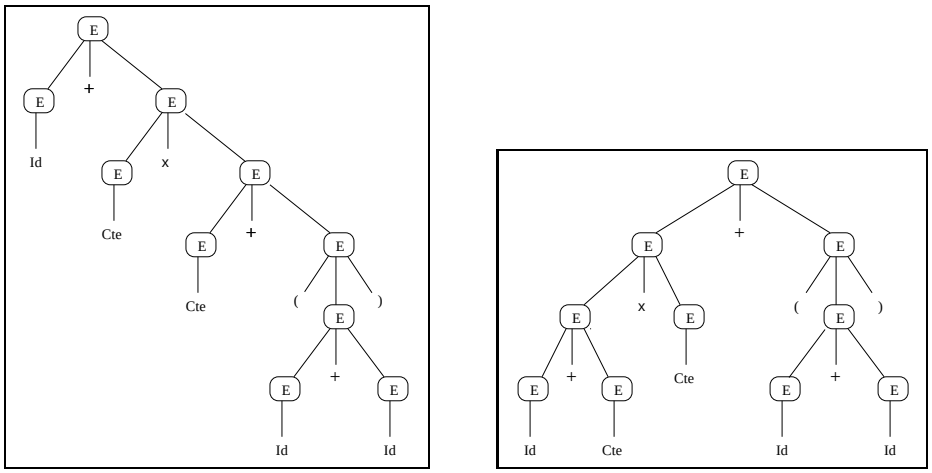


FIG. 4.1 – *Deux arbres de dérivation du mot* $Id + Cte \times Cte + (Id + Id)$ *dans la grammaire ambiguë des expressions arithmétiques*

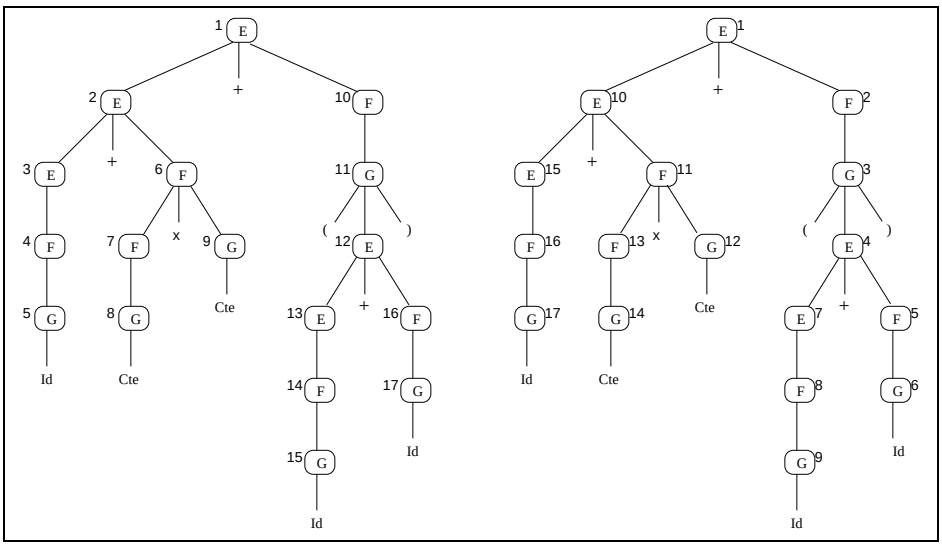


FIG. 4.2 – *Parcours gauche et droit de l'arbre de dérivation du mot* $Id + Cte \times Cte + (Id + Id)$ *dans la grammaire des expressions arithmétiques désambiguïsée*

arithmétiques. Partant de la racine, le parcours gauche consiste à descendre tant que cela est possible en choisissant en permanence la branche la plus à gauche. Lorsque l'on arrive à une feuille, il faut remonter au plus proche ancêtre dont un fils n'a pas encore été visité, puis recommencer la descente à partir du plus à gauche des fils non encore visités. La figure 4.2 indique les parcours gauches et droits de l'arbre de dérivation du mot $Id + Cte \times Cte + (Id + Id)$ dans la grammaire désambiguïsée, en numérotant les nœuds internes de l'arbre de dérivation dans l'ordre où ils sont visités.

La discussion précédente montre que le mot des feuilles d'un arbre de dérivation de la grammaire G appartient à $\mathcal{L}ang(G)$. Nous allons maintenant montrer la réciproque en construisant l'arbre de syntaxe associé à une dérivation. Notons que cette fois la dérivation est supposée quelconque : son origine est un non-terminal arbitraire, et le mot engendré peut

comporter des non-terminaux. On va donc construire des arbres de syntaxe qui ne seront pas nécessairement des arbres de dérivation. On procède par récurrence sur la longueur des dérivations de la façon suivante :

- à une dérivation de longueur nulle, c'est-à-dire à un symbole $\alpha \in V$, on associe un arbre de syntaxe réduit à une feuille étiquetée par α ;
- à une dérivation de longueur $n+1$, de la forme $N \to^* w_1 \underline{M} w_2 \to w_1\alpha_1 \dots \alpha_n w_2$, on associe l'arbre de syntaxe obtenu à partir de l'arbre de syntaxe de la dérivation $N \to^* w_1 M w_2$ en ajoutant, à la feuille étiquetée par M, n feuilles étiquetées respectivement par $\alpha_1, \dots, \alpha_n$.

On en déduit la propriété fondamentale suivante :

Théorème 4.3 *Étant donnée une grammaire $G = (V_t, V_n, S, R)$, un mot $u \in V_t^*$ appartient à $\mathcal{L}ang(G)$ si et seulement s'il possède un arbre de dérivation dans la grammaire G.*

On remarque facilement que des dérivations qui s'échangent par des permutations adéquates ont le même arbre de dérivation. On peut donc reformuler notre définition d'ambiguïté de la manière suivante :

Définition 4.4 *Une grammaire G est ambiguë s'il existe un mot $u \in \mathcal{L}ang(G)$ qui possède deux arbres de dérivation distincts dans la grammaire G.*

La figure 4.1 confirme que notre première grammaire des expressions arithmétiques était ambiguë.

4.3 Applications à l'analyse syntaxique

Il existe en fait deux façons bien différentes de concevoir l'analyse syntaxique d'un mot d'un langage décrit par une grammaire hors contexte. Lors de la lecture du mot (de la gauche vers la droite), l'arbre syntaxique peut être construit de haut en bas, ou de bas en haut, conformément à la figure 4.3. Nous allons maintenant illustrer ces deux méthodes sur des exemples.

4.3.1 Analyse syntaxique descendante

Considérons la grammaire de Dick :

$$S \quad \to \quad \varepsilon \mid (S)S$$

L'analyse descendante correspond à une dérivation gauche. Par exemple, pour le mot $(((())())())$, la dérivation gauche est la suivante:

$$S \to (S)S \to ((S)S)S \to (((S)S)S)S \to ((()S)S)S \to (((()S)S)S)S \to ((()()S)S)S$$

$$\to ((()())S)S \to ((()()))(S)S)S \to ((()())()S)S \to ((()())())S \to (((())())())$$

L'analyse syntaxique descendante va construire la suite des arbres d'analyse correspondant à cette dérivation, représentée à la figure 4.4.

Le choix entre les deux règles applicables peut se faire grâce à la connaissance du prochain caractère devant être engendré, appelé le *caractère d'avance*. Au départ, ce caractère est la parenthèse ouvrante, et la première règle à appliquer doit donc engendrer une parenthèse ouvrante. Il se trouve que la seconde règle engendre une parenthèse ouvrante, et ce sera donc elle que l'on appliquera chaque fois qu'une parenthèse ouvrante devra être engendrée. De

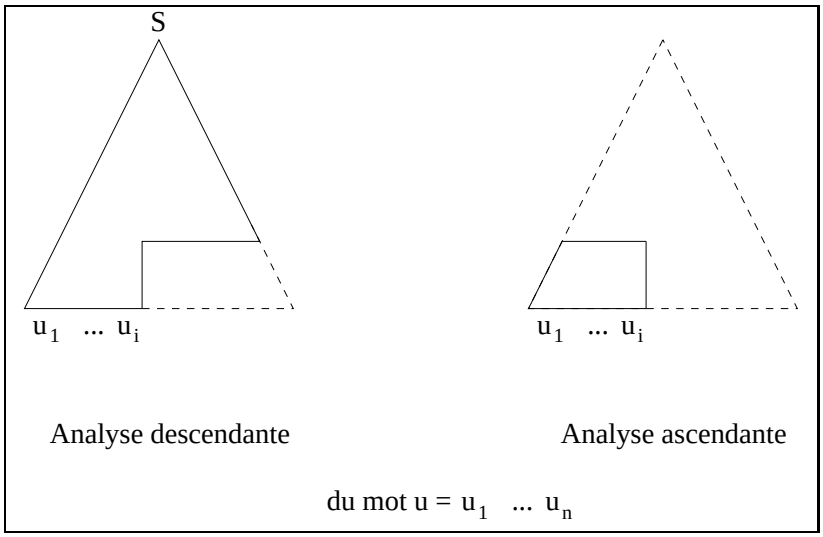


FIG. 4.3 – *Méthodes d'analyse syntaxique*

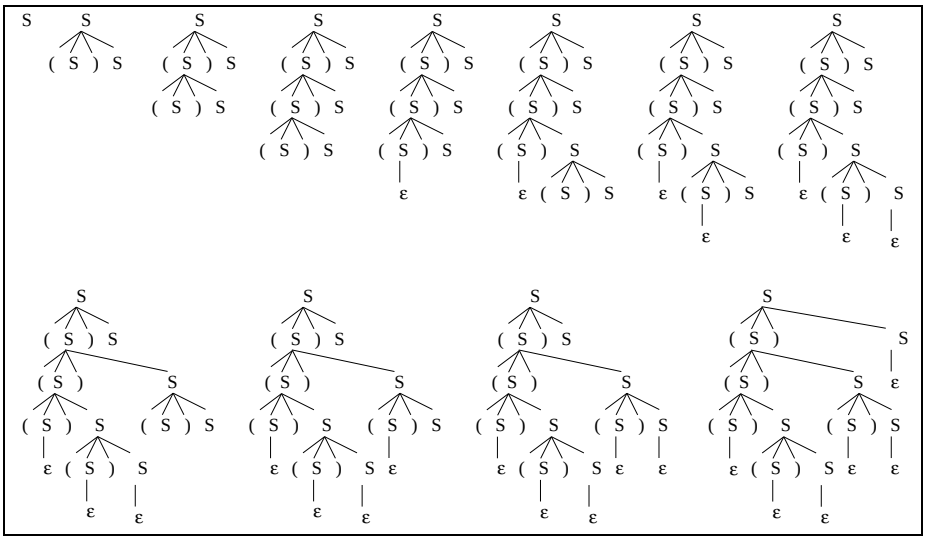


FIG. 4.4 – *Analyse syntaxique descendante*

son coté, la première règle fait disparaître le non-terminal S. Or, dans le membre droit de la seconde règle, la première occurrence du non-terminal S est placée devant une parenthèse fermante. L'application de la première règle à une telle occurrence de S engendre donc indirectement une parenthèse fermante. Il en est de même de la seconde occurrence de S (il faut pour s'en rendre compte regarder les arbres syntaxiques) sauf dans le cas où il s'agit de la dernière telle occurrence de S à remplacer avant de terminer (et alors il faut ne rien engendrer du tout, car le mot à lire est entièrement lu). Le choix de l'application d'une règle est donc entièrement déterminé par la connaissance du caractère d'avance. Une telle grammaire est dite LL(1), le premier L étant l'initiale du mot anglais Left, qui indique que la lecture du mot à analyser se fait de gauche à droite, le second L, initiale du même mot anglais, indiquant que l'on construit une dérivation gauche, et le 1 indiquant l'utilisation d'un unique caractère d'avance. Toutes les grammaires ne sont pas LL(1), certaines peuvent être LL(2), sans être LL(1), mais seul le cas LL(1) est utilisé en pratique. Se rendre compte qu'une grammaire est LL(1) demande le calcul des premiers caractères engendrés par l'utilisation de chaque règle. Par exemple, dans la grammaire de Dick, le premier caractère engendré par l'utilisation de la seconde règle est la parenthèse ouvrante. Ces calculs sont simples, mais nous nous contenterons de cette description intuitive. Notons enfin qu'une grammaire LL(1) est nécessairement non-ambiguë, puisque le choix d'une règle lors de l'analyse est entièrement déterminé par la connaissance du caractère d'avance.

4.3.2 Analyse syntaxique ascendante

L'analyse descendante est très simple à mettre en œuvre, puisqu'il s'agit tout simplement de construire une dérivation gauche. Malheureusement, elle échoue dans de très nombreux cas. Elle échoue en particulier chaque fois que la grammaire est « récursive gauche », c'est-à-dire qu'un membre droit de règle commence par le non-terminal qui en est le membre gauche. Considérons par exemple la grammaire désambiguïsée des expressions arithmétiques :

$$\begin{aligned} E &\rightarrow E+T \mid T \\ T &\rightarrow T \times F \mid F \\ F &\rightarrow (E) \mid Id \mid Cte \end{aligned}$$

Les règles $E \rightarrow E+T$ et $T \rightarrow T \times F$ sont récursives gauches. De plus, la technique précédente ne permettrait visiblement pas de choisir entre les règles $E \rightarrow E+T$ et $E \rightarrow T$. L'analyse ascendante que nous allons voir maintenant n'a pas cet inconvénient, mais elle est d'une mise en œuvre plus complexe. Le principe est de construire une dérivation droite à l'envers. La construction d'une dérivation droite à l'endroit nécessiterait en effet une lecture de droite à gauche du mot à analyser. La lecture de gauche à droite est donc responsable de cette construction à l'envers de la dérivation. Ce mode opératoire a pour conséquence que les objets manipulés au cours de l'analyse ne sont pas des arbres de syntaxe, mais des séquences d'arbres de syntaxe appelées *forêts*. Une forêt a pour *racine* la séquence des racines des arbres qui la composent, c'est-à-dire un mot sur l'alphabet $V_t \cup V_n$ de la grammaire. Par exemple,pour le mot $Id \times Id +$ Cte, la suite des forêts construites est représentée à la figure 4.5. La suite des racines de ces forêts est la suite de mots $Id, F, T, T\times, T \times Id, T \times F, E, E+, E + Id, E + F, E + T, E$.

Les opérations effectuées au cours de l'analyse ascendante, comme on peut le remarquer sur la figure 4.5, sont de deux sortes:

(i) la lecture d'un caractère, opération appelée *shift*.

(ii) l'enracinement d'une règle, opération appelée *reduce*. Cette opération consiste à écrire la forêt courante g sous la forme de la juxtaposition de deux forêts $f f'$, puis à construire une nouvelle forêt en juxtaposant la forêt f avec l'arbre de syntaxe de racine N, membre gauche d'une règle $N \rightarrow racine(f')$, et dont f' est la suite des sous-arbres.

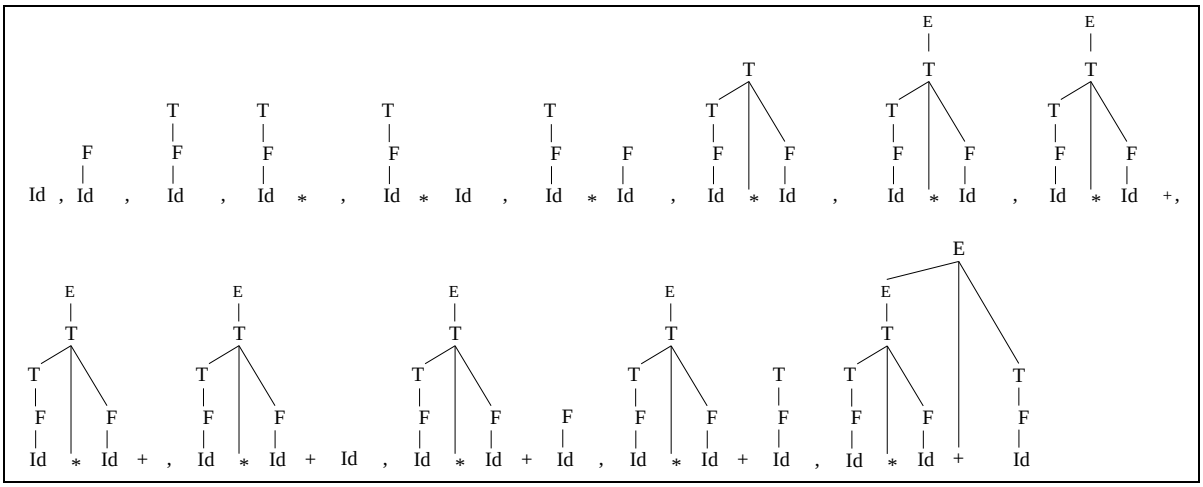


FIG. 4.5 – *Analyse syntaxique ascendante*

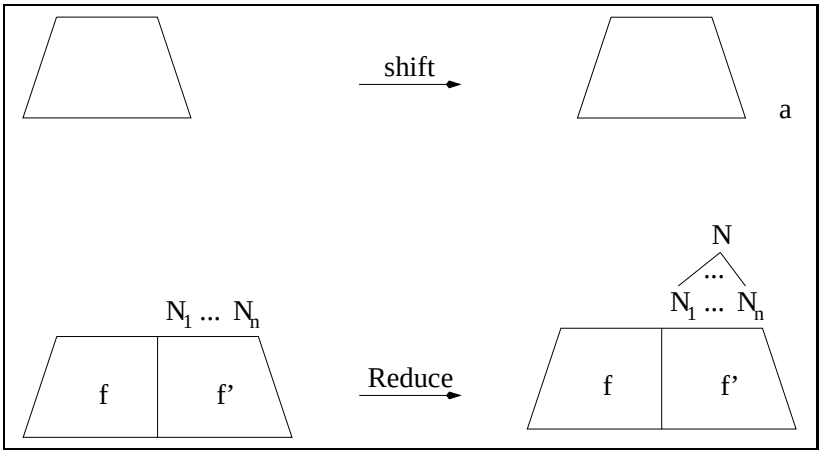


FIG. 4.6 – *Shift et Reduce*

Ces opérations sont décrites sur la figure 4.6.

Ces deux opérations sont bien sûr en général non-déterministes. On parle à ce propos de conflit shift-reduce, et de conflit reduce-reduce, car il se peut qu'il y ait plusieurs découpages $f_1 f_1'$ et $f_2 f_2'$ de f tels que les racines de f_1' et f_2' soient des membres droits de règles, de même qu'il peut y avoir plusieurs règles de même membre droit. Lorsqu'il est possible de résoudre le non-déterminisme par la lecture d'un caractère d'avance, on dit que la grammaire est LR(1). Le premier L a la même signification que précédemment, il correspond à la lecture de gauche à droite. Initiale du mot anglais Right, R correspond à la dérivation droite (faite à l'envers), et le 1 au nombre de caractères d'avance. Dans notre exemple, nous avons rencontré de nombreux problèmes de choix shift-reduce. Par exemple, après la lecture du premier token « *Id* » du mot à analyser, nous avons effectué une réduction alors que nous aurions pu tout aussi bien faire une nouvelle lecture. Dans ce cas, il n'aurait plus jamais été possible d'obtenir un arbre syntaxique dont le premier caractère lu soit « *Id* ». Par contre, lors de la troisième étape, c'est un shift qui a été préféré à un enracinement : il aurait été possible d'enraciner avec la règle $E \to T$, mais il n'aurait jamais été possible d'enraciner ultérieurement $E\times$,

qui n'apparaît pas dans un membre droit de règle. Des conflits shift-reduce se présentent également, à chaque fois que la racine de la forêt courante se termine par $E + T$ ou $T \times F$. On peut alors enraciner avec les règles $E \to T$ et $E \to E + T$ dans le premier cas, et avec les règles $T \to F$ et $T \to T \times F$ dans le second. Dans les deux cas, on préférera la règle de plus long membre droit, c'est à dire $E \to E + T$ ou $T \to T \times F$. De nouveau, le choix contraire interdirait l'enracinement ultérieur de $+E$ dans le premier cas et de $\times T$ dans le second. Comme il est possible de résoudre tous les conflits, la grammaire des expressions arithmétiques est LR(1). Notons là-encore qu'une grammaire LR(1) est non-ambiguë.

4.4 Génération d'analyseurs syntaxiques avec YACC

On dispose aujourd'hui d'outils qui permettent la génération d'analyseurs syntaxiques à partir de grammaires hors-contextes, dont le plus connu est YACC. Étant donnée une grammaire, YACC essaye de trouver une stratégie pour une analyse ascendante en utilisant la méthode décrite dans la section précédente. La classe de grammaires pour lesquelles YACC est capable d'engendrer un analyseur est une variante de LR(1) appellée LALR(1). En fait, YACC accepte comme données des grammaires qui peuvent être ambiguës (donc ne sont exactement pas LALR), pourvu qu'elles soient accompagnées des *règles de désambiguïsation* adéquates qui précisent les priorités respectives des différents opérateurs. Cette facilité permet de garder des grammaires simples, mais son utilisation nécessite une bonne compréhension du fonctionnement du logiciel. Nous nous interdirons pour cette raison cette possibilité de YACC : les grammaires que nous utiliserons devront toujours être non-ambiguës.

Dans la section précédente, nous avons décrit le principe de l'analyse ascendante comme une construction explicite de l'arbre de dérivation. Pour la réalisation d'un tel algorithme il n'est pas du tout nécessaire de garder toutes les forêts construites : comme toutes les actions de l'analyseur dépendent seulement du prochain caractère lu et des *racines* des forêts construites, YACC va économiser l'utilisation de la mémoire et stocker seulement les racines des forêts. Autrement dit, l'analyseur engendré par YACC par défaut fournit seulement une fonction disant si un mot donné est dans le langage engendré par la grammaire ou pas. Cependant, l'analyse syntaxique n'est en principe qu'un premier pas dans un logiciel plus complexe, et on a donc généralement besoin d'obtenir un résultat plus informatif. Dans ce but, YACC permet d'associer aux règles de la grammaire des expressions permettant de construire le résultat souhaité de l'analyse syntaxique, arbre de syntaxe, résultat d'évaluation, etc. Ces expressions prennent des valeurs dans une domaine défini dans le langage de programmation, et utilisent des opération prédéfinies (ou définies par le programmeur) dans ce langage. L'exemple ci-dessous décrit une grammaire des expressions arithmétiques, ainsi que l'évaluation des expressions reconnues (notre syntaxe dérive de la syntaxe réelle de YACC, à trouver dans un manuel de référence) :

Non-terminaux : E, T
Axiome : E
Tokens : Cte avec valeur de type entier, +, (,)

$$\begin{array}{lcll} E & \to & E \ + \ T & \{E + T\} \\ & | & T & \{T\} \\ T & \to & (E) & \{E\} \\ & | & \text{Cte} & \{\text{val(Cte)}\} \end{array}$$

On fournit donc à YACC les informations concernant la grammaire d'une part, c'est-à-dire la liste des non-terminaux, l'axiome, la liste des tokens, celle des règles, et d'autre part un mécanisme d'évaluation donné sous la forme d'expressions accollées aux règles. L'expression $\{E + T\}$ accollée à la règle $E \to E + T$ spécifie un calcul à effectuer sur l'arbre

de syntaxe. La règle $E \to E + T$ signifie que l'arbre de syntaxe de racine E est formé d'un arbre de racine E (le E du membre gauche de règle), dont les trois sous-arbres ont pour racines respectives E (le E du membre droit de règle), $+$ et T. Notons par A_1 et A_2 les deux sous-arbres de syntaxe dont les racines E et T sont des non-terminaux. Supposons (c'est notre hypothèse de récurrence) calculées les valeurs v_1 et v_2 des sous-arbres A_1 et A_2. Le calcul de la valeur de l'arbre tout entier s'exprime comme une fonction, qui ne dépend que de la règle $E \to E + T$, dont les arguments sont v_1 et v_2. Dans le cas présent, ce sera la fonction qui calcule la somme $v_1 + v_2$. Quand il y a plusieurs occurrences du même token ou non-terminal dans le membre droit d'une règle de la grammaire, on doit les numéroter pour les distinguer, comme dans l'exemple suivant, qui calcule le nombre de paires de parenthèses dans un mot de la grammaire de Dick à une parenthèse :

Non-terminaux : S
Axiome: S
Tokens: (,)

$$\begin{array}{rcll} S & \to & \varepsilon & \{0\} \\ & | & (S)S & \{S_1 + S_2 + 1\} \end{array}$$

Ce mécanisme est utilisé de manière systématique pour calculer une représentation compacte de l'arbre de dérivation, comme expliqué dans le chapitre 5.

4.5 Exercices

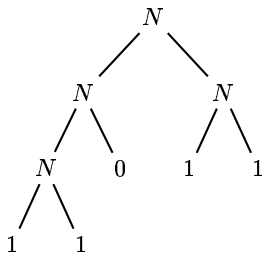
Exercice 4.1
On considère la grammaire suivante :

$$G = (\{0, 1\}, \{N\}, N, \{N \to 11 \mid 1001 \mid N0 \mid NN\})$$

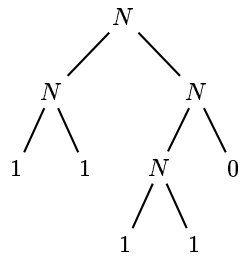
1. Dessiner les arbres de dérivation pour les mots 11011 et 11110.

   Correction :

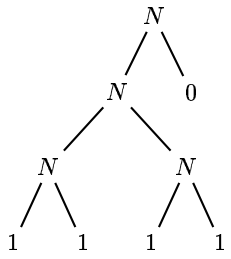
   - $\underline{N} \to \underline{N}N \to \underline{N}0N \to 110\underline{N} \to 11011$.



   - $\underline{N} \to \underline{N}N \to 11\underline{N} \to 11\underline{N}0 \to 11110$



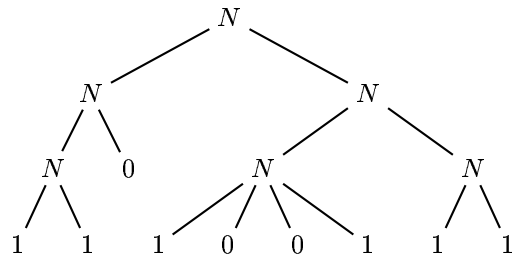
ou $\underline{N} \to \underline{N}0 \to \underline{N}N0 \to 11\underline{N}0 \to 11110$



2. Est-ce que la grammaire est ambiguë?

   Correction : *Oui. Voir les dérivations du second mot ci-dessus.*

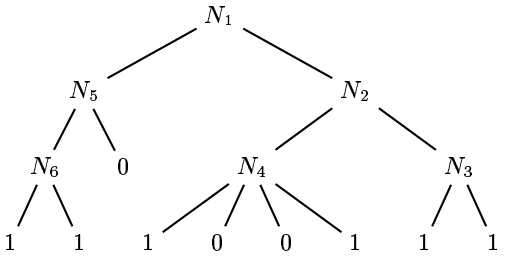
3. On regarde l'arbre syntaxique suivant:



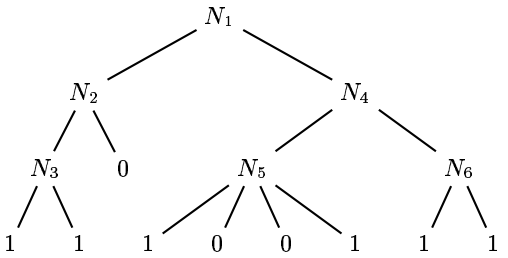
   Donner la dérivation gauche et la dérivation droite pour cet arbre.

   Correction :

   - *Doite:* $\underline{N} \to N\underline{N} \to NN\underline{N} \to N\underline{N}11 \to \underline{N}100111 \to \underline{N}0100111 \to 110100111$.



   - *Gauche:* $\underline{N} \to \underline{N}N \to \underline{N}0N \to 110\underline{N} \to 110\underline{N}N \to 1101001\underline{N} \to 110100111$.



Exercice 4.2
La grammaire G de l'exercice 1.15, est-elle ambiguëe?
Correction : *Oui, on considère le mot aabbab.*

Exercice 4.3
Trouver une grammaire pour le langage suivant :

$$\{a^i b^j c^k \mid i,j,k \geq 0, i = j \text{ ou } j = k\}$$

Votre grammaire est-elle ambiguë? Pourquoi?
Correction :

$$\begin{array}{rcl} S & \rightarrow & XY \mid WZ \\ X & \rightarrow & aXb \mid \varepsilon \\ Y & \rightarrow & cY \mid \varepsilon \\ W & \rightarrow & aW \mid \varepsilon \\ Z & \rightarrow & bZc \mid \varepsilon \end{array}$$

La grammaire est nécessairement ambiguë car dans le cas où le nombre de a, de b et de c sont égaux, il y a ambiguïté dans la définition : soit le mot est dans le langage parce que $i = j$ soit parce que $j = k$. On peux s'en convaincre avec abc qui possède deux dérivations gauches distinctes :

$$\begin{array}{rcl} S & \rightarrow & XY \rightarrow aXbY \rightarrow abY \rightarrow abc \\ S & \rightarrow & WZ \rightarrow aZ \rightarrow abZc \rightarrow abc \end{array}$$

Exercice 4.4
On considère le langage des expressions arithmétiques avec addition, opérateur infixe noté $+$, soustraction, opérateur infixe noté $-$, opposé, opérateur unaire préfixe noté $-$, multiplication, opérateur binaire noté $*$, division, opérateur binaire infixe noté $/$, et exponentiation, opérateur binaire noté $\uparrow$. Les opérateurs binaires associent tous à gauche sauf l'exponentiation qui associe à droite, et l'ordre de priorité est $\uparrow > -$(opposé) $> \{*, /\} > \{+, -$(soustraction)$\}$.

1. Donner le bon parenthésage des mots $3 - 2 - 1$ et $2 \uparrow 3 \uparrow 4$.

   Correction : *Pour le premier mot, il faut appliquer la règle d'associativité gauche de la soustraction. La bonne lectture est donc $(3 - 2) - 1$. Pour lire le second mot il fait utiliser l'associativité droite de $\uparrow$. Cela donne $2 \uparrow (3 \uparrow 4)$.*

2. Donner le bon parenthésage du mot $x + -y \uparrow 2 - 3 * x$.

   Correction : *L'opérateur le plus prioritaire est $\uparrow$. La bonne lecture est donc $(x + (-(y \uparrow 2))) - (3 * x)$.*

3. Donner une grammaire non-ambiguë décrivant ce langage.

   Correction : *L'associativité à gauche d'un opérateur op peut être codée dans un grammaire à l'aide de règles de production de la forme $E \rightarrow E$ op E' où le E' est soit un terminal soit un symbole non terminal pour la production des expressions ayant un opérateur principal de plus grande priorité que op.*

   La priorité à droite se code avec des règles de la forme $E \rightarrow E'$ op E où le E' est soit un terminal soit un symbole non terminal pour la production des expressions ayant un opérateur principal de plus grande priorité que op.

   Pour pouvoir engendrer toutes les expressions il ne faut pas oublier qu'une expression n'a pas forcément comme symbole principal l'opérateur de plus faible priorité.

   $E \rightarrow E + E_1 | E - E_1 | E_1$
   $E_1 \rightarrow E_1 * E_2 | E_1 / E_2 | E_2$
   $E_2 \rightarrow -E_2 | E_3$
   $E_3 \rightarrow int \uparrow E_3 | int$

Exercice 4.5

1. Spécifier un type abstrait « arbre de syntaxe », avec ses fonctions de construction et d'accès.

   Correction :

```
type abstrait << ads >> (pour <<arbre de syntaxe >> ):

  constructeurs:

     noeud : lettre x (sequence d'ads) -> ads
   { noeud(a,s) construit l'arbre de syntaxe dont la racine est la
lettre a et dont les sous-arbres sont les arbres de la sequence s }

  acces:

     racine : ads -> lettre
   { racine(a) retourne la racine de l'arbre a }

     sous_arbres : ads -> sequence d'ads
   { sous_arbres(a) retourne la sequence des sous-arbres de a }

le type sequence est suppose connu, defini par :

type abstrait << sequence de <element> >>:

  constantes:

    seq_vide
   { seq_vide représente la sequence vide }

  constructeurs:

    cons_seq : element x sequence -> sequence
   { cons_seq(x,s) retourne la sequence formee de x suivi des elements
de s }

  tests:

    seq_est_vide : sequence -> booleen
   { seq_est_vide(s) retourne vrai si la sequence est vide }

  acces:
    debut : sequence -> element
   { debut(s) retourne le premier element de la sequence s (erreur si
s est vide }

    reste : sequence -> sequence
   { reste(s) retourne la sequence formee de s exceptee le premier
element (erreur si s est vide }
```

2. Spécifier puis réaliser une fonction de parcours gauche et une fonction de parcours droit.

   Correction :

```
fonction parcours_gauche:

specifications:

  parcours_gauche : ads -> sequence de lettres
  { parcours_gauche(a) retourne ... }

  parcours_gauche_sequence : sequence d'ads -> sequence de lettres
  { parcours_gauche_sequence(s) retourne ... }

realisations :

parcours_gauche(a):
  cons_seq(racine(a),parcours_gauche_sequence(sous_arbres(a))

parcours_gauche_sequence(s):
 si seq_est_vide(s)
    alors seq_vide
    sinon concat(parcours_gauche(debut(s)),parcours_gauche_sequence(reste(s)))
```

3. Coder ce type abstrait et ces fonctions de parcours en CAML puis en PASCAL.

Exercice 4.6

1. Construire l'arbre de dérivation du mot $Id + \text{Cte} \times \text{Cte} + (Id + Id)$ dans la grammaire ambiguë des expressions arithmétiques, correspondant au parenthésage $((x + (2.5 \times 4)) + (y + z))$.

2. Construire l'arbre de dérivation correspondant à la dérivation gauche du mot $Id + \text{Cte} \times \text{Cte} + (Id + Id)$ dans la grammaire désambiguïsée des expressions arithmétiques.

Exercice 4.7

Considérons le langage L_C des expressions conditionnelles, où l'on peut construire des conditionnelles avec partie *else* (dites closes) ou sans partie *else* (dites non closes). Par exemple, les expressions suivantes sont dans L_C (c représente une condition quelconque, et i représente une instruction quelconque):

`if` c `then` i
`if` c `then` i `else` i
`if` c `then` i `else if` c `then` i `else` i

1. Trouvez une grammaire pour L_C.

   Correction : *Deux solutions immédiates et raisonnables sont:* $I \rightarrow$ `if` C `then` I `else` I| `if` C `then` I *et* $I \rightarrow$ `if` C `then` I Opt, $Opt \rightarrow \varepsilon$|`else` I.

2. Donnez un arbre de dérivation pour la troisième expression. Votre grammaire est-elle ambiguë?

3. Trouvez une grammaire non-ambiguë pour L_C. On souhaite implémenter la convention courante qui veut que le `else` soit rattaché au `if then` le plus proche. (Considérez deux types de règle de production, une pour les instructions closes, l'autre pour les instructions non closes.)

   Correction : *I_{clos} (resp. $I_{nonclos}$) engendre tous les expressions où le dernier condtionel est clos (non clos).*

$$\begin{array}{lcl} I & \rightarrow & I_{clos} | I_{non\ clos} \\ I_{clos} & \rightarrow & \texttt{if}\ C\ \texttt{then}\ I_{clos}\ \texttt{else}\ I_{clos} \\ I_{non\ clos} & \rightarrow & \texttt{if}\ C\ \texttt{then}\ I_{clos}\ \texttt{else}\ I_{non\ clos} \\ & | & \texttt{if}\ C\ \texttt{then}\ I \end{array}$$

4. Est-ce que vous conseillez la première grammaire pour la définition d'un langage de programmation ou bien la deuxième ? Quels changements dans la syntaxe proposez-vous pour ne pas avoir à choisir.

   Correction : *Solutions proposées: Ajouter des parenthèses (lisp) ou begin end (?), ajouter un* `fi` *(sh) un* `endif` *(csh, cpp) à la fin de la construction, faire la partie* else (peut-être avec une instruction vide) obligatoire (ML sur sa grammaire abstraite ne considère que les conditionnelles complète. Une absence de `else` est prise comme un `else` () implicite).

Exercice 4.8
Le langage $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ est-il engendré par une grammaire hors-contexte ? On pourra s'inspirer de l'exercice 2.3.

Exercice 4.9
Un palindrome non trivial est un mot de longueur au moins deux sur l'alphabet $a - z$ qui se lit aussi bien de droite à gauche que de gauche à droite. Par exemple, *eve*, *anna*, *otto* sont des palindromes. Une suite-palindrome sera un mot sur ce même alphabet obtenu en concaténant plusieurs palindrômes, comme par exemple *annaeveotto*.

1. Donner une grammaire pour les suites-palindrome.

   Correction :

   $$G = (\{a, \ldots, z\}, \{S\}, S, \{S\}, \{S \to aSa \mid \ldots \mid zSz \mid S \to aTa \mid \ldots \mid zTz \mid SS \mid \varepsilon, T \to a \mid \ldots \mid z$$

2. Dessiner les arbres de dérivation des mots *annaeveotto* et *aaaa*

   Correction : *un arbre pour annaeveotto et deux pour aaaa.*

3. Est-ce que la grammaire est ambiguë ?

   Correction : *Oui. Voir les dérivations du second mot ci-dessus.*

4. Peut-on trouver une grammaire non ambiguë pour le langage ?

   Correction : *Non,car le même mot peut avoir plusieurs découpages inclus les uns dans les autres, ou au contraire se chevauchant, et le choix d'un découpage minimal ou maximal ne peut se faire par une grammaire contect-free (je pense).*

Exercice 4.10
On souhaite engendrer tous les mouvements d'une machine à découper le tissus. Pour faciliter le travail, nous allons supposer que la pièce de tissus a une longueur et une hauteur infinie. Initialement la machine se trouve à l'angle inférieur gauche de la pièce de tissus.

1. À quelle découpe correspond le mot $\uparrow\uparrow\rightarrow\rightarrow\downarrow\downarrow$?

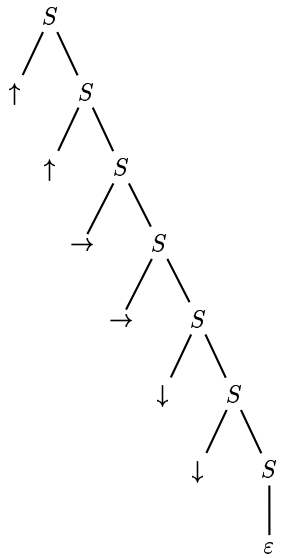
   Correction : *C'est un carré.*

2. Trouver une grammaire qui permet d'engendrer toutes les découpes.

   Correction : $S \to \downarrow S \mid \uparrow S \mid \leftarrow S \mid \rightarrow S \mid \varepsilon$.

3. Donnez un arbre de dérivation de l'exemple pour votre grammaire. Votre grammaire est-elle ambiguë ?

Correction :



Non cette grammaire n'est pas ambiguë : une seule règle s'applique pour chaque début de mot.

4. Donnez une grammaire qui ne permet de découper que des rectangles.

   Correction : $S \to \uparrow S \downarrow | U$
   $U \to \varepsilon | \to U$

5. Donnez une grammaire qui permet de ne découper que des carrés.

   Correction : *On veut donc des mots de la forme* $\uparrow^n \to^n \downarrow^n$. *ce n'est pas possible, les grammaires non contextuelles ne permettent pas de tels mots. Pour la correction de cette question, il va falloir réfléchir et trouver un contre-exemple à toutes leurs propositions.*

6. On suppose que le mouvement $\leftarrow$ n'est plus possible. Donner une grammaire non-ambiguë qui engendre toutes les découpes possibles sans sortir de la pièce de tissu.

   Correction : $S \to S \uparrow S \downarrow S | \varepsilon | \to$

7. Si maintenant le mouvement $\leftarrow$ est de nouveau possible, pouvez-vous donner une grammaire qui engendre toutes les découpes qui ne sortent pas de la pièce de tissu ?

   Correction : *Non.*

Exercice 4.11
On considère le langage des expressions arithmétiques avec addition, soustraction, opposé, multiplication, division, et exponentiation (notée $\uparrow$) :

$$\begin{array}{rcl} E & \to & E + E \mid E - E \mid -E \mid E \times E \mid E/E \mid E \uparrow E \mid (E) \mid Int \\ Int & \to & [0-9]^+ \end{array}$$

Cette grammaire étant ambiguë, on adopte les règles de désambiguïsation suivantes : les opérateurs binaires associent à gauche sauf l'exponentiation qui associe à droite, et l'ordre de priorité décroissant est l'exponentiation, puis l'opposé, puis la multiplication et la division, puis l'addition et la soustraction.

1. Donner le parenthésage implicite correct du mot $1 - 3 \times 4 + -5 \uparrow 2$.

   Correction : $(1 - (3 \times 4)) + (-(5 \uparrow 2))$.

2. Définir une grammaire non-ambiguë décrivant ce langage, respectant les règles de désambiguïsation. On utilisera un nouveau non-terminal par niveau de priorité.

   Correction :

$$\begin{array}{rcl} E & \rightarrow & E+F \mid E-F \mid F \\ F & \rightarrow & F \times G \mid F/G \mid G \\ G & \rightarrow & -G \mid H \\ H & \rightarrow & I \uparrow H \mid I \\ I & \rightarrow & (E) \mid Int \end{array}$$

3. Associer à chaque règle de cette grammaire un calcul permettant de déterminer le nombre d'opérateurs de l'expression reconnue.

   Correction :

$$\begin{array}{rcll} E & \rightarrow & E+F & \{E_1+F_1+1\} \\ E & \rightarrow & E-F & \{E_1+F_1+1\} \\ E & \rightarrow & F & \{F_1\} \\ F & \rightarrow & F \times G & \{F_1+G_1+1\} \\ F & \rightarrow & F/G & \{F_1+G_1+1\} \\ F & \rightarrow & G & \{G_1\} \\ G & \rightarrow & -G & \{G_1+1\} \\ G & \rightarrow & H & \{H_1\} \\ H & \rightarrow & I \uparrow H & \{I_1+H_1+1\} \\ H & \rightarrow & I & \{I_1\} \\ I & \rightarrow & (E) & \{E_1\} \\ I & \rightarrow & Int & \{0\} \end{array}$$

4. Associer à chaque règle de la grammaire non ambiguë un calcul permettant d'évaluer l'expression reconnue.

   Correction :

$$\begin{array}{rcll} E & \rightarrow & E+F & \{E_1+F_1\} \\ E & \rightarrow & E-F & \{E_1-F_1\} \\ E & \rightarrow & F & \{F_1\} \\ F & \rightarrow & F \times G & \{F_1 G_1\} \\ F & \rightarrow & F/G & \{F_1/G_1\} \\ F & \rightarrow & G & \{G_1\} \\ G & \rightarrow & -G & \{-G_1\} \\ G & \rightarrow & H & \{H_1\} \\ H & \rightarrow & I \uparrow H & \{{I_1}^{H_1}\} \\ H & \rightarrow & I & \{I_1\} \\ I & \rightarrow & (E) & \{E_1\} \\ I & \rightarrow & Int & \{val(Int_1)\} \end{array}$$

Exercice 4.12

On se pose le problème d'afficher un texte sur un terminal dont le nombre de colonnes est fixé C, de manière à ce qu'aucun mot ne soit à cheval sur deux lignes. On veut aussi « compresser » le texte en supprimant les occurrences consécutives des séparateurs (caractères blancs et passages à la ligne, notés respectivement `_` et `/` ci-dessous). Ainsi un texte source peut être :

```
qui___veut__voyager_loin__/_menage_sa__monture
```

La sortie d'un tel texte sur 20 colonnes doit être

```
qui_veut_voyager/loin_menage_sa/monture
```

Le texte source respecte la grammaire suivante :

$$\begin{array}{rcl} Phrase & \rightarrow & Mot \mid Phrase\ Sep\ Phrase \\ Mot & \rightarrow & Car \mid Mot\ Mot \\ Car & \rightarrow & \texttt{a} \mid \cdots \mid \texttt{z} \\ Sep & \rightarrow & \texttt{\_} \mid \texttt{/} \mid Sep\ Sep \end{array}$$

1. Montrer que cette grammaire est ambiguë.

2. Définir une grammaire non-ambiguë engendrant le même langage.

3. Associer à chaque règle de la grammaire non-ambiguë obtenue un calcul qui permette de calculer le texte formaté correctement. Les valeurs calculées seront des enregistrements $\{long : int; texte : string\}$ où *texte* est le texte formaté et *long* donne la longueur de la chaîne *texte* après le dernier retour à la ligne.

   Correction :

   Pour avoir un calcul simple, il faut faire des règles récursives gauches.

   $$\begin{array}{rcll} M & \rightarrow & c & \{(long = 1, texte = val(c))\} \\ M & \rightarrow & Mc & \{(long = M_1.long + 1, texte = M_1.texte\hat{}val(c))\} \\ P & \rightarrow & M & \{M_1\} \\ P & \rightarrow & PSM & \{si\ P_1.long + 1 + M_1.long > C \\ & & & \quad alors\ (long = M_1.long, texte = P_1.texte\hat{}"/"\hat{}M_1.texte) \\ & & & \quad sinon\ (long = P_1.long + 1 + M_1.long, texte = P_1.texte\hat{}"\_"\hat{}M_1.texte\} \\ S & \rightarrow & \_ & \{\} \\ S & \rightarrow & / & \{\} \\ S & \rightarrow & S\_ & \{\} \\ S & \rightarrow & S/ & \{\} \end{array}$$

4. Déterminer la valeur calculée par cette grammaire pour l'exemple ci-dessus, au cours d'une analyse ascendante gauche.

   Correction : *On calcule au cours d'une analyse ascendante du texte*

   ```
   qui___veut__voyager_loin__/_menage_sa__monture
   ```

   $M \rightarrow C$

   ```
   M ui___veut__voyager_loin__/_menage_sa__monture
   ```

   où $M \Rightarrow (long = 1, texte = "q")$
   $M \rightarrow MC$

   ```
   M i___veut__voyager_loin__/_menage_sa__monture
   ```

   x où $M \Rightarrow (long = 2, texte = "qu")$
   $M \rightarrow MC$

   ```
   M ___veut__voyager_loin__/_menage_sa__monture
   ```

où $M \Rightarrow (long = 3, texte = "qui")$
$P \rightarrow M$

```
P ___veut__voyager_loin__/_menage_sa__monture
```

où $P \Rightarrow (long = 3, texte = "qui")$
$S \rightarrow \_$

```
P S __veut__voyager_loin__/_menage_sa__monture
```

où $P \Rightarrow (long = 3, texte = "qui")$
$S \rightarrow S\_$

```
P S _veut__voyager_loin__/_menage_sa__monture
```

où $P \Rightarrow (long = 3, texte = "qui")$
etc.

```
P S M __voyager_loin__/_menage_sa__monture
```

où $P \Rightarrow (long = 3, texte = "qui")$ *et* $M \Rightarrow (long = 4, texte = "veut")$
$P \rightarrow PSM$

```
P __voyager_loin__/_menage_sa__monture
```

où $P \Rightarrow (long = 8, texte = "qui\_veut")$
etc.

```
P S M __/_menage_sa__monture
```

où $P \Rightarrow (long = 16, texte = "qui\_veut\_voyager")$ *et* $M \Rightarrow (long = 4, texte = "loin")$
comme $16 + 1 + 4 > 20$ *:*

```
P __/_menage_sa__monture
```

où $P \Rightarrow (long = 4, texte = "qui\_veut\_voyager/loin")$
etc.

5. Modifier les calculs de la grammaire de manière à ce que lorsqu'une séquence de séparateurs contient au moins un retour à la ligne, alors cela force la passage à la ligne dans le texte formaté. Sur l'exemple, cela doit donner :

   `qui_veut_voyager/loin/menage_sa_monture`

car le texte source contient un retour à la ligne entre « loin » et « menage ». (On pourra donner une valeur booléenne aux non-terminaux *Sep.*)

Correction :

$$
\begin{array}{lcll}
S & \to & \_ & \{false\} \\
S & \to & / & \{true\} \\
S & \to & S\_ & \{S_1\} \\
S & \to & S/ & \{true\} \\
M & \to & c & \{(long = 1, texte = val(c))\} \\
M & \to & Mc & \{(long = M_1.long + 1, texte = M_1.texte\hat{}val(c))\} \\
P & \to & M & \{M_1\} \\
P & \to & PSM & \{si\ S_1 ou P_1.long + 1 + M_1.long > C \\
 & & & \quad alors\ (long = M_1.long, texte = P_1.texte\hat{}"/"\hat{}M_1.texte) \\
 & & & \quad sinon\ (long = P_1.long + 1 + M_1.long, texte = P_1.texte\hat{}"\_"\hat{}M_1.texte\}
\end{array}
$$

6. (question optionnelle) Comment peut-on faire pour « justifier » le texte c'est-à-dire répartir des blancs entre les mots pour que toutes les lignes soient pleines? (On demande de ne pas utiliser de fonction auxiliaire qui reparcourerait le texte à afficher.)

Exercice 4.13

On veut piloter un traceur graphique au moyen de mots sur l'alphabet $\{\uparrow, \downarrow, \rightarrow, \leftarrow\}$. Le langage accepté est donné par la grammaire suivante :

$$L \to LL \mid \uparrow \mid \downarrow \mid \rightarrow \mid \leftarrow \mid \varepsilon$$

Le traceur est situé initialement en position $(0, 0)$. Chaque mouvement le déplace de 1mm.

1. Donnez une grammaire non ambiguë pour ce langage.

   Correction : *il existe une variante recursive droite :*

   $$L \to \varepsilon \mid \uparrow L \mid \downarrow L \mid \leftarrow L \mid \rightarrow L$$

   et une récursive gauche :

   $$L \to \varepsilon \mid L\uparrow \mid L\downarrow \mid L\leftarrow \mid L\rightarrow$$

2. Donnez les expressions associées à cette grammaire pour calculer la position du pointeur en fin de lecture du mot. Ces expressions seront de type enregistrement $\{x : int; y : int\}$.

   Correction : *Il vaut mieux prendre la variante récursive gauche :*

   $$
   \begin{array}{lcll}
   L & \to & \varepsilon & \{x = 0 \ ; \ y = 0\} \\
   L & \to & L\uparrow & \{x = L_1.x \ ; \ y = L_1.y + 1\} \\
   L & \to & L\downarrow & \{x = L_1.x \ ; \ y = L_1.y - 1\} \\
   L & \to & L\leftarrow & \{x = L_1.x - 1 \ ; \ y = L_1.y\} \\
   L & \to & L\rightarrow & \{x = L_1.x + 1 \ ; \ y = L_1.y\}
   \end{array}
   $$

   Attention à bien respecter la notation N_k pour désigner la valeur associée à la k-ième occurrence du non-terminal N dans le membre droit d'une règle.

3. Déterminer les calculs effectués lors de l'analyse ascendante gauche du mot $\uparrow\rightarrow\uparrow\leftarrow$. On dessinera au préalable l'arbre de dérivation et l'on effectuera les calculs en remontant dans cet arbre, puis on montrera comment ces calculs sont effectués par l'analyse ascendante en écrivant à chaque étape les racines des arbres de dérivation intermédiaires.

Correction : *Il faut profiter de cette question pour bien montrer l'analyse ascendante gauche. Le dessin de l'arbre est laissé au lecteur, on donne ici simplement les racines des arbres lors de l'analyse.*

$$\uparrow\rightarrow\uparrow\leftarrow$$

Réduction par la règle $L \rightarrow \varepsilon$

$$L\uparrow\rightarrow\uparrow\leftarrow$$

avec $L \Rightarrow \{x = 0; y = 0\}$

Réduction par la règle $L \rightarrow L\uparrow$

$$L\rightarrow\uparrow\leftarrow$$

avec $L \Rightarrow \{x = 0; y = 1\}$

Réduction par la règle $L \rightarrow L\rightarrow$

$$L\uparrow\leftarrow$$

avec $L \Rightarrow \{x = 1; y = 1\}$

Réduction par la règle $L \rightarrow L\uparrow$

$$L\leftarrow$$

avec $L \Rightarrow \{x = 1; y = 2\}$

Réduction par la règle $L \rightarrow L\leftarrow$

$$L$$

avec $L \Rightarrow \{x = 0; y = 2\}$

Exercice 4.14
Les deux grammaires suivantes engendrent le langage L_s des séquences d'identificateurs :

$$\begin{array}{lcl} S & \rightarrow & \varepsilon \mid Id\ S \\ S & \rightarrow & \varepsilon \mid S\ Id \end{array}$$

1. Laquelle de ces deux grammaires permet-elle une analyse descendante du langage L_s?
2. Supposons qu'on veuille réaliser une analyse ascendante pour le langage L_s avec un outil comme YACC. Laquelle des deux grammaires est préférable si on veut économiser la consommation de mémoire?

Exercice 4.15
Dans le fichier YACC des expressions arithmétiques, ajouter des expressions pour calculer :

1. le nombre d'opérateurs ;
2. l'ensemble des identificateurs utilisés ;
3. l'arbre de dérivation.

Dans chaque cas, spécifier d'abord les opérations auxiliaires nécessaires.

Chapitre 5

Syntaxe abstraite des langages

L'objectif de ce chapitre est de définir une structure de données permettant de représenter le résultat de l'analyse syntaxique sous une forme adaptée à l'analyse sémantique.

Le type et la signification d'une phrase d'un langage sont calculés à partir de son arbre de syntaxe. Mais la syntaxe (dite *concrète*) d'un langage est souvent encombrée de complications nécessaires à l'analyse syntaxique (visant en particulier à traiter les problèmes d'ambiguïté), mais nuisibles à la compréhension, donc à la définition de sa sémantique. Cela rend nécessaire la définition d'un niveau intermédiaire, celui de la *syntaxe abstraite*, dont le but est d'éliminer de l'arbre de dérivation les détails devenus inutiles à ce stade. Plus précisément, la construction d'un *arbre de syntaxe abstraite* a trois objectifs :

- éviter les ambiguïtés inhérentes aux grammaires en construisant une structure nécessairement non-ambiguë ;
- éliminer les complications inutiles introduites dans les arbres de dérivations par l'abondance de mots clés et de symboles de ponctuation ;
- réunir en une seule notion les arbres de dérivation décrivant la structure d'une phrase du langage et les valeurs associées aux tokens de cette phrase.

Dans le cas des expressions arithmétiques, les expressions de syntaxe abstraite que l'on souhaite obtenir sont de la forme $+(\mathrm{Cte}, \times(\mathrm{Cte}, \mathrm{Cte}))$. Ces expressions se prêtent en effet particulièrement bien à une évaluation, car elles correspondent à des arbres dont les nœuds sont étiquetés par des opération arithmétiques. Une évaluation revient donc à propager les calculs d'opérations arithmétiques dans l'arbre.

5.1 Grammaires abstraites

Définition 5.1 *Soit T un ensemble (fini ou infini) appelé ensemble des* étiquettes abstraites *et soit V_t le vocabulaire formé par T augmenté des parenthèses et de la virgule. Une grammaire G sur le vocabulaire terminal V_t est dite* abstraite *si*

1. *toutes les règles sont de la forme $N \to t$ ou bien $N \to t(N_1, \ldots, N_n)$ $(n \geq 1)$ où les N_i sont des non-terminaux et $t \in T$;*
2. *une étiquette t donnée n'apparaît qu'une seule fois dans l'ensemble des règles.*

Un mot engendré par une grammaire abstraite s'appelle une expression de syntaxe abstraite *ou, plus simplement,* expression abstraite.

Par exemple, si $T = \{+, \times, \text{Cte}\}$ alors la grammaire

$$E \to +(E, E) \mid \times(E, E) \mid \text{Cte}$$

sera notre grammaire abstraite des expressions arithmétiques formées d'additions, de multiplications et de constantes.

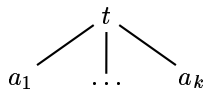
Grâce à la propriété qu'une étiquette t donnée n'apparaît qu'une seule fois dans l'ensemble des règles, on a la propriété fondamentale suivante :

Proposition 5.2 *Une grammaire abstraite est non-ambiguë.*

Notre premier objectif est donc atteint.

5.2 Arbres de syntaxe abstraite

Nous allons maintenant donner une représentation arborescente des expressions abstraites. Cela est possible grâce à la forme $N \to t(N_1, \ldots, N_k)$ des règles d'une grammaire abstraite. Une expression engendrée par le non-terminal N de cette règle sera représentée par l'arbre

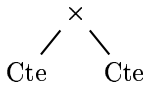


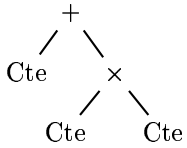
où a_i est, récursivement, l'arbre associé à l'expression abstraite e_i engendrée par le non-terminal N_i. On dira alors que cet arbre est de *sorte* N. D'où la définition :

Définition 5.3 *Étant donnée une grammaire abstraite* $G = (V_t, V_n, S, R)$*, un* arbre de syntaxe abstraite *de* sorte $N \in V_n$ *(ou plus simplement* arbre abstrait*) est un arbre* A *dont les nœuds sont étiquetés par des symboles* $t \in T$*, tel que :*

- *soit il existe une règle* $N \to t \in R$ *et* A *est réduit à une feuille étiquetée par* t *;*
- *soit il existe une règle* $N \to t(N_1, \ldots N_n)$ *telle que*
  - *(i) la racine de* A *soit étiquetée par* t *;*
  - *(ii)* A *ait exactement* n *fils qui soient, récursivement, des arbres de syntaxe abstraite de sortes respectives* $N_1, \ldots, N_n$*.*

Pour la grammaire abstraite précédente, on a par exemple les arbres de syntaxe abstraite suivants :





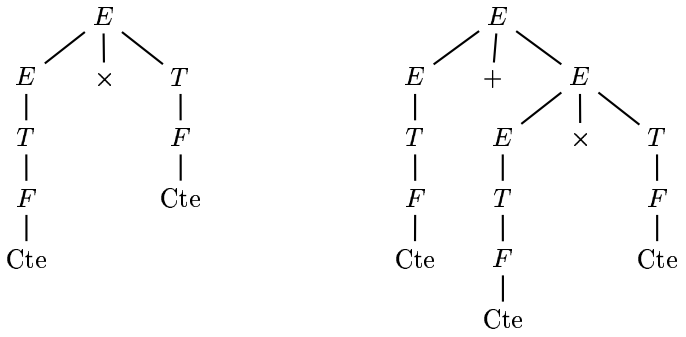
Il existe une correspondance biunivoque entre ces arbres et les mots reconnu par la grammaire abstraite : le parcours gauche d'un arbre de syntaxe abstraite permet de construire le mot correspondant. Ainsi, les exemples d'arbres précédents correspondent aux mots

$$\times(\text{Cte}, \text{Cte}) \qquad +(\text{Cte}, \times(\text{Cte}, \text{Cte}))$$

Réciproquement, on associe à chaque mot engendré par la grammaire abstraite un arbre de syntaxe abstraite par une construction analogue à la construction récursive élaborée au paragraphe 4.2 :

- à un mot engendré par une règle $N \to t$ on associe l'arbre réduit à une feuille étiquetée par t ;
- à un mot engendré par une règle $N \to t(N_1, \ldots, N_n)$ on associe l'arbre dont la racine est étiquetée par t et dont les fils sont, dans cet ordre, les arbres associés aux mots engendrés par $N_1, \ldots, N_n$.

Si l'on compare les arbres de syntaxe abstraite des mots exemples précédents avec leurs arbres de dérivation dans la grammaire concrète non-ambiguë du langage des expressions arithmétiques :



on constate que l'on a atteint notre deuxième objectif : les non-terminaux introduits pour désambiguïser ont été éliminés. Remarquons aussi que si l'expression originale contenait des parenthèses, celles-ci ne sont plus présentes dans l'arbre : c'est la structure de l'arbre qui donne le parenthésage.

5.3 Représentation des valeurs des tokens dans l'arbre

Il nous reste à représenter les valeurs des tokens dans l'arbre de syntaxe abstraite lui-même. Pour ce faire, il n'est pas nécessaire de changer la notion d'arbre : il suffit d'utiliser la possibilité d'avoir un ensemble infini d'étiquettes abstraites. Cet ensemble sera donc en fait un langage régulier (les entiers, les chaînes de caractères, etc.) aussi cet ensemble d'étiquettes abstraites sera souvent informellement donné en extension avec des points de suspension, ou encore sous forme d'une expression rationnelle.

Ainsi, pour représenter les expressions arithmétiques avec les valeurs des constantes, il suffit de prendre $T = \{+, \times, \mathrm{Cte}\} \cup \mathbb{N}$ et la grammaire abstraite suivante :

$$\begin{array}{rcl} E & \to & +(E,E) \mid \times(E,E) \mid \mathrm{Cte}(\mathit{Nat}) \\ \mathit{Nat} & \to & (0 \mid 1 \mid \cdots \mid 9)^+ \end{array}$$

Le langage défini par cette grammaire contiendra donc par exemple $\times(\mathrm{Cte}(1), \mathrm{Cte}(2))$ et $+(\mathrm{Cte}(1), \times(\mathrm{Cte}(2), \mathrm{Cte}(3)))$.

Pour résumer, la figure 5.1 montre un tableau donnant deux exemples d'expressions arithmétiques, accompagnées de leur arbre de dérivation et leur arbre abstrait associé, où l'on constate à nouveau la simplicité et la concision apportée par la notion d'arbre abstrait. Nous avons donc atteint notre dernier objectif.

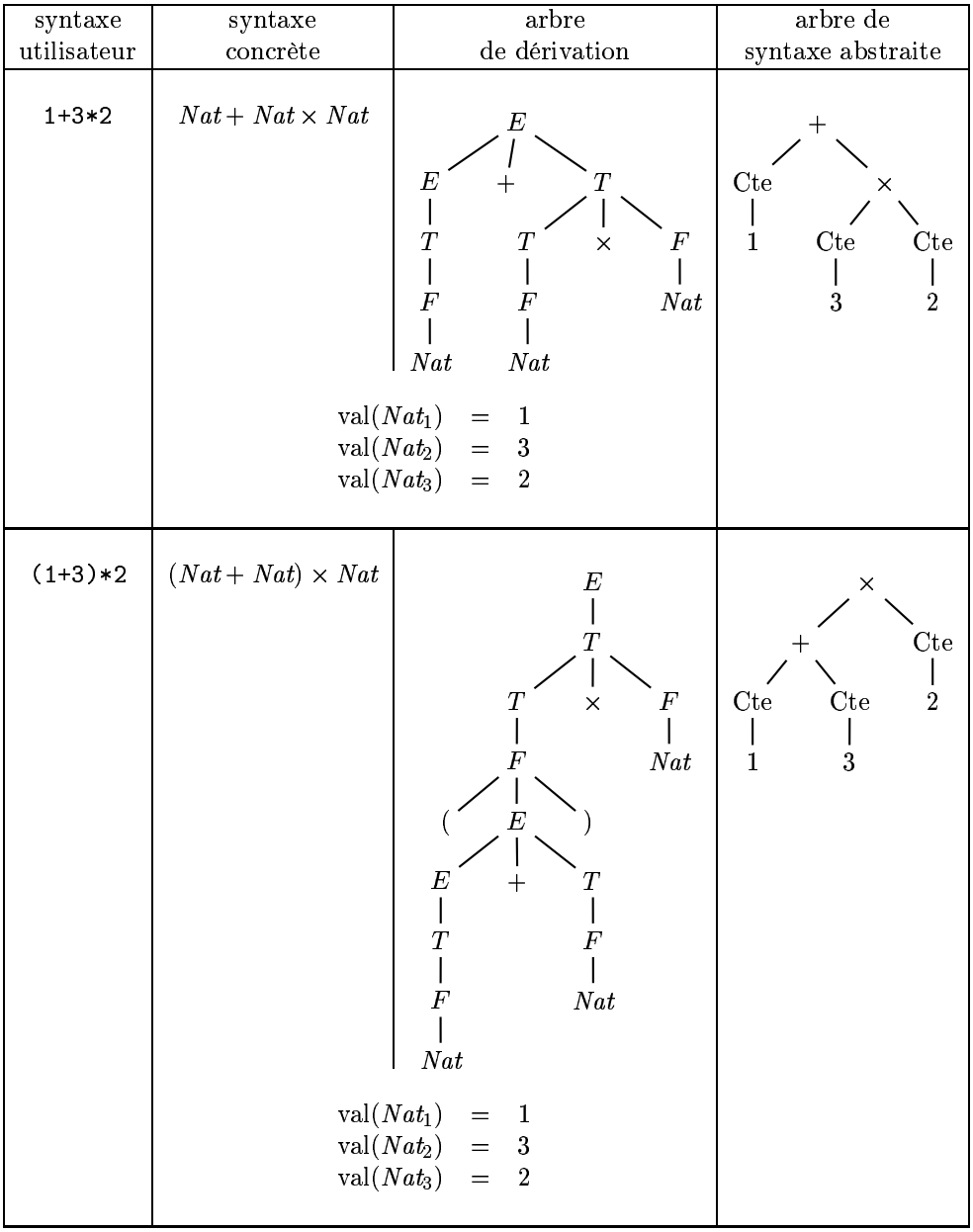


FIG. 5.1 – *Comparaison des syntaxes abstraites et concrètes*

5.4 Calcul des arbres de syntaxe abstraite

Le dernier point qui reste à traiter dans ce chapitre est la façon d'associer à chaque mot d'un langage son arbre abstrait. Cela se fait facilement dans le cadre du logiciel YACC comme expliqué au paragraphe 4.4. Ainsi, si l'on poursuit notre exemple des expressions arithmétiques, on aura :

$$\begin{array}{lcll} E & \to & E+T & \{+(E_1,T_1)\} \\ E & \to & T & \{T_1\} \\ T & \to & T\times F & \{\times(T_1,F_1)\} \\ T & \to & F & \{F_1\} \\ F & \to & Nat & \{\mathrm{Cte}(\mathrm{val}(Nat_1))\} \\ F & \to & (E) & \{E_1\} \end{array}$$

Noter l'utilisation de la fonction val pour accéder à l'unité lexicale représentée par le token *Nat*.

5.5 Codages des arbres de syntaxe abstraite

Nous terminons ce chapitre par quelques indications sur la méthode que l'on peut utiliser pour programmer ces arbres de syntaxe abstraite.

Dans le langage CAML, ce codage est immédiat en utilisant des types sommes. Ainsi, on définira le type des expressions arithmétiques par :

```
type expr =
  Cte of int
| Plus of expr * expr
| Mult of expr * expr
```

D'une manière générale, il y aura autant de types à définir que de sortes d'arbres de syntaxe abstraite, et il y aura autant de constructeurs que de règles de grammaire abstraite. Les ensembles infinis d'étiquettes abstraites comme les entiers ou les identificateurs seront codés simplement par le type de base correspondant : `int` ou `string`. Cela est notre seule façon de coder des ensembles infinis d'étiquettes abstraites. Nous imposerons donc que tout ensemble infini d'étiquettes abstraites corresponde à l'un des types de base parmi `int`, `string`, et `real`.

Dans un langage comme PASCAL ou C n'offrant pas cette possibilité des types sommes et d'allocation automatique de mémoire, il est nécessaire de coder ces arbres à l'aide de pointeurs sur des enregistrements, contenant un champ pour l'étiquette abstraite et des champs pour les sous-arbres. Dans un langage avec objets comme C++ ou JAVA, on aura avantage à coder les arbres par des classes, la construction et la destruction de ces arbres en seront facilitées.

5.6 Exercices

Exercice 5.1

1. Écrire une grammaire abstraite pour les expressions booléennes formées avec les opérations « ou », « et » et « non », et les constantes « vrai » et « faux ».

2. Donner une grammaire concrète non-ambiguë pour le langage des expressions booléennes avec toutes les opérations ci-dessus, et donner les règles de calcul de la syntaxe abstraite associées.

3. Donner un type CAML représentant cette syntaxe abstraite.

4. Écrire une fonction CAML permettant de transformer une chaine de caractères représentant une expression booléenne en une expression abstraite, en utilisant l'outil CAMLYACC.

Exercice 5.2

1. Écrire une grammaire abstraite pour les expressions arithmétiques sur l'addition, la multiplication, la soustraction, la division, l'opposé, l'exponentiation, les constantes entières ou réelles et les variables. (On ne distinguera pas les constantes entières, on les codera par des réels.)

2. Donner une grammaire concrète non-ambiguë pour le langage des expressions ci-dessus, et donner les règles de calcul de la syntaxe abstraite associées.

3. Donner un type CAML représentant cette syntaxe abstraite.

4. Écrire une fonction CAML permettant de transformer une chaîne de caractères représentant une expression arithmétique en une expression abstraite, en utilisant l'outil CAMLYACC.

Deuxième partie

Sémantique

Chapitre 1

Généralités sur la sémantique des langages

Décrire, implémenter, et même utiliser un langage de programmation nécessite une description précise et complète de la signification, on dit la *sémantique*, des différentes phrases que l'on peut écrire dans le langage.

Nous procéderons toujours en trois temps pour définir la sémantique d'un langage: définition de la syntaxe abstraite dans un premier temps; définition de la fonction de typage dans un second temps, qui est une application des expressions de syntaxe abstraite dans l'ensemble des types; définition de la fonction d'évaluation dans un troisième temps, qui est une application des expressions de syntaxe abstraite dans un ensemble de valeurs indexé par l'ensemble des types. Cette structuration de la sémantique fait clairement apparaître que le typage ne dépend pas de l'évaluation. On dit que le typage est statique. Par contre, l'évaluation peut dépendre du typage. Dans le cas où l'évaluation ne dépend pas du typage, typage et évaluation calculeront à partir de la même expression de syntaxe abstraite. Le typage ne sera alors qu'un contrôle, oublié dès qu'il a réussi. Mais le typage peut également transformer l'expression de syntaxe abstraite afin d'y inclure des informations de type utiliséees lors de l'évaluation.

Décrire la sémantique d'un langage de programmation est une œuvre complexe, et le formalisme utilisé peut dépendre du type de langage à décrire : fonctionnel, impératif, logique, à objets... Dans ce cours, nous allons utiliser un formalisme très général, celui des *règles sémantiques*, qui généralise la notion de grammaire, et même celle de grammaire avec actions présente dans YACC. L'idée consiste à calculer par récurrence sur des arbres à l'aide de règles de calcul appropriées.

1.1 Notion de jugement

Un jugement est une affirmation portant sur une expression abstraite. Avant d'en donner une définition précise, nous présentons plusieurs exemples simples de jugements :

- « l'expression $+(\mathrm{Cte}(1), \times(\mathrm{Cte}(2), \mathrm{Cte}(3)))$ a pour valeur 7 » ;
- « l'expression $+(\mathrm{Cte}(1), \times(\mathrm{Cte}(2), \mathrm{Cte}(3)))$ a pour type Int » ;
- « l'instruction $\mathrm{Write}(+(\mathrm{Cte}(3), \mathrm{Cte}(4)))$ affiche 7 » ;
- « l'expression $+(\mathrm{Cte}(3), \mathrm{Cte}(5))$ a pour code machine [PUSHI 3;PUSHI 5;ADD] ».

Un jugement affirme qu'à une certaine construction d'un langage (expression arithmétique, instruction, etc.) exprimée dans la syntaxe abstraite choisie, est associée une valeur d'un certain ensemble : un entier, un type, une chaîne de caractères, du code machine, etc.

Dans des cas plus complexes, l'affirmation pourra dépendre d'un contexte. Par exemple :

- « dans le contexte où x est une variable valant 4, l'expression $+(\text{Cte}(1), \times(\text{Var}(x), \text{Cte}(3)))$ a pour valeur 13 » ;
- « dans le contexte où x est une variable de type Real, l'expression $+(\text{Cte}(1), \times(\text{Var}(x), \text{Cte}(3)))$ a pour type Real » ;
- pour un compilateur devant engendrer du code machine à une adresse donnée, on pourra avoir des jugements du style « dans le contexte où l'adresse courante du code est 250 et x est une variable dont l'adresse est 100, l'expression $\times(\text{Cte}(3), \text{Var}(x))$ a pour code machine [250 PUSHI 3; 251 PUSH 100; 252 MUL] ».

Ces exemples conduisent à la définition suivante :

Définition 1.1 *Un* jugement *est un triplet* $(C, a, v) \in D^c \times A \times D^v$ *tel que :*

(i) *A est un ensemble d'arbres de syntaxe abstraite, contenant l'arbre a de sorte N ;*

(ii) *D^v est un ensemble appelé* domaine de valeurs *; la valeur v appartient au sous-ensemble D^v_N des valeurs de sorte N.*

(iii) *D^c est un ensemble appelé* domaine de contextes *; la valeur C appartient au sous-ensemble D^c_N des valeurs de sorte N.*

On note habituellement le jugement $(C, a, v) \in D^c \times A \times D^v$ par « $C \vdash a \Rightarrow v$ ». Dans les cas simples où le domaine de contexte est absent (c.-à-d. plus précisément $D^c = \{\emptyset\}$, le seul contexte possible est le contexte vide), on notera en général simplement $a \Rightarrow v$. Ainsi :

- $+(\text{Cte}(1), \times(\text{Cte}(2), \text{Cte}(3))) \Rightarrow 7$ est le jugement « l'expression $1 + 2 \times 3$ a pour valeur 7 », ici $D^v = \mathbb{N}$ et $D^c = \{\emptyset\}$;
- $\{x \Rightarrow 4\} \vdash +(\text{Cte}(1), \times(\text{Var}(x), \text{Cte}(3))) \Rightarrow 13$ est le jugement « dans le contexte où x est une variable valant 4, l'expression $1 + x \times 3$ a pour valeur 13 », ici $D^v = \mathbb{N}$ et D^c est l'ensemble des *environnements d'évaluation* (cf. chapitre 3).

Lorsque le domaine de valeurs D^v est un ensemble de types, par exemple $D^v = \{\text{Int}, \text{Bool}, \text{Real}\}$, nous utiliserons une variante classique de cette notation, $C \vdash a : t$, où $t \in D^v$. On écrira donc :

- $+(\text{Cte}(1), \times(\text{Cte}(2), \text{Cte}(3))) : \text{Int}$ pour le jugement « l'expression $1 + 2 \times 3$ a pour type Int » ;
- $\{x : \text{Real}\} \vdash +(\text{Cte}(1), \times(\text{Var}(x), \text{Cte}(3))) : \text{Real}$ pour le jugement « dans le contexte où x est une variable de type Real, l'expression $1 + x \times 3$ a pour type Real ».

1.2 Règles sémantiques

La définition précédente autorise l'écriture de jugements vrais aussi bien que de jugements faux :

- « l'expression $+(\text{Cte}(1), \times(\text{Cte}(2), \text{Cte}(3)))$ a pour valeur 12 » ;
- « l'expression $+(\text{Cte}(1), \times(\text{Cte}(2), \text{Cte}(3)))$ a pour type Bool » ;

sont également des jugements [1]

Étant donné un langage et une certaine catégorie de jugements associés, l'étape suivante dans la définition de la sémantique du langage consiste donc à déterminer quels sont les jugements vrais et les jugements faux, ce que nous allons faire par récurrence sur les arbres de syntaxe abstraite. Nous allons utiliser pour cela un outil classique permettant de construire des récurrences sur des arbres, appelé *règles sémantiques*. Parmi ces règles, certaines correspondent au cas de base de la récurrence, donc s'appliquent aux feuilles de l'arbre de syntaxe abstraite. Ce sont les *axiomes*, qui affirment qu'un certain jugement (ou ensemble de jugements) est vrai. Par exemple :

- si n est une constante entière, alors le jugement $\mathrm{Cte}(n) \Rightarrow n$ (c.-à-d. « l'expression réduite à la constante n a pour valeur n ») est vrai ;
- si n est une constante entière, alors le jugement $\mathrm{Cte}(n)$: Int (c.-à-d. « l'expression réduite à la constante n a pour type Int ») est vrai ;
- dans un contexte C où x est une variable de type Bool, le jugement $C \vdash \mathrm{Var}(x)$: Bool (c-à-d « l'expression réduite à l'identificateur x a pour type Bool ») est vrai.

Les règles proprement dites correspondent au cas général de la récurrence. Elles affirment la vérité d'un jugement qui constitue la *conclusion* de la règle sous l'hypothèse (de récurrence) de la vérité d'un ou plusieurs jugements appartenant aux *prémisses* de la règle. Par exemple :

- si le jugement e_1 : Int (c-à-d « l'expression e_1 a pour type Int ») est vrai et si le jugement e_2 : Int (c-à-d « l'expression e_2 a pour type Int ») est vrai alors le jugement $+(e_1, e_2)$: Int (c-à-d « l'expression $+(e_1, e_2)$ a pour type Int ») est également vrai.

Ces considérations nous conduisent à la définition formelle suivante :

Définition 1.2 *Une* règle sémantique *est un k-uplet de jugements, avec $k \geq 1$. Si $k = 1$, la règle est appelée un* axiome. *Dans le cas d'une règle $(J_1, \ldots, J_k)$ avec $k \geq 2$, J_1, ..., J_{k-1} constituent les* prémisses *de la règle et J_k en est la* conclusion.

La notation usuelle pour la règle $(J_1, \ldots, J_k)$ consiste à écrire une fraction avec les prémisses au-dessus du trait et la conclusion en-dessous :

$$\frac{J_1, \ldots, J_{k-1}}{J_k}$$

Lorsque l'ensemble des prémisses est vide, c'est-à-dire dans le cas d'un axiome, rien ne figure au dessus du trait de fraction.

Imaginons par exemple que l'on veuille décrire l'évaluation d'une expression arithmétique à l'aide de règles sémantiques. Nous sommes amenés à écrire tous les axiomes :

$$\frac{}{\mathrm{Cte}(0) \Rightarrow 0} \qquad \frac{}{\mathrm{Cte}(1) \Rightarrow 1} \qquad \text{etc.}$$

ce qui, par une notation mathématique habituelle, se note de manière finie par le *schéma d'axiomes*

$$\text{pour tout } n \in \mathbb{N}: \qquad \frac{}{\mathrm{Cte}(n) \Rightarrow n}$$

1. Nous sommes ici au cœur d'une discussion philosophique chère aux logiciens : la vérité préexiste-t-elle ou non ? Pour les uns, les règles sémantiques telles que nous allons les introduire servent à vérifier si un jugement appartient à la catégorie, connue a-priori, des jugements vrais; Pour les autres, la vérité est donnée par les règles sémantiques elles-mêmes.

De la même façon, on a les schémas de règles suivantes pour l'addition et la multiplication d'entiers :

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{+(e_1, e_2) \Rightarrow v_1 +_{int} v_2}$$

et

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{\times(e_1, e_2) \Rightarrow v_1 \times_{int} v_2}$$

En effet, e_1 n'est pas une expression abstraite, de même que v_1 n'est pas une valeur. Les schémas de règles ci-dessus désignent en fait l'ensemble infini de toutes les règles obtenues en remplaçant e_1 et e_2 par des expressions abstraites, ainsi que v_1 et v_2 par des valeurs entières. Nous ne reviendrons plus sur cette subtile nuance par la suite.

Notons la différence entre le symbole + de l'expression de syntaxe abstraite, qui est un symbole du vocabulaire de la grammaire abstraite, et le symbole $+_{int}$ que nous avons volontairement indexé par le mot *int*, qui est l'opération mathématique habituelle sur l'ensemble des valeurs entières. Par la suite, la paresse pourra nous inciter à omettre cet indice, sans pour cela créer la moindre ambiguïté.

Notons également la ressemblance entre les jugements ci-dessus et l'évaluation des arbres syntaxiques telle qu'on peut la faire dans YACC comme expliqué au paragraphe 4.4. En fait, YACC permet, grâce aux expressions accompagnant la description de la syntaxe concrète d'un langage, de spécifier des jugements mais portant dans ce cas sur les arbres de syntaxe concrète.

1.3 Notion de dérivation d'un jugement

Les règles sémantiques vont nous servir à calculer des jugements vrais en partant des axiomes et en appliquant les règles un nombre quelconque de fois :

Définition 1.3 *Une* dérivation sémantique *est une suite finie de jugements $J_0, \ldots, J_n$ telles que pour tout $i \in [0, \ldots, n]$:*

- *soit J_i est un axiome ;*
- *soit il existe une règle*

$$\frac{J_{i_1}, \ldots, J_{i_k}}{J_i}$$

avec $i_1, \ldots, i_k \in [0, \ldots, i-1]$. On dira encore qu'il s'agit d'une dérivation du jugement *J_n.*

Souvent, on omettra l'adjectif sémantique quand on parlera clairement de dérivation de jugement et pas d'autres types de dérivation. Voici par exemple une dérivation du jugement $+(\text{Cte}(1), \times(\text{Cte}(2), \text{Cte}(3))) \Rightarrow 7$:

$J_0 =$	$\text{Cte}(1) \Rightarrow 1$	axiome
$J_1 =$	$\text{Cte}(2) \Rightarrow 2$	axiome
$J_2 =$	$\text{Cte}(3) \Rightarrow 3$	axiome
$J_3 =$	$\times(\text{Cte}(2), \text{Cte}(3)) \Rightarrow 6$	règle de multiplication de prémisses J_1 et J_2
$J_4 =$	$+(\text{Cte}(1), \times(\text{Cte}(2), \text{Cte}(3))) \Rightarrow 7$	règle d'addition de prémisses J_0 et J_3

Cette notation linéaire des dérivations est ni très lisible ni très pratique. Une notation beaucoup plus agréable consiste à reproduire la structure de la dérivation. On obtient alors

un *arbre de dérivation d'un jugement* qui mime l'arbre de syntaxe abstraite. Sur l'exemple ci-dessus cela donne

$$\begin{array}{ccc} & +(\mathrm{Cte}(1), \times(\mathrm{Cte}(2), \mathrm{Cte}(3))) \Rightarrow 7 & \\ \swarrow & & \searrow \\ \mathrm{Cte}(1) \Rightarrow 1 & & \times(\mathrm{Cte}(2), \mathrm{Cte}(3)) \Rightarrow 6 \\ & \swarrow \quad \searrow & \\ & \mathrm{Cte}(2) \Rightarrow 2 \qquad \mathrm{Cte}(3) \Rightarrow 3 & \end{array}$$

Un arbre de dérivation est donc un arbre étiqueté par des jugements, dont chaque feuille est étiquetée par un axiome, et dont le jugement étiquetant un nœud interne est obtenu par une règle sémantique à partir des jugements étiquetant les fils de ce nœud. On appelle alors cet arbre *arbre de dérivation* du jugement étiquetant la racine.

Afin de reproduire l'écriture des règles sémantiques sous forme de fractions, l'habitude a été prise[2] de dessiner l'arbre de dérivation d'un jugement avec la racine en bas et les feuilles en haut. On dessine alors des traits de fraction en lieu et place des traits reliant habituellement un nœud et ses fils. Sur l'exemple, nous obtenons :

$$\dfrac{\dfrac{}{\mathrm{Cte}(1) \Rightarrow 1} \quad \dfrac{\dfrac{}{\mathrm{Cte}(2) \Rightarrow 2} \quad \dfrac{}{\mathrm{Cte}(3) \Rightarrow 3}}{\times(\mathrm{Cte}(2), \mathrm{Cte}(3)) \Rightarrow 6}}{+(\mathrm{Cte}(1), \times(\mathrm{Cte}(2), \mathrm{Cte}(3))) \Rightarrow 7}$$

1.4 Codage des règles sémantiques

Les règles sémantiques permettent de décrire la sémantique d'un langage. Mais elles constituent un langage de description très abstrait, qu'il nous faut maintenant implémenter sous la forme, par exemple, d'un interprète.

L'idée la plus naturelle serait de considérer que la sémantique d'un programme est calculée par une fonction qui prend un jugement en argument, et retourne une valeur booléenne indiquant si le jugement en question est vrai ou non :

> sémantique : contexte × expression × valeur → booléen
> {sémantique(C, e, v) retourne vrai si le jugement $C \vdash e \Rightarrow v$ est vrai }

Mais ce n'est pas ce que l'on attend d'un interprète : on a besoin de déterminer la valeur d'une expression à l'aide d'une fonction dont la spécification est la suivante :

> sémantique : contexte × expression → valeur
> { sémantique(C, e) retourne une valeur v telle que le jugement $C \vdash e \Rightarrow v$ soit vrai }

Bien entendu, une telle valeur n'existe pas toujours, auquel cas la fonction déclenchera un message d'erreur : ce sont les *erreurs de sémantique* : variable non déclarée, expression mal typée, etc. Il se peut également que plusieurs valeurs soient possibles. Nous veillerons toujours, dans ce cours, à éviter de tels systèmes de règles, dits non-de'terministes.

Le codage de cette fonction sera directement inspiré de la formulation des règles sémantiques elles-même : on utilisera une décomposition en cas suivant la forme de l'expression abstraite fournie en argument, et on codera les différents jugements figurant dans les prémisses d'une règle sous la forme d'appels récursifs. Par exemple, le jugement d'évaluation des expressions arithmétiques peut être codé en la fonction CAML suivante :

```
let rec valeur = function
```

2. Depuis trop longtemps pour en changer : la notion d'arbre de dérivation est née dans la communauté de logique mathématique. Même si cette communauté et celles de linguistique et d'informatique sont depuis longtemps conscientes des liens étroits qui existent entre leur disciplines, la formulation des concepts communs et leur notation sont souvent dissemblables d'une communaute' a' l'autre.

```
  Cte(n) -> n                (* règle (Constante)      *)
| Plus(e1,e2) ->
    let v1 = (valeur e1)     (* règle (Addition)       *)
    and v2 = (valeur e2)
    in v1+v2
| Mult(e1,e2) ->
    let v1 = (valeur e1)     (* règle (Multiplication) *)
    and v2 = (valeur e2)
    in v1*v2
```

D'une manière générale, il y aura autant de fonctions à coder que de catégories de jugements, et chaque fonction comportera autant de cas qu'il y a de règles sémantiques ou d'axiomes pour la catégorie de jugements correspondante.

1.5 Exercices

Exercice 1.1
On considère un langage qui autorise l'écriture des constantes entières en binaire en les préfixant par le caractère #. Par exemple, on pourra écrire `#10010110` pour la constante 150. La syntaxe de ces constantes en binaire est donnée par la grammaire

$$\begin{array}{lcl} N & \to & \texttt{\#}B \\ B & \to & C \mid BC \\ C & \to & \texttt{0} \mid \texttt{1} \end{array}$$

Pour déterminer la valeur de ces constantes, on définit au préalable la syntaxe abstraite suivante :

$$\begin{array}{lcl} B & \to & \mathrm{Singl}(C) \mid \mathrm{Seq}(B, C) \\ C & \to & 0 \mid 1 \end{array}$$

Le nombre `#1101` sera ainsi représenté par $\mathrm{Seq}(\mathrm{Seq}(\mathrm{Seq}(\mathrm{Singl}(1), 1), 0), 1)$.

1. Associer à chaque règle de grammaire un calcul de syntaxe abstraite.
   Correction :
   $$\begin{array}{lcll} N & \to & \texttt{\#}B & \{B_1\} \\ B & \to & C & \{\mathit{Singl}(C_1)\} \\ B & \to & BC & \{\mathit{Seq}(B_1, C_1)\} \\ C & \to & \texttt{0} & \{0\} \\ C & \to & \texttt{1} & \{1\} \end{array}$$

2. Déterminer des règles sémantiques pour l'évaluation des constantes binaires.
   Correction :
   $$\frac{}{0 \Rightarrow 0} \qquad \frac{}{1 \Rightarrow 1} \qquad \frac{}{\mathit{Singl}(c) \Rightarrow c} \qquad \frac{b \Rightarrow n}{\mathit{Seq}(b, c) \Rightarrow 2n + c}$$

3. Construire l'arbre de dérivation du jugement $\mathrm{Seq}(\mathrm{Seq}(\mathrm{Seq}(\mathrm{Singl}(1), 1), 0), 1) \Rightarrow 13$

Exercice 1.2
On veut généraliser l'exercice précédent de manière à pouvoir écrire dans une base quelconque (entre 2 et 16). On veut écrire les nombres en base b sous la forme `$`b`#`n où n est une suite de chiffres entre 0 et $b-1$. Par exemple, on pourra écrire `$5#1100` pour la constante 150 en base 5. La syntaxe de ces constantes est donnée par la pseudo-grammaire

$$\begin{array}{rcl} N & \to & \texttt{\$}\ Nat\ \texttt{\#}\ D \\ D & \to & C \mid DC \\ C & \to & \texttt{0} \mid \cdots \mid \texttt{9} \mid \texttt{A} \mid \cdots \mid \texttt{F} \\ Nat & \to & (\texttt{0} \mid \cdots \mid \texttt{9})^+ \end{array}$$

1. Définir une syntaxe abstraite pour ce langage.
   Correction :

$$\begin{array}{rcl} N & \to & Nombre(Int, D) \\ D & \to & Singl(Char) \mid Seq(D, Char) \end{array}$$

2. Associer à chaque règle de grammaire un calcul de syntaxe abstraite.
   Correction :

$$\begin{array}{rcll} N & \to & \texttt{\$}Int\texttt{\#}D & \{Nombre(val(Int_1), D_1)\} \\ D & \to & C & \{Singl(C_1)\} \\ D & \to & DC & \{Seq(D_1, C_1)\} \\ C & \to & Char & \{val(Char_1)\} \end{array}$$

3. Déterminer des règles sémantiques pour l'évaluation des constantes en base quelconque. On utilisera pour les arbres de sorte D un jugement de la forme $b \vdash d \Rightarrow v$ où b est la base, d la syntaxe abstraite représentant la séquence de chiffres et v la valeur. On fera attention à vérifier que la base est entre 2 et 16 et que tous les chiffres sont entre 0 et $b-1$.
   Correction :

$$\frac{}{n \Rightarrow n}\ pour\ tout\ n \in \{0, \ldots, 9\}$$

$$\frac{}{A \Rightarrow 10} \qquad \frac{}{B \Rightarrow 11} \qquad \vdots \qquad \frac{}{F \Rightarrow 15}$$

$$\frac{c \Rightarrow n}{b \vdash Singl(c) \Rightarrow n}\ si\ n < b$$

$$\frac{b \vdash d \Rightarrow v \qquad c \Rightarrow n}{b \vdash Seq(d, c) \Rightarrow bv + n}\ si\ n < b$$

$$\frac{b \vdash d \Rightarrow v}{Nombre(b, d) \Rightarrow v}\ si\ 2 \leq b \leq 16$$

4. Construire l'arbre de dérivation du jugement $a \Rightarrow v$ où a est l'arbre de syntaxe abstraite représentant `$5#1432` et où v est la valeur appropriée, que l'on déterminera justement en construisant cet arbre.
   Correction :

$$Nombre(5, Seq(Seq(Seq(Singl(1), 4), 3), 2) \Rightarrow ?$$

5. Donner deux exemples d'expressions concrètes et leur représentations abstraites déclenchant respectivement les deux erreurs de sémantiques possibles. Dans chacun des cas, écrire les portions d'arbres de dérivations en montrant à quel endroit l'erreur se produit.

6. Écrire un interpréteur de ce langage en CAML à l'aide de CAMLYACC.

Chapitre 2

Sémantique des expressions arithmétiques et logiques

Nous allons maintenant étudier un langage simple, que l'on trouve dans tous les langages de programmation, celui des expressions arithmétiques et logiques. Nous nous contenterons tout d'abord de traiter les expressions arithmétiques entières. Nous traiterons ensuite les expressions logiques, avant de considérer le mélange d'expressions numériques et logiques. Le traitement d'expressions arithmétiques quelconques, comportant à la fois des entiers et des réels est beaucoup plus complexe, et nous l'aborderons seulement en fin de chapitre, en ne traitant qu'un sous-cas simple. Dans tous les cas, nous réserverons le traitement des identificateurs au chapitre suivant.

Dans chaque section, nous introduirons d'abord la syntaxe abstraite choisie, avant de considérer les problèmes de typage s'il y a lieu, puis les problèmes d'évaluation.

2.1 Expressions arithmétiques

Nous conservons la syntaxe concrète des chapitres précédents, en nous limitant à l'addition et la multiplication. Ce paragraphe ne fait que regrouper éléments épars, en respectant la démarche :

1. définition de la syntaxe abstraite ;
2. définition de la grammaire concrète non-ambiguë et calcul de la syntaxe abstraite ;
3. définition de la sémantique, typage puis évaluation.

2.1.1 Syntaxe abstraite

La syntaxe abstraite que nous choisissons pour les expressions est la suivante :

$$\begin{array}{rcl} E & \to & +(E,E) \mid \times(E,E) \mid \mathrm{Cte}(Nat) \\ Nat & \to & 0 \mid 1 \mid 2 \mid \cdots \end{array}$$

$$\frac{}{\mathrm{Cte}(n) \Rightarrow n}$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{+(e_1, e_2) \Rightarrow v_1 + v_2}$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{\times(e_1, e_2) \Rightarrow v_1 \times v_2}$$

FIG. 2.1 – *Règles d'évaluation des expressions arithmétiques*

La grammaire concrète non-ambiguë accompagnée des règles de calcul de la syntaxe abstraite est alors :

$$\begin{array}{lcll} E & \to & E + T & \{+(E_1, T_1)\} \\ E & \to & T & \{T_1\} \\ T & \to & T \times F & \{\times(T_1, F_1)\} \\ T & \to & F & \{F_1\} \\ F & \to & Nat & \{\mathrm{Cte}(\mathrm{val}(Nat_1))\} \\ F & \to & (E) & \{E_1\} \end{array}$$

2.1.2 Évaluation

Nous pouvons maintenant définir la sémantique des expressions arithmétiques comme une fonction des arbres de syntaxe abstraite dans le domaine sémantique. Les règles d'évaluation sont représentées sur la figure 2.1. Dans ces règles, les opérateurs $+$ et $\times$ apparaissant à droite du symbole $\Rightarrow$ sont les opérations arithmétiques habituelles.

Par exemple, l'évaluation de l'expression arithmétique $(1 + 2) * 3$ est obtenue par l'arbre de dérivation suivant :

$$\frac{\dfrac{\dfrac{}{\mathrm{Cte}(1) \Rightarrow 1} \quad \dfrac{}{\mathrm{Cte}(2) \Rightarrow 2}}{+(\mathrm{Cte}(1), \mathrm{Cte}(2)) \Rightarrow 3} \qquad \dfrac{}{\mathrm{Cte}(3) \Rightarrow 3}}{\times(+(\mathrm{Cte}(1), \mathrm{Cte}(2)), \mathrm{Cte}(3)) \Rightarrow 9}$$

2.2 Expressions logiques

2.2.1 Syntaxe abstraite

La syntaxe abstraite que nous choisissons pour les expressions logiques est la suivante :

$$L \quad \to \quad \mathrm{And}(L, L) \mid \mathrm{Or}(L, L) \mid \mathrm{Not}(L) \mid \mathrm{True} \mid \mathrm{False}$$

Pour définir une grammaire concrète non-ambiguë pour de telles expressions, il nous faut choisir une priorité entre les opérateurs `and` , `or` et `not` . Le choix usuel dans les langages de programmation est de prendre `not` de priorité supérieure à `and` , lui-même prioritaire sur `or` , et ces deux derniers doivent associer à gauche. La grammaire accompagnée des

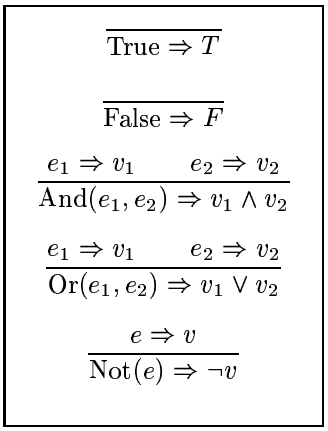


FIG. 2.2 – *Règles d'évaluation des expressions logiques.*

règles de calcul de la syntaxe abstraite est alors :

L	$\rightarrow$	L `or` M	$\{\text{Or}(L_1, M_1)\}$
L	$\rightarrow$	M	$\{M_1\}$
M	$\rightarrow$	M `and` N	$\{\text{And}(M_1, N_1)\}$
M	$\rightarrow$	N	$\{N_1\}$
N	$\rightarrow$	`not` N	$\{\text{Not}(N_1)\}$
N	$\rightarrow$	O	$\{O_1\}$
O	$\rightarrow$	`true`	$\{\text{True}\}$
O	$\rightarrow$	`false`	$\{\text{False}\}$
O	$\rightarrow$	(L)	$\{L_1\}$

2.2.2 Évaluation

Les règles d'évaluation sont représentées sur la figure 2.2. Dans ces règles, les opérateurs $\neg$, $\wedge$ et $\vee$ apparaissant à droite du symbole $\Rightarrow$ sont les opérations booléennes sur l'ensemble $\mathbb{B} = \{T, F\}$ des valeurs booléennes, correspondants respectivement à la négation, la conjonction (et) et la disjonction (ou).

L'évaluation de l'expression logique « (`true and false`) `or not true` » est obtenue par la dérivation suivante :

$$\frac{\dfrac{\overline{\text{False} \Rightarrow F} \quad \overline{\text{True} \Rightarrow T}}{\text{And}(\text{True}, \text{False}) \Rightarrow F} \qquad \dfrac{\overline{\text{True} \Rightarrow T}}{\text{Not}(\text{True}) \Rightarrow F}}{\text{Or}(\text{And}(\text{True}, \text{False}), \text{Not}(\text{True})) \Rightarrow F}$$

2.3 Expressions arithmético-logiques

Le mélange des expressions arithmétiques et logiques se produit lorsque l'on fait intervenir des opérateurs de comparaison sur les nombres. Pour simplifier, nous ne considérons ici que les opérateurs $=$ et $>$. Le mélange se produit également en présence d'une opération conditionnelle de la forme « `if then else` » rendant un entier comme résultat.

Nous allons successivement voir deux possibilités de définir la syntaxe abstraite des expressions obtenues, appelées expressions arithmético-logiques. La première est une syntaxe

abstraite à deux sortes, les expressions de sorte « arithmétique » et celles de sorte « booléenne », obtenue grosso modo en juxtaposant la syntaxe abstraite des expressions arithmétiques avec celle des expressions booléennes. Cette approche réduit le typage à sa plus simple expression, types et sortes étant identifiés. La seconde est une syntaxe abstraite à une sorte, et il faudra alors donner des règles de typage. Cette seconde approche est plus robuste, (et elle s'avérera indispensable lorsque le fragment de langage considéré comportera des déclarations d'identificateurs) mais elle nécessite une phase de vérification de types, pour rejeter les expressions sans aucun sens comme « `1 or true` » ou « `false+2` ». Nous souhaitons en effet qu'avant d'évaluer une expression, toutes les vérifications de types soient effectuées pour garantir l'aboutissement des calculs. Un langage de programmation où le typage garantit que l'évaluation pourra se faire s'appelle un langage *fortement typé*.

2.3.1 Syntaxe abstraite à deux sortes

La syntaxe abstraite à deux sortes est donnée par la grammaire :

$$\begin{array}{rcl} L & \to & \mathrm{And}(L,L) \mid \mathrm{Or}(L,L) \mid \mathrm{Not}(L) \mid \mathrm{True} \mid \mathrm{False} \mid =(E,E) \mid >(E,E) \\ E & \to & \mathrm{If}(L,E,E) \mid +(E,E) \mid \times(E,E) \mid \mathrm{Cte}(\mathit{Nat}) \\ \mathit{Nat} & \to & 0 \mid 1 \mid 2 \mid \cdots \end{array}$$

La grammaire concrète non-ambiguë accompagnée des règles de calcul de la syntaxe abstraite est alors :

L	$\to$	L `or` M	$\{\mathrm{Or}(L_1, M_1)\}$
L	$\to$	M	$\{M_1\}$
M	$\to$	M `and` N	$\{\mathrm{And}(M_1, N_1)\}$
M	$\to$	N	$\{N_1\}$
N	$\to$	`not` N	$\{\mathrm{Not}(N_1)\}$
N	$\to$	O	$\{O_1\}$
O	$\to$	`true`	$\{\mathrm{True}\}$
O	$\to$	`false`	$\{\mathrm{False}\}$
O	$\to$	(L)	$\{L_1\}$
O	$\to$	$E = E$	$\{=(E_1, E_2)\}$
O	$\to$	$E > E$	$\{>(E_1, E_2)\}$
E	$\to$	`if` L `then` E `else` E	$\{\mathrm{If}(L_1, E_1, E_2)\}$
E	$\to$	F	$\{F_1\}$
F	$\to$	$F + G$	$\{+(F_1, G_1)\}$
F	$\to$	G	$\{G_1\}$
G	$\to$	$G * H$	$\{\times(G_1, H_1)\}$
G	$\to$	H	$\{H_1\}$
H	$\to$	Nat	$\{\mathrm{Cte}(\mathrm{val}(\mathit{Nat}_1))\}$
H	$\to$	(E)	$\{E_1\}$

Notons que l'ambiguïté d'une expression de la forme « `if` E `then` E `else` $E + E$ » a été résolue en donnant priorité à l'addition.

2.3.2 Évaluation

Les règles d'évaluation sont représentées sur la figure 2.3. Remarquons que les règles d'évaluation des opérations de comparaison sont « conditionnelles » : comme mentionné au paragraphe 1.2 ce sont des schémas de règles qui dénotent l'ensemble des règles obtenues en remplaçant les variables e_1 et e_2 par des expressions abstraites quelconques, et v_1 et v_2 par des entiers satisfaisant la condition donnée.

$$\frac{}{\text{True} \Rightarrow T}$$

$$\frac{}{\text{False} \Rightarrow F}$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{\text{And}(e_1, e_2) \Rightarrow v_1 \wedge v_2}$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{\text{Or}(e_1, e_2) \Rightarrow v_1 \vee v_2}$$

$$\frac{e \Rightarrow v}{\text{Not}(e) \Rightarrow \neg v}$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{= (e_1, e_2) \Rightarrow T} \text{ si } v_1 = v_2$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{= (e_1, e_2) \Rightarrow F} \text{ si } v_1 \neq v_2$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{> (e_1, e_2) \Rightarrow T} \text{ si } v_1 > v_2$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{> (e_1, e_2) \Rightarrow F} \text{ si } v_1 \not> v_2$$

$$\frac{b \Rightarrow T \qquad e_1 \Rightarrow v_1}{\text{If}(b, e_1, e_2) \Rightarrow v_1}$$

$$\frac{b \Rightarrow F \qquad e_2 \Rightarrow v_2}{\text{If}(b, e_1, e_2) \Rightarrow v_2}$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{+(e_1, e_2) \Rightarrow v_1 + v_2}$$

$$\frac{e_1 \Rightarrow v_1 \qquad e_2 \Rightarrow v_2}{\times(e_1, e_2) \Rightarrow v_1 \times v_2}$$

$$\frac{}{\text{Cte}(n) \Rightarrow n}$$

FIG. 2.3 – *Règles d'évaluation des expressions arithmético-logiques.*

On obtient l'évaluation de l'expression arithmétique « `2>3 or 2+2=4` » par la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{}{\mathrm{Cte}(2)\Rightarrow 2}\quad\dfrac{}{\mathrm{Cte}(3)\Rightarrow 3}}{>(\mathrm{Cte}(2),\mathrm{Cte}(3))\Rightarrow \mathrm{False}}\quad\dfrac{\dfrac{\dfrac{}{\mathrm{Cte}(2)\Rightarrow 2}\quad\dfrac{}{\mathrm{Cte}(2)\Rightarrow 2}}{+(\mathrm{Cte}(2),\mathrm{Cte}(2))\Rightarrow 4}\quad\dfrac{}{\mathrm{Cte}(4)\Rightarrow 4}}{=(+(\mathrm{Cte}(2),\mathrm{Cte}(2)),\mathrm{Cte}(4))\Rightarrow \mathrm{True}}}{\mathrm{Or}(>(\mathrm{Cte}(2),\mathrm{Cte}(3)),=(+(\mathrm{Cte}(2),\mathrm{Cte}(2)),\mathrm{Cte}(4)))\Rightarrow \mathrm{True}}$$

2.3.3 Syntaxe abstraite à une sorte

Dans les langages de programmation classiques, la grammaire ne fait pas de distinction entre les expressions numériques et booléennes. Il n'y a donc qu'une seule sorte d'expressions abstraites, dont la syntaxe est alors la suivante :

$$\begin{array}{rcl} E & \to & \mathrm{And}(E,E) \mid \mathrm{Or}(E,E) \mid \mathrm{Not}(E) \mid \mathrm{True} \mid \mathrm{False} \\ & \mid & \mathrm{If}(E,E,E) \mid =(E,E) \mid >(E,E) \\ & \mid & +(E,E) \mid \times(E,E) \mid \mathrm{Cte}(\mathit{Nat}) \\ \mathit{Nat} & \to & (0 \mid \cdots \mid 9)^+ \end{array}$$

Pour écrire la grammaire concrète non-ambiguë, nous devons faire des conventions supplémentaires : on suppose classiquement que les opérateurs de comparaison sont prioritaires sur les opérateurs logiques, mais moins prioritaires que les opérations arithmétiques. Ces conventions permettent de parenthéser correctement une expression comme « $1+2>4$ `and` $5=6$ » en « $((1+2)>4)$ `and` $(5=6)$ ». La grammaire accompagnée des règles de calcul de la syntaxe abstraite est alors la suivante :

$$\begin{array}{rcll} E & \to & \texttt{if } E \texttt{ then } E \texttt{ else } E & \{\mathrm{If}(E_1,E_2,E_3)\} \\ E & \to & F & \{F_1\} \\ F & \to & F \texttt{ or } G & \{\mathrm{Or}(F_1,G_1)\} \\ F & \to & G & \{G_1\} \\ G & \to & G \texttt{ and } H & \{\mathrm{And}(G_1,H_1)\} \\ G & \to & H & \{H_1\} \\ H & \to & \texttt{not } H & \{\mathrm{Not}(H_1)\} \\ H & \to & I & \{I_1\} \\ I & \to & I=J & \{=(I_1,J_1)\} \\ I & \to & I>J & \{>(I_1,J_1)\} \\ I & \to & J & \{J_1\} \\ J & \to & J+K & \{+(J_1,K_1)\} \\ J & \to & K & \{K_1\} \\ K & \to & K*L & \{\times(K_1,L_1)\} \\ K & \to & L & \{L_1\} \\ L & \to & \texttt{true} & \{\mathrm{True}\} \\ L & \to & \texttt{false} & \{\mathrm{False}\} \\ L & \to & \mathit{Nat} & \{\mathrm{Cte}(\mathrm{val}(\mathit{Nat}_1))\} \\ L & \to & (E) & \{E_1\} \end{array}$$

2.3.4 Typage

Notre grammaire des expressions arithmétiques et logiques admet une seule sorte d'expressions abstraites, alors qu'il y a des expressions de deux types différents, Int et Bool.

$$\frac{}{\text{True} : \text{Bool}}$$

$$\frac{}{\text{False} : \text{Bool}}$$

$$\frac{e_1 : \text{Bool} \qquad e_2 : \text{Bool}}{\text{Or}(e_1, e_2) : \text{Bool}}$$

$$\frac{e_1 : \text{Bool} \qquad e_2 : \text{Bool}}{\text{And}(e_1, e_2) : \text{Bool}}$$

$$\frac{e : \text{Bool}}{\text{Not}(e) : \text{Bool}}$$

$$\frac{e_1 : \text{Int} \qquad e_2 : \text{Int}}{= (e_1, e_2) : \text{Bool}}$$

$$\frac{e_1 : \text{Int} \qquad e_2 : \text{Int}}{> (e_1, e_2) : \text{Bool}}$$

$$\frac{b : \text{Bool} \qquad e_1 : \text{Int} \qquad e_2 : \text{Int}}{\text{If}(b, e_1, e_2) : \text{Int}}$$

$$\frac{e_1 : \text{Int} \qquad e_2 : \text{Int}}{+(e_1, e_2) : \text{Int}}$$

$$\frac{e_1 : \text{Int} \qquad e_2 : \text{Int}}{\times(e_1, e_2) : \text{Int}}$$

$$\frac{}{\text{Cte}(n) : \text{Int}}$$

FIG. 2.4 – *Typage des expressions arithmético-logiques*

Le *type* d'une expression sera alors déterminé au cours de la phase d'analyse sémantique, définie au moyen des règles de typage de la figure 2.4.

Le jugement de typage de l'expression arithmético-logique « `2>3 or 2+2=4` » est alors le suivant :

$$\dfrac{\dfrac{\overline{\mathrm{Cte}(2) : \mathrm{Int}} \quad \overline{\mathrm{Cte}(3) : \mathrm{Int}}}{> (\mathrm{Cte}(2), \mathrm{Cte}(3)) : \mathrm{Bool}} \quad \dfrac{\dfrac{\overline{\mathrm{Cte}(2) : \mathrm{Int}} \quad \overline{\mathrm{Cte}(2) : \mathrm{Int}}}{+(\mathrm{Cte}(2), \mathrm{Cte}(2)) : \mathrm{Int}} \quad \overline{\mathrm{Cte}(4) : \mathrm{Int}}}{= (+(\mathrm{Cte}(2), \mathrm{Cte}(2)), \mathrm{Cte}(4)) : \mathrm{Bool}}}{\mathrm{Or}(> (\mathrm{Cte}(2), \mathrm{Cte}(3)), = (+(\mathrm{Cte}(2), \mathrm{Cte}(2)), \mathrm{Cte}(4))) : \mathrm{Bool}}$$

Sur une expression mal typée comme par exemple « `(2=3)+4` », l'échec du typage et mis en évidence par l'impossibilité de compléter l'arbre de dérivation :

$$\dfrac{\dfrac{\overline{\mathrm{Cte}(2) : \mathrm{Int}} \quad \overline{\mathrm{Cte}(3) : \mathrm{Int}}}{= (\mathrm{Cte}(2), \mathrm{Cte}(3)) : \mathrm{Bool}} \quad \overline{\mathrm{Cte}(4) : \mathrm{Int}}}{+(= (\mathrm{Cte}(2), \mathrm{Cte}(3)), \mathrm{Cte}(4)) : ?}$$

2.3.5 Évaluation

La syntaxe à une sorte possède plus d'expressions possibles que la syntaxe à deux sortes, mais le typage permet de discriminer les expressions qui n'ont pas de sens (mal typées) de celles qui en ont un, qui correspondent alors exactement avec les expressions de la grammaire à deux sortes. Il n'y a donc aucune différence avec l'évaluation des expressions arithmético-logiques décrite au paragraphe 2.3.2, puisque les expressions de syntaxe abstraite correctement typées engendrées par la grammaire à une sorte, sont identiques à celles engendrées par la grammaire à deux sortes.

2.3.6 Notion de typage fort

Dans la section sur les expressions arithmético-logiques, nous avons du faire face à deux situations différentes. Dans le cas des expressions à deux sortes, toute expression de syntaxe abstraite était évaluable, il n'y avait donc nul besoin de contrôle de type. Au contraire, dans le cas des expressions à une sorte, certaines expressions de syntaxe abstraite ne pouvaient pas être évaluées, comme l'addition des expressions `Cte(n)` et `True`. Les règles de typage servent précisément à éliminer toutes les expressions de syntaxe abstraite pour lesquelles il n'existe pas de dérivation d'évaluation.

Définition 2.1 *Un langage de programmation défini par sa syntaxe abstraite, ses règles de typage et ses règles d'évaluation sera dit* fortement typé*, s'il existe une dérivation d'évaluation pour toute expression pour laquelle il existe une dérivation de typage.*

La définition qui précède s'applique parfaitement au langage des expressions arithmétiques de cette section, dont on peut montrer sans difficulté qu'il est fortement typé:

Théorème 2.2 *Le langage des expressions arithmético-logiques à une sorte est fortement typé.*

Notre définition du typage fort est toutefois un peu simpliste, car elle ne prend pas en compte la possibilité qu'une exécution soit interrompue par la *levée d'une exception*, comme la division par zéro dans un langage d'expressions arithmétiques avec division. De façon générale, la notion d'exception est très directement liée celle de fonction partielle. Ainsi, la division est une fonction partielle, car elle n'est pas définie pour toute valeur de son second

argument. De même l'accès aux éléments d'un tableau, ou certaines fonctions de traitement des chaînes de caractères sont des fonctions partielles. Formaliser les notions de fonction partielle et d'exception sort du cadre de ce cours. Mais il faut retenir que la notion de typage fort doit prendre en compte l'existence de fonctions partielles, la levée d'une exception étant considérée comme l'aboutissement d'une dérivation d'évaluation.

Un langage de programmation fortement typé garantit que tout programme qui se compile sans erreur s'exécutera jusqu'à son terme (sauf à lever une exception, comme on vient de le voir). À titre d'exemple, le langage PASCAL n'est pas fortement typé, à cause des types pointeurs d'une part et des enregistrements avec variantes d'autre part. Le langage C ne l'est pas non plus, à cause des pointeurs, des types union, et aussi de l'absence de vérification des types des fonctions d'un module à l'autre. Les langages fortement typés les plus connus sont ceux de la famille ML (en particulier, le langage CAML), ADA et JAVA.

Notons pour terminer cette discussion que la notion de typage fort ne garantit pas que le programme soit conforme à sa spécification formelle lorsqu'elle existe, ou à une spécification informelle décrite dans un cahier des charges. Elle ne fait que garantir que l'exécution du programme *à l'aide des règles sémantiques* peut être menée à son terme. Elle ne garantit pas non plus que l'exécution du programme sur une machine particulière ira jusqu'à son terme. Il se peut en effet que les ressources de la machine (pile d'exécution, par exemple) soient insuffisantes pour une exécution donnée.

2.4 Expressions arithmétiques surchargées

La grammaire non ambiguë des expressions arithmétiques que nous avons utilisée admet un seul type d'expressions aritmétiques, les expressions arithmétiques entières, reconnues par un arbre de racine E. Mais la plupart des langages réels, comme PASCAL, admettent ce que l'on appelle de la *surcharge d'opérateurs*, c'est-à-dire que l'argument d'une opération arithmétique comme + pourra être indifféremment un entier ou un réel (représenté par exemple en virgule flottante simple précision). Nous devrons donc gérer la coexistence de plusieurs types d'expressions arithmétiques, les expressions arithmétiques entières, les expressions arithmétiques réelles simple précision, et, peut-être, les expressions arithmétiques réelles double précision, voire les expressions arithmétiques entières en précision arbitraire. Les constantes arithmétiques de ces différents types pourront être aisément distinguées par l'analyseur lexical : les constantes entières sont des suites de chiffres, comme 31415926535, alors que les constantes réelles simple précision sont constituées de deux suites de chiffres séparées par un point (dit décimal), comme 3.1415926535, suivies éventuellement d'un exposant, comme $6.02e23$. Nous supposons par la suite que l'analyseur lexical renvoie le token *Nat* pour une constante entière, et le token *Real* pour une constante réelle.

2.4.1 Syntaxe abstraite

La grammaire abstraite des expressions arithmétiques surchargées est la suivante :

$$\begin{array}{rcl} E & \rightarrow & +(E,E) \mid \times(E,E) \mid \mathrm{Cte}(Nat) \mid \mathrm{Ctereal}(Real) \\ Nat & \rightarrow & (0 \mid \cdots \mid 9)^+ \\ Real & \rightarrow & \text{cf. exercice } \mathbf{??} \end{array}$$

La grammaire concrète non-ambiguë accompagnée des règles de calcul de la syntaxe abstraite est alors :

$$\begin{array}{lcll} E & \to & E + T & \{+(E_1, T_1)\} \\ E & \to & T & \{T_1\} \\ T & \to & T \times F & \{\times(T_1, F_1)\} \\ T & \to & F & \{F_1\} \\ F & \to & Nat & \{\mathrm{Cte}(\mathrm{val}(Nat_1))\} \\ F & \to & Real & \{\mathrm{Ctereal}(\mathrm{val}(Real_1))\} \\ F & \to & (E) & \{E_1\} \end{array}$$

2.4.2 Typage

La représentation machine d'un même nombre, par exemple le nombre 2, sera toujours très différente suivant qu'il s'agit d'un 2 entier (dont la représentation binaire sur 32 bits sera en général 00000000000000000000000000000010, le bit le plus à gauche donnant le signe, 0 pour positif, et 1 pour négatif), ou d'un 2 réel (dont la représentation binaire sur 32 bits pourra être par exemple 01000000000000000000000000000010, le bit le plus à gauche donnant le signe, les 20 bits suivants la mantisse, le bit suivant le signe de l'exposant, et les 10 derniers bits donnant la valeur de l'exposant). Les machines savent additionner des nombres dans une même représentation, par exemple des entiers, mais pas des nombres de deux représentations différentes. La raison n'est pas de principe, il s'agit d'un simple problème de coût : savoir additionner des entiers et des réels nécessiterait deux additionneurs supplémentaires (entier + réel, réel + entier), ce qui coûte cher. En présence de réels double précision, il faudrait cette fois 8 additionneurs au lieu de 3.

Il y a deux solutions à ce problème : soit on interdit le mélange d'entiers et de réels ; soit la responsabilité du choix de l'additionneur adéquat, ainsi que du choix de la représentation la plus adéquate de la donnée est déléguée au logiciel, interprète ou compilateur. Dans le premier cas, une simple analyse de type permettra de décider quel circuit d'addition doit être choisi pour l'évaluation. Le second cas est plus complexe, le principe est d'additionner des entiers chaque fois que cela est possible, et des réels quand on ne peut pas faire autrement. Notre choix, qui est aussi celui de ML, est d'interdire les mélanges d'entiers et de réels [1].

Notons enfin que, même dans le cas simple que nous avons choisi, le résultat du typage n'est pas une simple réponse par oui ou non, puisqu'il faudra distinguer les opérations de même nom, comme + et ×, suivant le type de leurs arguments. La phase de typage va donc devoir calculer une nouvelle syntaxe abstraite, dans laquelle figurent des informations de type sous la forme d'opérations d'addition et de multiplication désambiguïsées, c'est-à-dire indexées par le type de leurs arguments. C'est cette syntaxe abstraite là, décrite par la grammaire ci-dessous, qui sera utilisée à l'évaluation.

$$\begin{array}{lcl} E & \to & +_{int}(E,E) \mid +_{real}(E,E) \mid \times_{int}(E,E) \mid \times_{real}(E,E) \\ & \mid & \mathrm{Cte}(Nat) \mid \mathrm{Ctereal}(Real) \end{array}$$

Les jugements de typage donnés figure 2.5 sont donc un peu plus complexes, car le domaine de valeur sera $D^v = \{\mathrm{Int}, \mathrm{Real}\} \times A$ où A est l'ensemble des arbres abstraits de la grammaire ci-dessus. Nous les noterons sous la forme $e : t \Rightarrow e'$, où e est l'expression de syntaxe abstraite, t est le type de e, et e' est l'expression de syntaxe abstraite calculée.

Le jugement de typage de l'expression arithmétique $(1 + 3) \times 3.14$ est représenté à la figure 2.6. On remarque que ce jugement échoue, à cause du mélange de types.

1. Nous parlons ici de ML standard, CAML-light, lui, interdit complètement la surcharge.

$$\frac{e_1 : \text{Int} \Rightarrow e_1' \qquad e_2 : \text{Int} \Rightarrow e_2'}{+(e_1, e_2) : \text{Int} \Rightarrow +_{int}(e_1', e_2')}$$

$$\frac{e_1 : \text{Real} \Rightarrow e_1' \qquad e_2 : \text{Real} \Rightarrow e_2'}{+(e_1, e_2) : \text{Real} \Rightarrow +_{real}(e_1', e_2')}$$

$$\frac{e_1 : \text{Int} \Rightarrow e_1' \qquad e_2 : \text{Int} \Rightarrow e_2'}{\times(e_1, e_2) : \text{Int} \Rightarrow \times_{int}(e_1', e_2')}$$

$$\frac{e_1 : \text{Real} \Rightarrow e_1' \qquad e_2 : \text{Real} \Rightarrow e_2'}{\times(e_1, e_2) : \text{Real} \Rightarrow \times_{real}(e_1', e_2')}$$

$$\frac{}{\text{Cte}(n) : \text{Int} \Rightarrow \text{Cte}(n)}$$

$$\frac{}{\text{Ctereal}(r) : \text{Real} \Rightarrow \text{Ctereal}(r)}$$

FIG. 2.5 – *Typage des expressions arithmétiques surchargées*

$$\frac{\dfrac{\dfrac{}{\text{Cte}(3) : \text{Int} \Rightarrow \text{Cte}(3)} \quad \dfrac{}{\text{Cte}(1) : \text{Int} \Rightarrow \text{Cte}(1)}}{+(\text{Cte}(1), \text{Cte}(3)) : \text{Int} \Rightarrow +_{int}(\text{Cte}(1), \text{Cte}(3))} \qquad \dfrac{}{\text{Ctereal}(3.14) : \text{Real} \Rightarrow \text{Ctereal}(3.14)}}{\times(+(\text{Cte}(1), \text{Cte}(3)), \text{Ctereal}(3.14)) : ? \Rightarrow ?}$$

FIG. 2.6 – *Échec du jugement de typage de* $(1 + 3) \times 3.14$

2.4.3 Évaluation

Le typage d'une expression est toujours *statique*: il ne dépend pas de l'évaluation de l'expression. Par contre, l'évaluation des expressions peut dépendre du typage par le biais du calcul d'une nouvelle syntaxe abstraite comme précédemment. Les phases de typage et d'évaluation restent toutefois clairement séparées.

L'évaluation des expressions de syntaxe abstraite (décorée par les types) d'une arithmétique surchargée n'est pas très différente de l'évaluation déja donnée pour les expressions non-surchargées. Nous laissons au lecteur le soin d'en élaborer les règles.

2.5 Exercices

Exercice 2.1
On prolonge l'exercice 5.1.

1. Coder les règles d'évaluation des expressions booléennes en une fonction qui étant donnée une expression booléenne abstraite retourne sa valeur.

2. Écrire un programme CAML qui lit au clavier une expression booléenne et affiche sa valeur à l'écran.

Exercice 2.2
On prolonge l'exercice 5.2.

1. Coder les règles d'évaluation des expressions arithmétiques en une fonction qui étant donnée une expression arithmétique abstraite retourne sa valeur.

2. Écrire un programme CAML qui lit au clavier une expression arithmétique et affiche sa valeur à l'écran.

Exercice 2.3
Reprendre et combiner les deux exercices précédents pour écrire un programme CAML qui lit au clavier une expression arithmético-logique et affiche sa valeur à l'écran.

Exercice 2.4
Définir les syntaxe concrète puis abstraite, les règles de typage et d'évaluation pour un langage d'expressions mélangeant les entiers et les chaînes de caractères, muni d'un opérateur de concaténation noté « $\cdot$ », d'une fonction `len` retournant la longueur d'une chaîne, et d'une fonction `sub` à trois arguments telles que `sub`(s, n, m) retourne la sous-chaîne de s entre les n-ième et m-ième caractères. Donner les dérivations des jugements de typage et d'évaluation de

```
sub("bonjour",len("bon")+1,7)."soir"
```

Exercice 2.5

1. Montrer que le langage des expressions arithmétiques surchargées est fortement typé.

2. Définir un langage d'expressions arithmético-logiques avec surcharge des opérateurs arithmétiques. On donnera d'abord, (i) sa syntaxe abstraite à une sorte, (ii) sa syntaxe concrète acompagnée des règles de calcul de la syntaxe abstraite, (iii) ses règles de typage, (iv) ses règles d'évaluation. On montrera pour terminer que le langage obtenu est fortement typé.

Chapitre 3

Variables et environnements

Ce chapitre s'intéresse aux problèmes de sémantiques liés à l'utilisation de variables.

Nous avons supposé jusqu'à maintenant que nos expressions arithmétiques ne contenaient pas d'identificateurs. Nous allons maintenant définir des *expressions arithmétiques étendues*, en introduisant dans la grammaire une construction `let`. Nous supposerons dans un premier temps comme dans le paragraphe 2.1 que nous avons un seul type de constantes, les constantes entières. Les jugements de typage que nous allons introduire consisteront simplement à détecter les variables non déclarées. Dans un second temps, nous réintroduirons des booléens dans les expressions, et les jugements de typage détecteront également les variables dont le type déclaré n'est pas le bon.

3.1 Cas des expressions arithmétiques seules

Pour définir des variables locales à une expression, nous introduisons la syntaxe classique de ML : `let` $x = expr_1$ `in` $expr_2$. La syntaxe concrète est donc

$$\begin{array}{lcl} E & \rightarrow & \texttt{let}\ Id\ =\ E\ \texttt{in}\ E \mid E+E \mid E\times E \mid Nat \mid Id \mid (E) \\ Id & \rightarrow & \text{cf. exercice } \textbf{??} \end{array}$$

où *Id* reconnaît les identificateurs. Le `let` introduit une nouvelle ambiguïté dans cette grammaire : une expression comme « `let` $x = 1$ `in` $x + 2$ » peut être parenthésée comme « (`let` $x = 1$ `in` x)$+2$ » ou bien « `let` $x = 1$ `in` $(x+2)$ ». Le choix que nous faisons, celui qui est classique, est de donner priorité aux opérations par rapport au `let`, et le deuxième parenthésage est donc celui souhaité. Nous commençons par donner une grammaire concrète non-ambiguë en conséquence, après avoir défini au préalable la syntaxe abstraite voulue.

3.1.1 Syntaxe abstraite

La syntaxe abstraite choisie pour les expressions arithmétiques étendues est donnée par la grammaire

$$E \rightarrow \text{Let}(String, E, E) \mid +(E,E) \mid \times(E,E) \mid \text{Cte}(Int) \mid \text{Var}(String)$$

La grammaire concrète non-ambiguë accompagnée des règles de calcul de la syntaxe abstraite est alors :

$$
\begin{array}{lcll}
E & \to & \texttt{let}Id = E \texttt{ in } E & \{\mathrm{Let}(\mathrm{val}(Id_1), E_1, E_2)\} \\
E & \to & F & \{F_1\} \\
F & \to & F + G & \{+(F_1, G_1)\} \\
F & \to & G & \{G_1\} \\
G & \to & G * H & \{\times(G_1, H_1)\} \\
G & \to & H & \{H_1\} \\
H & \to & Nat & \{\mathrm{Cte}(\mathrm{val}(Nat_1))\} \\
H & \to & Id & \{\mathrm{Var}(\mathrm{val}(Id_1))\} \\
H & \to & (E) & \{E_1\}
\end{array}
$$

3.1.2 Typage

On pourrait penser que le typage des expressions arithmétiques étendues est très simple, dans la mesure où le seul type possible est Int. En fait, il est plus complexe qu'il n'y parait, car il se peut qu'une expression ne soit pas typable, lorsqu'un identificateur est utilisé dans une expression arithmétique sans être déclaré auparavant dans un `let`. Si cette vérification n'avait pas lieu, le typage n'assurerait pas son rôle, qui est de garantir que le programme peut être évalué. Cette vérification demande l'utilisation d'un *environnement de liaison* permettant de stocker le type des identificateurs, technique que nous serons également amenés à utiliser pour l'évaluation.

On peut se demander pourquoi la grammaire n'est pas modifiée afin d'éviter cet inconvénient : c'est parce que ce n'est pas possible avec une grammaire hors-contexte (cf. exercice 3.2).

Définition 3.1 *Un* environnement de typage *est une liste (donc ordonnée) de couples (identificateur,type), notée* $\{x_1 : t_1, \ldots, x_n : t_n\}$. *On notera par* $\{\}$ *l'environnement vide, et par* $\Gamma \cdot \Gamma'$ *la concaténation des environnements* Γ *et* Γ'. *Le même identificateur peut apparaître plusieurs fois dans un environnement, auquel cas la notation* $(x, v) \in \Gamma$ *signifie que* $\Gamma = \Gamma_1 \cdot \{(x, v)\} \cdot \Gamma_2$ *et qu'aucune paire de la forme* (x, v') *ne figure dans* Γ_2.

Notons que par conséquent, les environnements se comportent comme des piles vis-à-vis de l'opérateur d'appartenance, où la tête de pile est écrite à droite.

Définition 3.2 *Un* jugement de typage *est un jugement* (Γ, e, t) *que l'on écrit* $\Gamma \vdash e : t$, *où* Γ *est un environnement de typage (le domaine de contexte est donc l'ensemble des environnements de typage), e est une expression arithmétique abstraite étendue, et t est un* type *(le domaine de valeurs est donc l'ensemble des types).*

Pour l'instant l'ensemble des types est réduit à {Int}. Ce ne sera pas toujours le cas, en particulier s'il y a des expressions arithmético-logiques ou bien des expressions arithmétiques surchargées. Les règles de typage sont décrites à la figure 3.1.

La figure 3.2 montre le typage de l'expression arithmétique étendue `let` n $=$ 2 `in let` $m = n + 1$ `in` $m + n$ dans l'environnement vide. La dérivation de typage est présentée en trois parties, de manière à tenir dans la largeur de la page. Les références (1) et (2) indiquent où se poursuit la dérivation.

3.1.3 Évaluation

Afin de traiter l'évaluation de ces expressions arithmétiques étendues, nous utilisons un nouveau type de jugement, qui fait appel à un environnement de liaison dont le but est de

$$\frac{}{\Gamma \vdash \mathrm{Var}(x) : \mathrm{Int}} \text{ si } x : \mathrm{Int} \in \Gamma$$

$$\frac{}{\Gamma \vdash Cte(n) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash \times(e_1, e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash +(e_1, e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \cdot \{x : \mathrm{Int}\} \vdash e_2 : \mathrm{Int}}{\Gamma \vdash \mathrm{Let}(x, e_1, e_2) : \mathrm{Int}}$$

FIG. 3.1 – *Typage des expressions arithmétiques étendues*

$$(1) : \quad \frac{\overline{\{n : \mathrm{Int}\} \vdash \mathrm{Var}(n) : \mathrm{Int}} \quad \overline{\{n : \mathrm{Int}\} \vdash \mathrm{Cte}(1) : \mathrm{Int}}}{\{n : \mathrm{Int}\} \vdash +(\mathrm{Var}(n), \mathrm{Cte}(1)) : \mathrm{Int}}$$

$$(2) : \quad \frac{\overline{\{n : \mathrm{Int},\ m : \mathrm{Int}\} \vdash \mathrm{Var}(m) : \mathrm{Int}} \quad \overline{\{n : \mathrm{Int},\ m : \mathrm{Int}\} \vdash \mathrm{Var}(n) : \mathrm{Int}}}{\{n : \mathrm{Int},\ m : \mathrm{Int}\} \vdash +(\mathrm{Var}(m), \mathrm{Var}(n)) : \mathrm{Int}}$$

$$\frac{\overline{\{\} \vdash \mathrm{Cte}(2) : \mathrm{Int}} \quad \dfrac{(1) \quad (2)}{\{n : \mathrm{Int}\} \vdash \mathrm{Let}(m, +(\mathrm{Var}(n), \mathrm{Cte}(1)), +(\mathrm{Var}(m), \mathrm{Var}(n))) : \mathrm{Int}}}{\{\} \vdash \mathrm{Let}(n, \mathrm{Cte}(2), \mathrm{Let}(m, +(\mathrm{Var}(n), \mathrm{Cte}(1)), +(\mathrm{Var}(m), \mathrm{Var}(n)))) : \mathrm{Int}}$$

FIG. 3.2 – *Typage de l'expression arithmétique étendue* **let** $n = 2$ **in let** $m = n + 1$ **in** $m + n$

$$\frac{}{\Gamma \vdash \mathrm{Var}(x) \Rightarrow v} \text{ si } x = v \in \Gamma$$

$$\frac{}{\Gamma \vdash Cte(n) \Rightarrow n}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v \qquad \Gamma \vdash e_2 \Rightarrow v'}{\Gamma \vdash \times(e_1, e_2) \Rightarrow v \times v'}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v \qquad \Gamma \vdash e_2 \Rightarrow v'}{\Gamma \vdash +(e_1, e_2) \Rightarrow v + v'}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma \cdot \{x = v_1\} \vdash e_2 \Rightarrow v_2}{\Gamma \vdash \mathrm{Let}(x, e_1, e_2) \Rightarrow v_2}$$

FIG. 3.3 – *Évaluation des expressions arithmétiques étendues*

stocker cette fois les valeurs des identificateurs.

Définition 3.3 *Un* environnement de valeurs *est une liste de couples (identificateur,valeur), notée* $\{x_1 = v_1, \ldots, x_n = v_n\}$.

Un jugement d'évaluation *est un jugement* (Γ, e, v) *où* Γ *est un environnement de valeurs,* e *une expression arithmétique abstraite étendue, et* v *est une valeur.*

Étant donnée une expression arithmétique étendue e, le jugement d'évaluation s'écrira $\Gamma \vdash e \Rightarrow v$, où v est une valeur appartenant au domaine d'interprétation, c'est-à-dire les entiers naturels.

Les règles d'évaluation sont décrites sur la figure 3.3. La règle du `let` présente deux particularités. Tout d'abord, la valeur v_1 calculée dans la branche gauche de l'arbre de dérivation est réutilisée dans la branche droite. Cela impose un ordre, gauche-droit, sur le calcul de l'arbre de dérivation. Deuxièmement, l'identificateur x défini par l'expression `let` $x = e_1$ `in` e_2 prend la valeur v_1 dans l'expression e_2, et dans elle seule. On dit que la *portée* du `let` est l'expression e_2, ou que les occurrences de x dans e_2 sont dans la portée du `let`. Il se peut que cette expression `let` soit elle-même à l'intérieur d'une autre expression `let`, auquel cas il est important de lier les diverses occurrences de x aux bonnes valeurs. Cela est expliqué en détail plus loin, après les exemples.

Théorème 3.4 *Le langage des expressions arithmétiques étendues est fortement typé.*

La figure 3.4 montre l'évaluation de l'expression arithmétique étendue `let` n $=$ 2 `in let` $m = n + 1$ `in` $m + n$ dans l'environnement vide.

Ces exemples montrent que les identificateurs définis au moyen de la construction `let` ont une *portée* bien précise, qui est définie par le parenthésage des expressions `let`. Cette construction permet d'avoir ce que l'on appelle des *variables locales*, fondamentalement différentes des variables globales existant dans certains langages de programmation. Une conséquence importante est que le symbole d'égalité que nous avons utilisé dans la construction (`let` $x = e_1$ `in` e_2) désigne une *définition* ou une *redéfinition* de l'identificateur x, et non pas une *affectation*. En effet, une affectation consiste à écrire une valeur dans la cellule mémoire associée à l'identificateur x : toute valeur écrite antérieurement sera donc remplacée par la nouvelle (on dit *écrasée*). Une redéfinition consiste en une association temporaire d'une valeur à l'identificateur x. Elle ne doit surtout pas effacer une association plus ancienne encore *active*, c'est-à-dire susceptible d'être utilisée ultérieurement. L'exemple de la

$$
\dfrac{
\dfrac{
\dfrac{}{\{n=2,m=3\} \vdash \mathrm{Var}(m) \Rightarrow 3}
\quad
\dfrac{}{\{n=2,m=3\} \vdash \mathrm{Var}(n) \Rightarrow 2}
}{\{n=2,m=3\} \vdash +(\mathrm{Var}(m),\mathrm{Var}(n)) \Rightarrow 5}\ (3)
\qquad
\dfrac{
\dfrac{}{\{n=2\} \vdash \mathrm{Var}(n) \Rightarrow 2}
\quad
\dfrac{}{\{n=2\} \vdash \mathrm{Cte}(1) \Rightarrow 1}
}{\{n=2\} \vdash +(\mathrm{Var}(n),\mathrm{Cte}(1)) \Rightarrow 3}\ (2)
}{\{n=2\} \vdash \mathrm{Let}(m,+(\mathrm{Var}(n),\mathrm{Cte}(1)),+(\mathrm{Var}(m),\mathrm{Var}(n))) \Rightarrow 5}\ (1)
$$

$$
\dfrac{
\dfrac{}{\{\} \vdash \mathrm{Cte}(2) \Rightarrow 2}
\quad
\dfrac{
\dfrac{}{\{n=2\} \vdash \mathrm{Let}(m,+(\mathrm{Var}(n),\mathrm{Cte}(1)),+(\mathrm{Var}(m),\mathrm{Var}(n))) \Rightarrow 5}\ (1)
}{\{\} \vdash \mathrm{Let}(n,\mathrm{Cte}(2),\mathrm{Let}(m,+(\mathrm{Var}(n),\mathrm{Cte}(1)),+(\mathrm{Var}(m),\mathrm{Var}(n)))) \Rightarrow 5}
$$

FIG. 3.4 – *Évaluation de l'expression arithmétique étendue* `let` $n = 2$ `in let` $m = n + 1$ `in` $m + n$

$$
\dfrac{
\dfrac{
\dfrac{}{\{n=2\} \vdash \mathrm{Var}(n) \Rightarrow 2}
\quad
\dfrac{}{\{n=2\} \vdash \mathrm{Cte}(2) \Rightarrow 2}
}{\{n=2\} \vdash +(\mathrm{Var}(n),\mathrm{Cte}(2)) \Rightarrow 4}\ (3)
\qquad
\dfrac{}{\{n=2,n=4\} \vdash \mathrm{Var}(n) \Rightarrow 4}
}{\{n=2\} \vdash \mathrm{Let}(n,+(\mathrm{Var}(n),\mathrm{Cte}(2)),\mathrm{Var}(n)) \Rightarrow 4}\ (2)
$$

$$
\dfrac{
\dfrac{
\dfrac{}{\{n=2\} \vdash \mathrm{Let}(n,+(\mathrm{Var}(n),\mathrm{Cte}(2)),\mathrm{Var}(n)) \Rightarrow 4}\ (2)
\qquad
\dfrac{}{\{n=2\} \vdash \mathrm{Var}(n) \Rightarrow 2}
}{\{n=2\} \vdash +(\mathrm{Let}(n,+(\mathrm{Var}(n),\mathrm{Cte}(2)),\mathrm{Var}(n)),\mathrm{Var}(n)) \Rightarrow 6}\ (1)
$$

$$
\dfrac{
\dfrac{}{\{\} \vdash \mathrm{Cte}(2) \Rightarrow 2}
\quad
\dfrac{}{\{n=2\} \vdash +(\mathrm{Let}(n,+(\mathrm{Var}(n),\mathrm{Cte}(2)),\mathrm{Var}(n)),\mathrm{Var}(n)) \Rightarrow 6}\ (1)
}{\{\} \vdash \mathrm{Let}(n,\mathrm{Cte}(2),+(\mathrm{Let}(n,+(\mathrm{Var}(n),\mathrm{Cte}(2)),\mathrm{Var}(n)),\mathrm{Var}(n))) \Rightarrow 6}
$$

FIG. 3.5 – *Évaluation de l'expression arithmétique étendue* `let` $n = 2$ `in` (`let` $n = n + 2$ `in` n) $+ n$

figure 3.5 illustre le problème en imbriquant deux constructions `let` portant sur le même nom d'identificateur.

3.2 Les expressions arithmético-logiques

L'extension de l'étude précédente au cas des expressions arithmético-logiques ne pose pas de difficulté supplémentaires, il s'agit simplement d'introduire le type Bool dans l'ensemble des types.

On obtient la grammaire abstraite :

$$\begin{array}{rcl} E & \rightarrow & \mathrm{And}(E,E) \mid \mathrm{Or}(E,E) \mid \mathrm{Not}(E) \mid \mathrm{True} \mid \mathrm{False} \\ & \mid & \mathrm{If}(E,E,E) \mid \mathrm{Let}(Id,E,E) \mid =(E,E) \mid >(E,E) \\ & \mid & +(E,E) \mid \times(E,E) \mid \mathrm{Cte}(Nat) \mid \mathrm{Var}(Id) \\ Nat & \rightarrow & (0 \mid \cdots \mid 9)^+ \\ Id & \rightarrow & (\mathtt{a} \mid \ldots \mid \mathtt{z})^+ \end{array}$$

La grammaire concrète non ambiguë accompagnée des calculs de syntaxe abstraite devient :

$$\begin{array}{rcll} E & \rightarrow & \mathtt{if}\ E\ \mathtt{then}\ E\ \mathtt{else}\ E & \{\mathrm{If}(E_1,E_2,E_1)\} \\ E & \rightarrow & \mathtt{let}Id\ =\ E\ \mathtt{in}\ E & \{\mathrm{Let}(\mathrm{val}(Id_1),E_1,E_1)\} \\ E & \rightarrow & F & \{F_1\} \\ F & \rightarrow & F\ \ \mathtt{or}\ \ G & \{\mathrm{Or}(F_1,G_1)\} \\ F & \rightarrow & G & \{G_1\} \\ G & \rightarrow & G\ \ \mathtt{and}\ \ H & \{\mathrm{And}(G_1,H_1)\} \\ G & \rightarrow & H & \{H_1\} \\ H & \rightarrow & \mathtt{not}\ \ H & \{\mathrm{Not}(H_1)\} \\ H & \rightarrow & I & \{I_1\} \\ I & \rightarrow & I = J & \{=(I_1,J_1)\} \\ I & \rightarrow & I > J & \{>(I_1,J_1)\} \\ I & \rightarrow & J & \{J_1\} \\ J & \rightarrow & J + K & \{+(J_1,K_1)\} \\ J & \rightarrow & K & \{K_1\} \\ K & \rightarrow & K * L & \{\times(K_1,L_1)\} \\ K & \rightarrow & L & \{L_1\} \\ L & \rightarrow & \mathtt{true} & \{\mathrm{True}\} \\ L & \rightarrow & \mathtt{false} & \{\mathrm{False}\} \\ L & \rightarrow & Nat & \{\mathrm{Cte}(\mathrm{val}(Nat_1))\} \\ L & \rightarrow & Id & \{\mathrm{Var}(\mathrm{val}(Id_1))\} \\ L & \rightarrow & (E) & \{E_1\} \end{array}$$

Notons tout de même qu'ici, il ne serait définitivement pas possible d'étendre la grammaire à deux sortes du paragraphe 2.3.1 car le fragment de grammaire

$$\begin{array}{rcl} E & \rightarrow & Id \\ L & \rightarrow & Id \end{array}$$

introduirait forcément une ambiguïté.

3.2.1 Typage

Les règles de typage sont données figure 3.6.

3.2.2 Évaluation

Les règles d'évaluation sont présentées figure 3.7.

Théorème 3.5 *Le langage des expressions arithmético-logiques étendues est fortement typé.*

$$\frac{}{\Gamma \vdash \text{True} : \text{Bool}}$$

$$\frac{}{\Gamma \vdash \text{False} : \text{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{Bool} \qquad \Gamma \vdash e_2 : \text{Bool}}{\Gamma \vdash \text{Or}(e_1, e_2) : \text{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{Bool} \qquad \Gamma \vdash e_2 : \text{Bool}}{\Gamma \vdash \text{And}(e_1, e_2) : \text{Bool}}$$

$$\frac{\Gamma \vdash e : \text{Bool}}{\Gamma \vdash \text{Not}(e) : \text{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash = (e_1, e_2) : \text{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash > (e_1, e_2) : \text{Bool}}$$

$$\frac{\Gamma \vdash b : \text{Bool} \qquad \Gamma \vdash e_1 : t \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash \text{If}(b, e_1, e_2) : t}$$

$$\frac{}{\Gamma \vdash \text{Var}(x) : t} \text{ si } x : t \in \Gamma$$

$$\frac{}{\Gamma \vdash \text{Cte}(n) : \text{Int}}$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash \times (e_1, e_2) : \text{Int}}$$

$$\frac{\Gamma \vdash e_1 : \text{Int} \qquad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash + (e_1, e_2) : \text{Int}}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \cdot \{x : t_1\} \vdash e_2 : t_2}{\Gamma \vdash \text{Let}(x, e_1, e_2) : t_2}$$

FIG. 3.6 – *Typage des expressions arithmético-logiques étendues.*

$$\frac{}{\Gamma \vdash \text{True} \Rightarrow T}$$

$$\frac{}{\Gamma \vdash \text{False} \Rightarrow F}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash \text{And}(e_1, e_2) \Rightarrow v_1 \wedge v_2}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash \text{Or}(e_1, e_2) \Rightarrow v_1 \vee v_2}$$

$$\frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash \text{Not}(e) \Rightarrow \neg v}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash\ =(e_1, e_2) \Rightarrow T} \text{ si } v_1 = v_2$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash\ =(e_1, e_2) \Rightarrow F} \text{ si } v_1 \neq v_2$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash\ >(e_1, e_2) \Rightarrow T} \text{ si } v_1 > v_2$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash\ >(e_1, e_2) \Rightarrow F} \text{ si } v_1 \not> v_2$$

$$\frac{\Gamma \vdash b \Rightarrow T \qquad \Gamma \vdash e_1 \Rightarrow v_1}{\Gamma \vdash \text{If}(b, e_1, e_2) \Rightarrow v_1}$$

$$\frac{\Gamma \vdash b \Rightarrow F \qquad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash \text{If}(b, e_1, e_2) \Rightarrow v_2}$$

$$\frac{}{\Gamma \vdash \text{Var}(x) \Rightarrow v} \text{ si } x = v \in \Gamma$$

$$\frac{}{\Gamma \vdash \mathit{Cte}(n) \Rightarrow n}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v \qquad \Gamma \vdash e_2 \Rightarrow v'}{\Gamma \vdash\ \times(e_1, e_2) \Rightarrow v \times v'}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v \qquad \Gamma \vdash e_2 \Rightarrow v'}{\Gamma \vdash\ +(e_1, e_2) \Rightarrow v + v'}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma \cdot \{x = v_1\} \vdash e_2 \Rightarrow v_2}{\Gamma \vdash \text{Let}(x, e_1, e_2) \Rightarrow v_2}$$

FIG. 3.7 – *Évaluation des expressions arithmético-logiques étendues*

3.3 Exercices

Exercice 3.1
Pour chacune des expressions suivantes, donner sa syntaxe abstraite, la typer, puis l'évaluer :

1. `let` $x = 2$ `in let` $n = x + 1$ `in let` $n = n + x$ `in` $n * x$

2. `let` $x =$ `let` $x = 2$ `in` $x + 3$ `in let` $x = x + 4$ `in` $x + 5$

Exercice 3.2
Montrer que le langage des expressions arithmétiques étendues bien typés (c'est-à-dire dans lesquelles tout identificateur est déclaré avant d'être utilisé) ne peut pas être engendré par une grammaire hors-contexte.

Exercice 3.3
Le langage des polynômes à coefficients entiers sur la variable x est donné par la grammaire concrète ambiguë suivante :

$$\begin{array}{rcl} \mathit{Poly} & \rightarrow & \mathit{Poly}\ \texttt{+}\ \mathit{Poly} \mid \mathit{Poly}\ \texttt{*}\ \mathit{Poly} \mid \mathit{Poly}\ \texttt{\^{}}\ \mathit{Nat} \\ & \mid & \mathit{Nat} \mid \texttt{-}\ \mathit{Nat} \mid \texttt{x} \mid \texttt{(}\ \mathit{Poly}\ \texttt{)} \\ \mathit{Nat} & \rightarrow & (\texttt{0} - \texttt{9})^+ \end{array}$$

Pour résoudre les ambiguïtés, on convient que le `+` et le `*` associent à gauche, et que `^` est prioritaire sur `*` lui-même prioritaire sur `+`. Ainsi, le polynôme $P_0 = (x+1)(2x^2-1)$ pourra être écrit `(x+1)*(2*x^2+-1)`.

Pour représenter abstraitement les polynômes, on définit la grammaire abstraite suivante :

$$\begin{array}{rcl} \mathit{Poly} & \rightarrow & +(\ \mathit{Poly}\ ,\ \mathit{Poly}\ ) \mid \times(\ \mathit{Poly}\ ,\ \mathit{Poly}\ ) \mid \text{Puiss}(\ \mathit{Poly}\ , \mathit{Int}) \\ & \mid & \text{Cte}(\mathit{Int}) \mid \text{X} \\ \mathit{Int} & \rightarrow & (0 \mid \cdots \mid 9)^+ \mid -(0 \mid \cdots \mid 9)^+ \end{array}$$

1. Énumérez les étiquettes abstraites de cette grammaire.

2. Donner l'expression abstraite représentant P_0.

   Correction : $\times(+(X, Cte(1)), +(\times(Cte(2), Puiss(X, 2)), Cte(-1)))$

3. Donner une grammaire non ambiguë pour le langage des polynômes, respectant les priorités. Associer à chaque règle une action qui construit la représentation abstraite associée.

   Correction :

$$\begin{array}{rcll} \mathit{Poly} & \rightarrow & \mathit{Poly}\ +\ Q & \{+(\ \mathit{Poly}_{\ 1},\ Q_{\ 1})\} \\ \mathit{Poly} & \rightarrow & Q & \{\ Q_{\ 1}\} \\ Q & \rightarrow & Q\ \times\ R & \{\times(\ Q_{\ 1},\ R_{\ 1})\} \\ Q & \rightarrow & R & \{\ R_{\ 1}\} \\ R & \rightarrow & R\ \texttt{\^{}}\ \mathit{Nat} & \{\mathit{Puiss}(\ R_{\ 1}, \mathit{val}(\mathit{Nat}_1))\} \\ R & \rightarrow & S & \{\ S_{\ 1}\} \\ S & \rightarrow & (\ \mathit{Poly}\ ) & \{\ \mathit{Poly}_{\ 1}\} \\ S & \rightarrow & \mathit{Nat} & \{\mathit{Cte}(\mathit{val}(\mathit{Nat}_1))\} \\ S & \rightarrow & -\ \mathit{Nat} & \{\mathit{Cte}(-\mathit{val}(\mathit{Nat}_1))\} \\ S & \rightarrow & \texttt{x} & \{X\} \end{array}$$

4. On désire, à l'aide de règles sémantiques, effectuer le calcul de la dérivée d'un polynôme. On considère des jugements de la forme $P \Rightarrow Q$, où P et Q sont des expressions abstraites de polynômes, signifiant « Q est la dérivée de P ». Définir les règles sémantiques nécessaires.

   Correction : *Notons que l'exposant d'un polynôme ne peut qu'être positif ou nul après la phase d'analyse. Il n'est donc pas besoin de donner les règles pour le cas d'un exposant négatif.*

$$\frac{e \Rightarrow e' \qquad f \Rightarrow f'}{+(e,f) \Rightarrow +(e',f')} \qquad \frac{e \Rightarrow e' \qquad f \Rightarrow f'}{\times(e,f) \Rightarrow +(\times(e,f'),\times(e',f))}$$

$$\frac{e \Rightarrow e'}{Puiss\ (e,n) \Rightarrow \times(\ Cte\ (n), \times(e',\ Puiss\ (e,n-1)))}\ si\ n > 1$$

$$\frac{e \Rightarrow e'}{Puiss\ (e,1) \Rightarrow e'} \qquad \frac{}{Puiss(e,0) \Rightarrow\ Cte\ (0)}$$

$$\frac{}{Cte\ (n) \Rightarrow\ Cte\ (0)} \qquad \frac{}{X\ \Rightarrow\ Cte\ (1)}$$

5. Donner la dérivation du jugement $a \Rightarrow P$ où a est l'arbre abstrait de `((x^3)+(2*x))` et P est le polynôme approprié.

   Correction :

$$\frac{\dfrac{X\ \Rightarrow Cte(1)}{\begin{array}{c}Puiss\ (\ X\ ,3) \Rightarrow \\ \times(\ Cte\ (3), \times(Cte(1),\ Puiss\ (\ X\ ,2)))\end{array}} \qquad \dfrac{\dfrac{}{Cte\ (2) \Rightarrow\ Cte\ (0)} \qquad \dfrac{}{X\ \Rightarrow Cte(1)}}{\begin{array}{c}\times(\ Cte\ (2),\ X\ ) \Rightarrow \\ +(\times(\ Cte\ (0),\ X\ ), \times(Cte(2), Cte(1)))\end{array}}}{\begin{array}{c}+(\ Puiss\ (\ X\ ,3), \times(\ Cte\ (2),\ X\ )) \Rightarrow \\ +(\times(\ Cte\ (3),\ Puiss\ (\ X\ ,2)), +(\times(\ Cte\ (0),\ X\ ), \times(Cte(2), Cte(1))))\end{array}}$$

On désire rajouter dans le langage la possibilité de définir des variables associées à des polynômes à l'aide de la construction `let ... in ...`, comme par exemple

```
let P = x^2+-3*x+1 in let Q = P^2+x in (2*x*P+Q)^3
```

Les noms de variables seront formés d'une lettre majuscule suivie éventuellement de chiffres.

1. Donner les règles supplémentaires à ajouter dans la syntaxe concrète et la syntaxe abstraite.

   Correction : *Nouvelles règles de syntaxe concrète :*

$$\begin{array}{lcll} P & \rightarrow & \texttt{let}\ Id = P\ \texttt{in}\ P & \{\ Let\ (val(Id_1),\ P_1,\ P_2)\} \\ P & \rightarrow & Poly & \{\ Poly_1\} \\ S & \rightarrow & Id & \{\ Var\ (val(Id_1))\} \end{array}$$

   Nouvelles règles de syntaxe abstraite :

$$Poly \quad \rightarrow \quad Let\ (String,\ Poly\ ,\ Poly\ )\ \mid\ Var\ (String)$$

2. Définir un nouveau système de règles sémantiques pour le calcul de la dérivée, s'appliquant sur un jugement de la forme appropriée. Attention, pour le calcul de la dérivée d'une expression polynomiale contenant un `let`, on demande de préserver le `let` : par exemple, la dérivée de `let P=x in P^2` sera `let P=x in 2*P^1*1`.

   Correction : *Attention : il faut calculer la dérivée du polynôme e figurant dans $Let\ (P,e,f)$ dans le bon environnement. Cela impose de la calculer immédiatement, et les environnements*

seront donc de la forme $\{P_1 = e_1, \ldots\}$*, où* e_1 *désigne la dérivée du polynôme valeur de* P*, et non pas le polynôme valeur de* P *lui-même.*

$$\frac{\Gamma \vdash e \Rightarrow e' \qquad \Gamma \cdot \{P = e'\} \vdash f \Rightarrow f'}{\Gamma \vdash \mathit{Let}\ (P, e, f) \Rightarrow \mathit{Let}\ (P, e, f')} \qquad \frac{}{\Gamma \vdash \mathit{Var}\ (P) \Rightarrow e}\ \textit{si } P = e \in \Gamma$$

$$\frac{\Gamma \vdash e \Rightarrow e' \qquad \Gamma \vdash f \Rightarrow f'}{\Gamma \vdash +\,(e, f) \Rightarrow +(e', f')} \qquad \frac{\Gamma \vdash e \Rightarrow e' \qquad \Gamma \vdash f \Rightarrow f'}{\Gamma \vdash \times\,(e, f) \Rightarrow +(\times(e, f'), \times(e', f))}$$

$$\frac{\Gamma \vdash e \Rightarrow e'}{\Gamma \vdash \mathit{Puiss}\ (e, n) \Rightarrow \times(\ \mathit{Cte}\ (n), \times(e',\ \mathit{Puiss}\ (e, n-1)))}\ \textit{si } n > 1$$

$$\frac{\Gamma \vdash e \Rightarrow e'}{\Gamma \vdash \mathit{Puiss}\ (e, 1) \Rightarrow e'} \qquad \frac{}{\Gamma \vdash \mathit{Puiss}(e, 0) \Rightarrow \mathit{Cte}\ (0)}$$

$$\frac{}{\Gamma \vdash \mathit{Cte}\ (n) \Rightarrow \mathit{Cte}\ (0)} \qquad \frac{}{\Gamma \vdash X \Rightarrow \mathit{Cte}\ (1)}$$

3. On suppose maintenant que les polynômes sont écrits sous la forme d'une somme de monômes, et l'on se pose les même questions que précédemment, à savoir, la définition d'une syntaxe concrète non-ambiguë, d'une syntaxe abstraite, de règles de mise sous forme d'une somme de monômes pour une somme et un produit de polynômes, et de règles de calcul de la dérivée d'un polynôme.

Chapitre 4

Fonctions à un argument

Ce chapitre s'intéresse aux définitions de fonctions dans un langage de programmation.

Pour conserver un langage simple, nous allons supposer que nous définissons des fonctions locales à une expression arithmétique grâce à une construction `let`. Définir des fonctions « globales » ne poserait pas de difficulté supplémentaire, car on peut toujours considérer qu'une fonction globale est une fonction locale à tout le programme qui la suit.

Pour éviter d'alourdir la syntaxe, nous considérerons dans ce chapitre uniquement des fonctions à un argument. Définir des fonctions à plusieurs arguments peut se faire par les mêmes méthodes, au prix d'une complication de syntaxe, ceci sera l'objet de l'exercice 4.2.

Nous commençons par le cas des fonctions non récursives, puis le cas de la récursivité sera traité spécialement.

4.1 Fonctions non récursives

La syntaxe concrète que nous adoptons pour définir des fonctions est de la forme `let fun` $f(x : t_1) : t_2 = e_1$ `in` e_2 signifiant que l'on définit une fonction de *nom* f, de *paramètre formel* x de type t_1, dont le type du résultat est t_2. e_1 et e_2 sont des expressions quelconques, e_1 est le *corps* de la fonction et e_2 est l'expression dans laquelle f est utilisée. Nous supposerons que les types sont uniquement des types simples classiques : entiers, booléens, réels, chaînes de caractères, etc. Dans ce chapitre nous prenons les entiers et les booléens pour fixer les idées.

Enfin, l'application d'une fonction f à une expression e sera notée classiquement par $f(e)$.

4.1.1 Syntaxe abstraite

La syntaxe abstraite choisie pour les fonctions à un argument est donnée par la grammaire des expressions arithmético-logiques avec variables à laquelle on rajoute

$$\begin{array}{lcl} E & \rightarrow & \mathrm{LetFun}(\mathit{String}, \mathit{String}, T, T, E, E) \mid \mathrm{App}(\mathit{String}, E) \\ T & \rightarrow & \mathrm{Int} \mid \mathrm{Bool} \mid \cdots \end{array}$$

$$\frac{\Gamma \cdot \{x : t_1\} \vdash e_1 : t_2 \qquad \Gamma \cdot \{f : t_1 \to t_2\} \vdash e_2 : t_3}{\Gamma \vdash \mathrm{LetFun}(f, x, t_1, t_2, e_1, e_2) : t_3}$$

$$\frac{\Gamma \vdash e : t_1}{\Gamma \vdash \mathrm{App}(f, e) : t_2} \text{ si } f : t_1 \to t_2 \in \Gamma$$

FIG. 4.1 – *Typage des expressions arithmétiques fonctionnelles*

La grammaire concrète non-ambiguë accompagnée des règles de calcul de la syntaxe abstraite est alors :

E	$\to$	`let`$Id = E$ `in` E	$\{\mathrm{Let}(val(Id_1), E_1, E_2)\}$
E	$\to$	`let fun` $Id(Id : T) : T = E$ `in` E	$\{\mathrm{LetFun}(\mathrm{val}(Id_1), \mathrm{val}(Id_2), \mathrm{val}(T_1), \mathrm{val}(T_2), E_1, E_2)\}$
E	$\to$	F	$\{F_1\}$
F	$\to$	$F + G$	$\{+(F_1, G_1)\}$
F	$\to$	G	$\{G_1\}$
G	$\to$	$G * H$	$\{\times(G_1, H_1)\}$
G	$\to$	H	$\{H_1\}$
H	$\to$	Nat	$\{\mathrm{Cte}(val(Nat_1))\}$
H	$\to$	Id	$\{\mathrm{Var}(val(Id_1))\}$
H	$\to$	$Id(E)$	$\{\mathrm{App}(val(Id_1), E_1)\}$
H	$\to$	(E)	$\{E_1\}$
T	$\to$	`int`	$\{\mathrm{Int}\}$
T	$\to$	`bool`	$\{\mathrm{Bool}\}$

Par exemple, à l'expression concrète `let fun` $sqr(x$ `:int`$)$ `:int` $= x \times x$ `in` $sqr(3) + sqr(4)$ correspond la syntaxe abstraite

$$\mathrm{LetFun}(sqr, x, \mathrm{Int}, \mathrm{Int}, \times(\mathrm{Var}(x), \mathrm{Var}(x)), +(\mathrm{App}(sqr, \mathrm{Cte}(3)), \mathrm{App}(sqr, \mathrm{Cte}(4))))$$

4.1.2 Typage

Nos jugements de typage s'écriront encore $\Gamma \vdash e : t$, où e dénote une expression de syntaxe abstraite. Mais ici, l'ensemble des types est plus riche : nous aurons les types des fonctions des entiers ou les booléens vers les entiers ou les booléens. En effet, une fonction des entiers naturels dans les entiers naturels est un objet très différent des identificateurs de type entier. En particulier, un identificateur de fonction doit toujours être utilisé avec un argument, de type entier dans le cas qui nous intéresse. On dit qu'un identificateur de fonction a un *type fonctionnel*, que l'on note $t_1 \to t_2$ si l'argument est de type t_1 et le résultat est de type t_2.

Les règles de typage sont décrites sur la figure 4.1. Les anciennes règles de la figure 3.6 sont encore valables et ne sont pas répétées.

Notons à nouveau la règle de portée : dans la définition `let fun` $f(x : t_1) : t_2 = e_1$ `in` e_2, le nom x du paramètre formel peut être utilisé dans e_1 mais pas dans e_2, alors que le nom f de la fonction peut être utilisé dans e_2 mais pas dans e_1, il ne peut donc pas s'agir d'une fonction récursive, cas que nous allons traiter un peu plus loin.

La dérivation de typage de l'exemple de *sqr* ci-dessus est donné sur la figure 4.2.

4.1.3 Évaluation

Ces expressions arithmétiques fonctionnelles s'évaluent de manière similaire aux expressions arithmétiques usuelles, mais il va falloir cette fois un environnement liant le nom

$$(1): \quad \frac{\dfrac{\overline{\Gamma \vdash \mathrm{Cte}(3) : \mathrm{Int}}}{\Gamma \vdash \mathrm{App}(sqr, \mathrm{Cte}(3)) : \mathrm{Int}} \quad \dfrac{\overline{\Gamma \vdash \mathrm{Cte}(4) : \mathrm{Int}}}{\Gamma \vdash \mathrm{App}(sqr, \mathrm{Cte}(4)) : \mathrm{Int}}}{\Gamma \vdash \times(\mathrm{App}(sqr, \mathrm{Cte}(3)), \mathrm{App}(sqr, \mathrm{Cte}(4))) : \mathrm{Int}}$$

$$\frac{\dfrac{\overline{\{x : \mathrm{Int}\} \vdash \mathrm{Var}(x) : \mathrm{Int}} \quad \overline{\{x : \mathrm{Int}\} \vdash \mathrm{Var}(x) : \mathrm{Int}}}{\{x : \mathrm{Int}\} \vdash \times(\mathrm{Var}(x), \mathrm{Var}(x)) : \mathrm{Int}} \quad (1)}{\{\} \vdash \mathrm{LetFun}(sqr, x, \times(\mathrm{Var}(x), \mathrm{Var}(x)), +(\mathrm{App}(sqr, \mathrm{Cte}(3)), \mathrm{App}(sqr, \mathrm{Cte}(4)))) : \mathrm{Int}}$$

FIG. 4.2 – *Typage de l'expression arithmétique fonctionnelle* **`let fun`** $sqr(x) = x \times x$ **`in`** $sqr(3) + sqr(4)$ *où* $\Gamma = \{sqr : Int \to Int\}$

$$\frac{\Gamma \cdot \{f = (x, e_1)\} \vdash e_2 \Rightarrow v}{\Gamma \vdash \mathrm{LetFun}(f, x, e_1, e_2) \Rightarrow v}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma \cdot \{x = v_1\} \vdash e_2 \Rightarrow v_2}{\Gamma \vdash \mathrm{App}(f, e_1) \Rightarrow v_2} \text{ si } f = (x, e_2) \in \Gamma$$

FIG. 4.3 – *Évaluation des expressions arithmétiques fonctionnelles*

d'une fonction à son corps, c'est-à-dire à une expression arithmétique fonctionnelle. De plus, le corps dépend du paramètre formel de la fonction, paramètre qui sera lié lors de l'appel à la fonction. Il va donc falloir stocker dans l'environnement de liaison aussi bien le corps que le nom du paramètre formel de la fonction. On utilisera pour cela la notation $f = (x, e)$, où f est le nom de la fonction, x est le nom de son paramètre formel, et e est son corps. Sur l'exemple précédent l'environnement d'évaluation où sqr est définie est noté $\{sqr = (x, \times(\mathrm{Var}(x), \mathrm{Var}(x)))\}$.

Le domaine de valeur des jugements d'évaluation sera donc $D^v = \mathbb{N} \cup \{T, F\}$ et le domaine de contexte D^c sera l'ensemble des listes de couples (identificateur,valeur) où les valeurs sont soit des entiers ou des booléens pour les identificateurs de variables, soit des couples (identificateur,expression abstraite) pour les identificateurs de fonctions.

Les règles d'évaluation sont données sur la figure 4.3, les anciennes règles de la figure 3.7 étant encore valables.

Théorème 4.1 *Le langage des expressions arithmétiques fonctionnelles, avec la sémantique définie ci-dessus,* n'est pas fortement typé. *Voir exercice 4.4.*

La dérivation d'évaluation de notre exemple est traité sur la figure 4.4, où sur cette figure Γ dénote l'environnement $\{sqr = (x, \times(\mathrm{Var}(x), \mathrm{Var}(x)))\}$. D'autres exemples instructifs de typage et d'évaluation sont proposés en exercice. Notons que les calculs d'évaluation terminent toujours pour tous les langages définis jusqu'à présent.

4.2 Fonctions récursives

Nous avons traité dans le paragraphe précédent des fonctions non récursives, nous allons maintenant considérer le cas plus complexe des fonctions récursives.

Dans la syntaxe concrète nous choisissons de rajouter le mot clé `rec` pour spécifier que l'on définit une fonction récursive, et dans la syntaxe abstraite nous introduisons une construction LetFunRec analogue au LetFun.

$$
\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathrm{Cte}(3) \Rightarrow 3} \quad \dfrac{\text{trivial...}}{\Gamma \cdot \{x = 3\} \vdash \times(\mathrm{Var}(x), \mathrm{Var}(x)) \Rightarrow 9}}{\Gamma \vdash \mathrm{App}(sqr, \mathrm{Cte}(3)) \Rightarrow 9} \quad \dfrac{\dfrac{}{\Gamma \vdash \mathrm{Cte}(4) \Rightarrow 4} \quad \dfrac{\text{trivial...}}{\Gamma \cdot \{x = 4\} \vdash \times(\mathrm{Var}(x), \mathrm{Var}(x)) \Rightarrow 16}}{\Gamma \vdash \mathrm{App}(sqr, \mathrm{Cte}(4)) \Rightarrow 16}}{\dfrac{\dfrac{(1)}{\Gamma \vdash \mathrm{App}(sqr, \mathrm{Cte}(3)) \Rightarrow 9} \quad \dfrac{(2)}{\Gamma \vdash \mathrm{App}(sqr, \mathrm{Cte}(4)) \Rightarrow 16}}{\Gamma \vdash +(\mathrm{App}(sqr, \mathrm{Cte}(3)), \mathrm{App}(sqr, \mathrm{Cte}(4))) \Rightarrow 25}}{\{\} \vdash \mathrm{LetFun}(sqr, n, \times(\mathrm{Var}(x), \mathrm{Var}(x)), +(\mathrm{App}(sqr, \mathrm{Cte}(3)), \mathrm{App}(sqr, \mathrm{Cte}(4)))) \Rightarrow 25}
$$

(1) :

(2) :

FIG. 4.4 – *Évaluation de l'expression arithmétique fonctionnelle* **let fun** $sqr(x) = x \times x$ **in** $sqr(3) + sqr(4)$ *où* $\Gamma = \{sqr = (x, \times(Var(x), Var(x)))\}$.

$$
\frac{\Gamma \cdot \{f : t_1 \to t_2, x : t_1\} \vdash e_1 : t_2 \qquad \Gamma \cdot \{f : t_1 \to t_2\} \vdash e_2 : t}{\Gamma \vdash \mathrm{LetFunRec}(f, n, t_1, t_2, e_1, e_2) : t}
$$

FIG. 4.5 – *Typage des fonctions récursives*

Nous allons considérer l'exemple fondamental de la fonction factorielle définie dans notre syntaxe par

$$\texttt{let fun rec } f(x : \texttt{int}) : \texttt{int} = \texttt{if } x = 0 \texttt{ then } 1 \texttt{ else } x \times f(x-1) \texttt{ in } f(3)$$

4.2.1 Typage

Les règles de typage sont très similaires à celles des paragraphes précédents. La différence essentielle concerne la portée du nom de la fonction récursive définie par `let fun rec`. Le nom de la fonction peut en effet intervenir dans le corps de la fonction, et il faut donc ajouter le type de la fonction à l'environnement de typage avant d'en typer le corps. Cela suppose bien sûr que ce type est connu, ce qui est le cas dans le cadre que nous avons choisi [1]. Nous nous contentons de donner à la figure 4.5 la nouvelle règle de typage des expressions `let fun rec`. L'exemple donné au paragraphe précédent montre un cas où le résultat est du même type que la fonction. L'exemple ci-dessous donne un exemple de fonction qui a le type `int -> int` et pourtant le type de l'expression `let fun rec` est `bool` :

$$\texttt{let fun rec } f(x : \texttt{int}) : \texttt{int} = \texttt{if } x = 0 \texttt{ then } 1 \texttt{ else } x \times f(x-1) \texttt{ in } f(10) > 1000$$

Le typage de cette expression est laissé au lecteur !

4.2.2 Évaluation

Les règles d'évaluation sont inchangées pour les expressions LetFunRec par rapport aux expressions LetFun. En effet, d'une part l'information supplémentaire concernant le type de l'argument d'une fonction récursive n'est d'aucune utilité à l'évaluation ; d'autre part, la notion d'environnement que nous avons utilisée est suffisamment générale pour couvrir

1. l'inférence de types comme dans CAML est hors du cadre de ce cours.

le cas des fonctions récursives. On remarquera néanmoins l'importance de la gestion des environnements en pile, qui permet de récupérer à chaque fois la bonne valeur du paramètre formel de la fonction récursive. Ceci est mis en évidence sur l'évaluation de l'expression calculant la factorielle de 3, qui est donnée sur les figures 4.6 et 4.7.

4.3 Exercices

Exercice 4.1
Analyser, typer, puis évaluer si possible les expressions suivantes.

1. `let fun` $sqr(x : \texttt{int}) : \texttt{int} = x * x$ `in` $sqr(1) + sqr(sqr(2))$.
2. `let fun` $sqr(x : \texttt{int}) : \texttt{int} = x * x$ `in let` $x = sqr(1) + sqr(sqr(2))$ `in` $sqr(x)$.
3. `let fun` $f(x : \texttt{int}) : \texttt{int} = x * f(x)$ `in` $f(1)$.
4. `let fun` $f(x : \texttt{int}) : \texttt{int} = x * x$ `in` $f(x + 1)$.

Exercice 4.2
On veut généraliser l'étude de ce chapitre aux fonctions à plusieurs arguments. Pour ce faire, on modifie la règle de la syntaxe abstraite pour le LetFun de la manière suivante :

$$\begin{array}{rcl} E & \rightarrow & \text{LetFun}(\mathit{String}, \mathit{Args}, T, E, E) \\ \mathit{Args} & \rightarrow & \text{Argvide} \mid \text{Argcons}(\mathit{String}, T, \mathit{Args}) \end{array}$$

1. Donner la nouvelle grammaire concrète non-ambiguë accompagnée des calculs de syntaxe abstraite.
2. Donner les nouvelles règles de typage.
3. Donner les nouvelles règles d'évaluation.
4. Donner les dérivations de typage et d'évaluation du programme `let fun` $f(x : \texttt{int}, y : \texttt{int}) : \texttt{int} = x * x + y * y$ `in` $f(3, 4)$

Exercice 4.3
On veut généraliser l'étude de ce chapitre aux fonctions mutuellement récursives. Il faut pour cela être capable de définir simultanément deux fonctions, et les constructions `let fun` doivent donc être capable de traiter une liste quelconque de définitions. Nous choisissons alors la syntaxe abstraite suivante :

$$\begin{array}{rcl} E & \rightarrow & \text{LetFunRec}(\mathit{Defs}, E) \\ \mathit{Defs} & \rightarrow & \text{Defvide} \mid \text{Defcons}(\mathit{String}, \mathit{String}, T, T, E, \mathit{Defs}) \end{array}$$

1. Donner la nouvelle grammaire concrète non-ambiguë accompagnée des calculs de syntaxe abstraite.
2. Donner les nouvelles règles de typage.
3. Donner les nouvelles règles d'évaluation.
4. Donner les dérivations de typage et d'évaluation du programme

$$\begin{array}{l} \texttt{let fun rec}\, f(x : \texttt{int}) : \texttt{bool} = \texttt{if}\ x = 0\ \texttt{then true else}\ g(x - 1) \\ \texttt{and also}\ g(x : \texttt{int}) : \texttt{bool} = \texttt{if}\ x = 0\ \texttt{then false else}\ f(x - 1) \\ \texttt{in}\ f(3) \end{array}$$

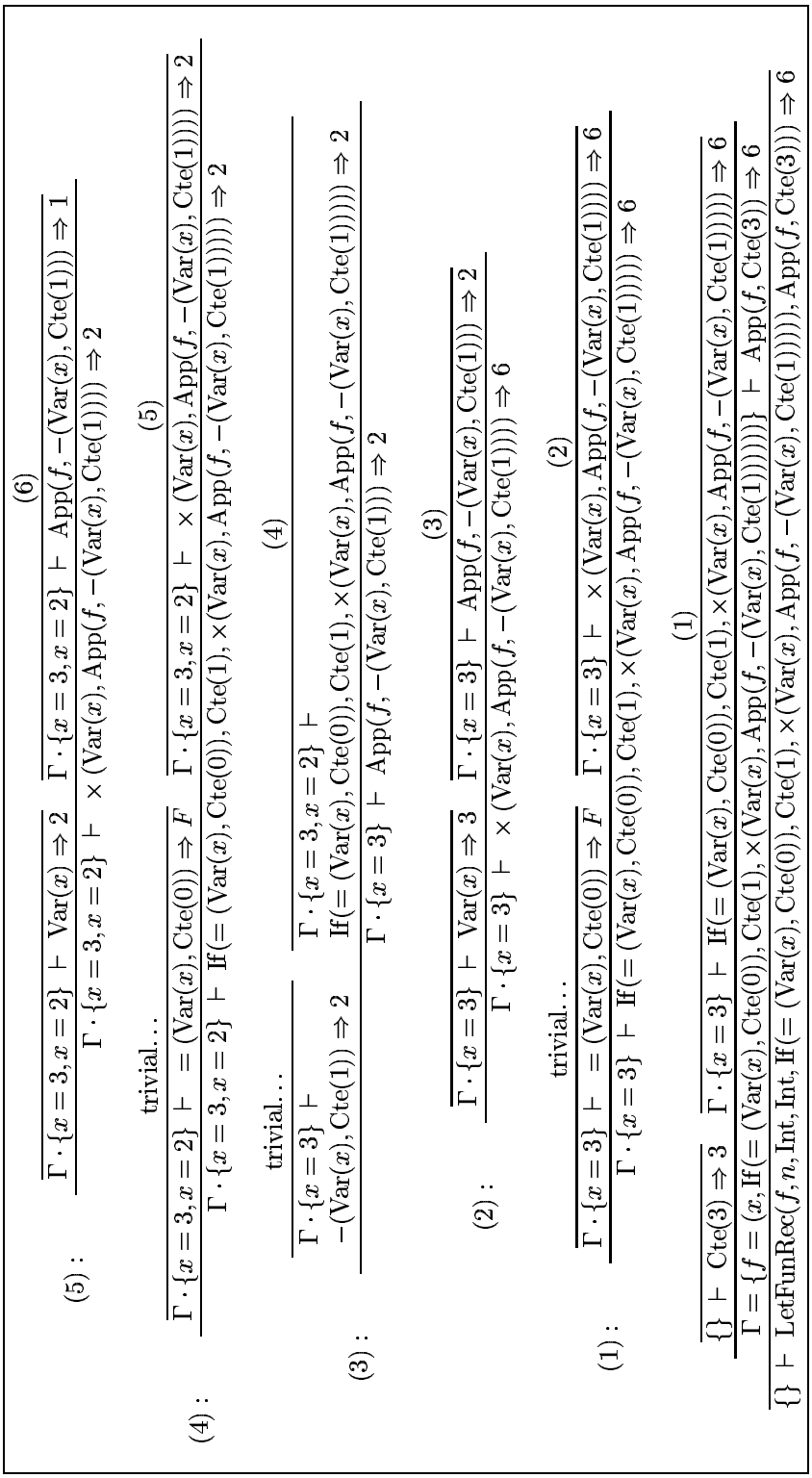


FIG. 4.6 – *Évaluation de l'expression fonctionnelle récursive* `let fun rec` $f(x$: `int`$)$: `int` $=$ `if` $x = 0$ `then` 1 `else` $x \times f(x-1)$ `in` $f(3)$, *partie 1*

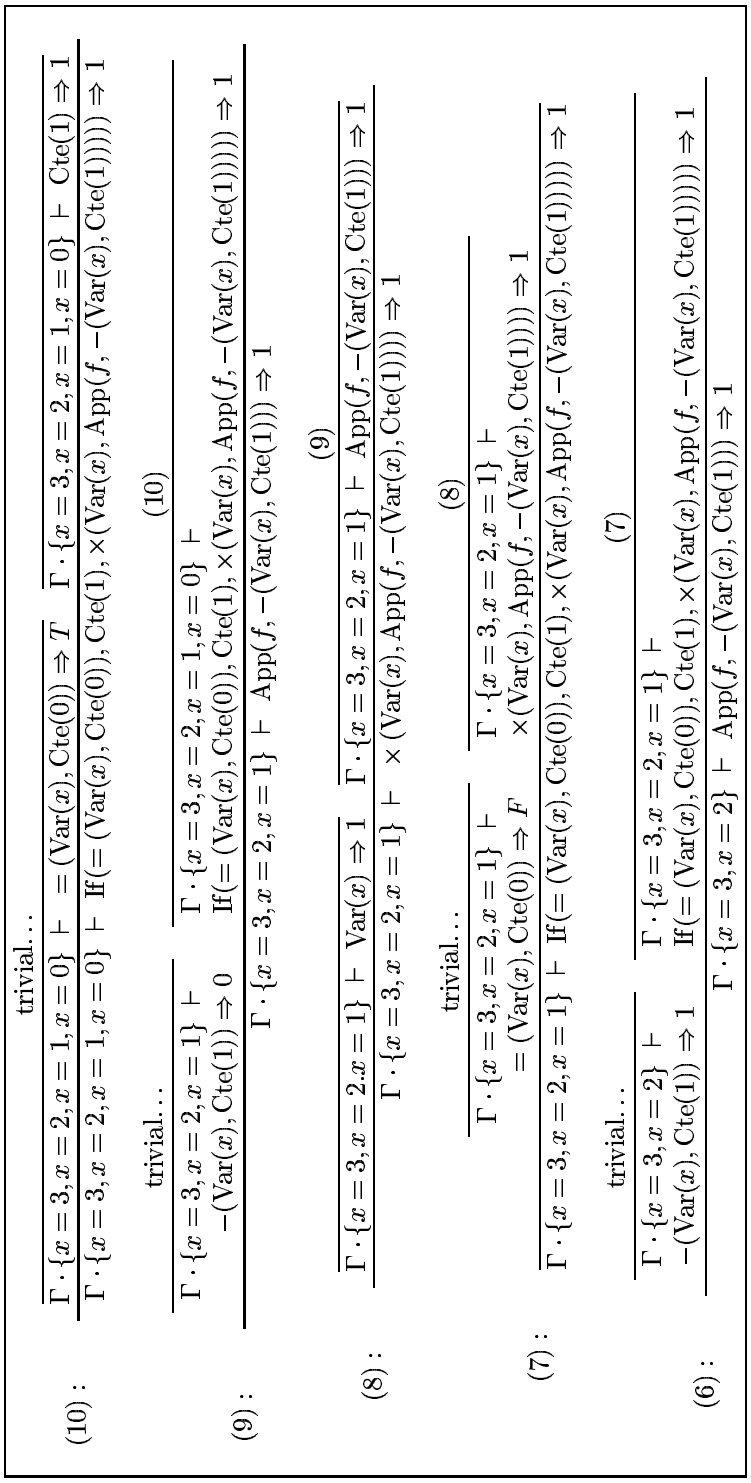


FIG. 4.7 – *Évaluation de l'expression fonctionnelle récursive* `let fun rec` $f(x$: `int`$)$: `int` $=$ `if` $x = 0$ `then` 1 `else` $x \times f(x-1)$ `in` $f(3)$, *partie 2*

Exercice 4.4
La sémantique définie par les règles données dans ce chapitre est la sémantique par *liaison dynamique*. Les règles de sémantique pour la *liaison statique* sont :

$$\frac{\Gamma \cdot \{f = (x, e_1, \Gamma)\} \vdash e_2 \Rightarrow v}{\Gamma \vdash \mathrm{LetFun}(f, x, e_1, e_2) \Rightarrow v}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma' \cdot \{x = v_1\} \vdash e_2 \Rightarrow v_2}{\Gamma \vdash \mathrm{App}(f, e_1) \Rightarrow v_2} \text{ si } f = (x, e_2, \Gamma') \in \Gamma$$

1. Évaluer le programme suivant pour chacune des deux sémantiques :

```
let x=1 in
let fun f(y:int):int = x+y in
let x=2 in f(3)
```

2. Commenter les résultats obtenus.

Chapitre 5

Sémantique d'un langage fonctionnel

Notre traitement des expressions arithmétiques permettait des définitions de fonctions à un argument. Nous pourrions étendre ce traitement de manière à traiter des fonctions à n arguments. Il nous faudrait pour cela ajouter deux nouvelles règles de typage, et deux nouvelles règles sémantiques pour chaque valeur de n. Nous ne pourrions donc traiter qu'un nombre fini de valeurs particulières de n, sauf à utiliser des règles paramétrées par n. De plus, nos fonctions seraient à valeur dans un type de base, entier ou réel.

De fait, ce qui caractérise les langages dits *fonctionnels* est que les fonctions sont des valeurs au même titre que les entiers, les réels ou les booléens. En particulier, les arguments d'une fonction peuvent être des fonctions, et le résultat peut lui aussi être une fonction.

En fait, ce qui distinguera une fonction d'une valeur de base comme un entier ou un booléen sera son type : on dira qu'une fonction est une expression de type *fonctionnel*, un type fonctionnel étant un type de la forme $t_1 \to t_2$ désignant le type des fonctions de t_1 vers t_2. L'ensemble des types de notre langage sera décrit par la grammaire de types suivante :

$$T \to int \mid bool \mid T\texttt{->}T$$

Cette grammaire est ambiguë car un type de la forme $t_1 \to t_2 \to t_3$ peut être reconnu sous les deux formes différentes $(t_1 \to t_2) \to t_3$ et $t_1 \to (t_2 \to t_3)$, or ces deux types sont tout à fait différents puisque le premier dénote les fonctions dont l'argument est une fonction de t_1 dans t_2, alors que le second dénote les fonctions dont le résultat est une fonction de t_2 dans t_3. Traditionnellement, on considère que la flèche associe à droite, c'est-à-dire que la seconde forme $t_1 \to (t_2 \to t_3)$ est la forme par défaut. Pour spécifier une fonction dont le type est $(t_1 \to t_2) \to t_3$, on utilisera des parenthèses. La grammaire non-ambiguë des types de notre langage est donc

$$\begin{aligned} T &\to T'\texttt{->}T \mid T' \\ T' &\to int \mid bool \mid \texttt{(}T\texttt{)} \end{aligned}$$

Dans ce chapitre, nous allons dans un premier temps décrire deux primitives à la base de tous les calculs fonctionnels, l'abstraction et l'application. Puis nous verrons comment coder les fonctions à n arguments grâce à ces nouvelles primitives, ainsi que les instructions *let* déja vues.

5.1 Syntaxes concrète et abstraite

La fonction qui à la variable x associe le calcul décrit par l'expression e est appelée une *abstraction* et est notée (il s'agit de la syntaxe concrète) `fun` $x : t$ `->` e. Cela correspond à la notation mathématique $x \mapsto f(x)$, et permet de définir une fonction qui n'a pas de nom. L'expression e est appelée le *corps* de la fonction.

L'appel fonctionnel sera maintenant remplacé par l'application d'une abstraction `fun` $x : t$ `->` e_1 à un argument e_2. Cette application sera notée (en syntaxe concrète) (`fun` $x : t$ `->` e_1) e_2. Plus généralement, si e_1 et e_2 sont deux expressions, on notera l'application de e_1 à e_2 par $(e_1\ e_2)$, les parenthèses pouvant être omises s'il n'y a aucune ambiguïté. La syntaxe concrète de nos expressions est donc décrite par la grammaire YACC non-ambiguë ci-dessous, qui calcule en même temps la syntaxe abstraite des expressions :

E	$\rightarrow$	`fun` $Id : T$ `->` E	$\{\text{Fun}(\text{val}(Id_1), T_1, E_1)\}$
E	$\rightarrow$	`let`Id `=` E `in` E	$\{\text{Let}(\text{val}(Id_1), E_1, E_1)\}$
E	$\rightarrow$	`if` E `then` E `else` E	$\{\text{If}(E_1, E_2, E_1)\}$
E	$\rightarrow$	F	$\{F_1\}$
F	$\rightarrow$	F `or` G	$\{\text{Or}(F_1, G_1)\}$
F	$\rightarrow$	G	$\{G_1\}$
G	$\rightarrow$	G `and` H	$\{\text{And}(G_1, H_1)\}$
G	$\rightarrow$	H	$\{H_1\}$
H	$\rightarrow$	`not` H	$\{\text{Not}(H_1)\}$
H	$\rightarrow$	I	$\{I_1\}$
I	$\rightarrow$	$I = J$	$\{=(I_1, J_1)\}$
I	$\rightarrow$	$I > J$	$\{>(I_1, J_1)\}$
I	$\rightarrow$	J	$\{J_1\}$
J	$\rightarrow$	$J + K$	$\{+(J_1, K_1)\}$
J	$\rightarrow$	K	$\{K_1\}$
K	$\rightarrow$	$K * L$	$\{\times(K_1, L_1)\}$
K	$\rightarrow$	L	$\{L_1\}$
L	$\rightarrow$	`true`	$\{\text{True}\}$
L	$\rightarrow$	`false`	$\{\text{False}\}$
L	$\rightarrow$	Nat	$\{\text{Cte}(\text{val}(Nat_1))\}$
L	$\rightarrow$	Id	$\{\text{Var}(\text{val}(Id_1))\}$
L	$\rightarrow$	(E)	$\{E_1\}$
L	$\rightarrow$	$(E\ E)$	$\{\text{App}(E_1, E_2)\}$
T	$\rightarrow$	T' `->` T	$\{\text{Fleche}(T_1, T_2)\}$
T'	$\rightarrow$	(T)	$\{T_1\}$
T'	$\rightarrow$	`int`	$\{\text{Int}\}$
T'	$\rightarrow$	`bool`	$\{\text{Bool}\}$

5.2 Typage

Les expressions de syntaxe abstraites contiennent des informations qui n'ont pour but que de faciliter la phase de contrôle de type. Comme elles ne sont d'aucune utilité pour la phase d'évaluation, on va pouvoir nettoyer les expressions de syntaxe abstraite lors de la phase de typage, de façon à obtenir de nouvelles expressions de syntaxe abstraite plus simples. Pour cela, nous utiliserons une syntaxe des jugements analogue à celle du paragraphe 2.4.2. Les règles de typage de notre langage fonctionnel sont données à la figure 5.1.

$$\frac{\Gamma \cdot \{x : t_1\} \vdash e_1 : t_2 \Rightarrow e_2}{\Gamma \vdash \mathrm{Fun}(x, t_1, e_1) : \mathrm{Fleche}(t_1, t_2) \Rightarrow \mathrm{Fun}(x, e_2)}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Fleche}(t_1, t_2) \Rightarrow e_1' \qquad \Gamma \vdash e_2 : t_1 \Rightarrow e_2'}{\Gamma \vdash \mathrm{App}(e_1, e_2) : t_2 \Rightarrow \mathrm{App}(e_1', e_2')}$$

FIG. 5.1 – *Typage des expressions fonctionnelles*

$$\frac{}{\Gamma \vdash \mathrm{Fun}(x, e) \Rightarrow (x, e)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow (x, e) \qquad \Gamma \vdash e_2 \Rightarrow v_2 \qquad \Gamma \cdot \{x = v_2\} \vdash e \Rightarrow v_3}{\Gamma \vdash \mathrm{App}(e_1, e_2) \Rightarrow v_3}$$

FIG. 5.2 – *Évaluation des expressions fonctionnelles*

5.3 Évaluation

Les expressions fonctionnelles sont maintenant des expressions comme les autres, elles doivent donc avoir une valeur. En première approximation, la valeur sera un couple formé par le nom de la variable et l'expression définissant cette fonction. Il s'agit tout simplement du type d'information stocké dans l'environnement dans le cas de l'évaluation des fonctions à un argument, voir figure 4.3.

Les règles d'évaluation sont alors celles de la figure 5.2.

Par exemple, l'expression suivante

```
let succ = fun x:int -> x+1 in (succ 2)
```

conduit à évaluer d'abord l'expression $\mathrm{App}(\mathrm{Var}(succ), \mathrm{Cte}(2))$ dans l'environnement $\Gamma_1 = \{succ = (x, +(\mathrm{Var}(x), \mathrm{Cte}(1)))\}$, puis à évaluer dans l'environnement $\Gamma_1 = \Gamma_1 \cdot \{x = 2\}$ l'expression $+(\mathrm{Var}(x), \mathrm{Cte}(1))$ qui donne bien le résultat escompté.

Un exemple de fonction avec fonction en argument est donné par

```
let eval0 = fun f:(int->int) -> (f 0)
in (eval0 (fun x:int -> x+1))
```

qui conduira à évaluer dans l'environnement $\Gamma = \{f = (x, +(\mathrm{Var}(x), \mathrm{Cte}(1)))\}$ l'expression $\mathrm{App}(\mathrm{Var}(f), \mathrm{Cte}(0))$ qui retournera à son tour la valeur escomptée 1.

5.4 Fonctions à plusieurs arguments

Une fonction étant une expression parmi d'autres, une fonction à deux arguments est une abstraction `fun` x `->` e_1 dont le corps est une fonction à un argument $e_1 =$ `fun` y `->` e_2. La fonction s'écrit donc `fun` x `->` `fun` y `->` e_2. On peut bien sûr tout aussi bien définir des fonctions d'un nombre quelconque d'arguments. Afin de faciliter la lecture des fonctions de multiples arguments, on a l'habitude de grouper les variables sous les `fun` successifs, avec l'écriture `fun` $x_1 \cdots x_n$ `->` e, qui est par définition une abbréviation de l'écriture `fun` x_1 `->` `fun` x_2 `->` $\cdots$`fun` x_n `->` e. Cette convention d'écriture cache donc le fait qu'une fonction de deux arguments x et y et de corps e n'est rien d'autre qu'une fonction de l'unique argument x dont le corps est la fonction `fun` y `->` e. Grâce à notre nouvelle notation, les fonctions de n arguments sont donc devenues des fonctions de un argument.

$$\frac{\Gamma \vdash e \Rightarrow e' \qquad \Gamma \vdash g \Rightarrow g'}{\Gamma \vdash \mathrm{Let}(n, e, g) \Rightarrow \mathrm{App}(\mathrm{Fun}(n, g'), e')}$$

$$\frac{\Gamma \vdash e \Rightarrow e' \qquad \Gamma \vdash g \Rightarrow g'}{\Gamma \vdash \mathrm{LetFun}(\mathrm{Var}(f), \mathrm{Var}(n), e, g) \Rightarrow \mathrm{App}(\mathrm{Fun}(f, g'), \mathrm{Fun}(n, e'))}$$

FIG. 5.3 – *Codages des expressions* `let`

5.5 Problème de la liaison dynamique

Le problème de la liaison dynamique entrevu dans l'exercice 4.4 se présente ici de manière cruciale. Par exemple considérons l'expression

```
let f = fun x:int -> fun y:int -> 2*x+3*y
in ((f 2) 3)
```

Avec les règles ci-dessus, l'évaluation de `(f 2)` va demander d'évaluer l'expression `fun y:int -> 2*x+3*y` dans l'environnement $\{\mathtt{x} = 2\}$ qui donne le résultat $(\mathtt{y}, \mathtt{2*x+3*y})$. Mais alors l'évaluation de `((f 2) 3)` va demander d'évaluer l'expression `2*x+3*y` dans l'environnement $\{\mathtt{y} = 3\}$ qui va échouer car `x` n'y a pas de valeur. Pour évaluer correctement l'expression, il est nécessaire de procéder comme dans l'exercice 4.4, en liant la fonction à l'environnement dans lequel elle est définie. La valeur d'une expression de type fonctionnel sera donc un triplet formé du nom de la variable, de l'environnement de définition de la fonction et de son corps.

Les règles d'évaluation correspondantes, dites avec *liaison statique*, sont :

$$\frac{}{\Gamma \vdash \mathrm{Fun}(x, e) \Rightarrow (x, \Gamma, e)}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow (x, \Gamma', e) \qquad \Gamma \vdash e_2 \Rightarrow v_2 \qquad \Gamma' \cdot \{x = v_2\} \vdash e \Rightarrow v_3}{\Gamma \vdash \mathrm{App}(e_1, e_2) \Rightarrow v_3}$$

Dans l'exemple ci-dessus, l'évaluation de `(f 2)` retourne $(\mathtt{y}, \{\mathtt{x} = 2\}, \mathtt{2*x+3*y})$ et donc l'évaluation de `((f 2) 3)` s'effectuera correctement.

Théorème 5.1 *Le langage des expressions arithmétiques fonctionnelles, avec la sémantique par liaison statique définie ci-dessus, est fortement typé.*

5.6 Codage des expressions `let`

Montrons maintenant le codage des expressions `let` étudiées au chapitre 4 que l'on trouve dans les langages fonctionnels habituels. Dans la mesure où ce codage exprime la signification des différentes expressions `let` en fonction des primitives d'abstraction et d'application, nous pouvons le décrire avec les règles sémantiques données figure 5.3.

La seconde règle se ramène en fait à la première, en considérant qu'une fonction d'un argument est un identificateur dont le corps est une abstraction. Ces règles peuvent surprendre, puisque c'est le corps de la fonction qui est passé en argument de l'expression g qui figure après le mot clé $\in$ dans la syntaxe concrète. Mais cela est assez naturel, puisque l'expression g utilise la fonction définie par le let, qui va donc être tout normalement substituée par son corps ...

La traduction que nous venons de présenter pour les `let fun` ne fonctionne que pour les définitions non récursives, une autre traduction est nécessaire pour les fonctions récursives, donc pour les expressions `let fun rec` . Cela explique la présence de deux mots clé

distincts, `let fun` et `let fun rec` pour définir d'une part les fonctions non récursives et d'autre part les fonctions récursives dans les langages de la famille ML.

Chapitre 6

Types structurés

Ce chapitre s'intéresse aux problèmes de sémantiques liés à l'utilisation de types appelés *structurés*, qui désignent d'ordinaire des types non fonctionnels obtenus à partir d'autres types, en particulier des types de base, à l'aide de constructeurs de types.

On classe les types structurés en deux catégories : les types produits et les types sommes. Les types produits correspondent à la construction d'un type structuré par produit cartésien de types déjà définis, un élément d'un type produit sera donc défini comme un n-uplets d'éléments ; alors que les types sommes correspondent à la construction d'un type structuré par union disjointe, un élément d'un type somme sera donc défini par une valeur parmi l'union des éléments possibles.

6.1 Les types produits

Nous nous intéressons d'abord aux types produits. Nous les classons en deux classes : les produits anonymes et les produits étiquetés. Les produits anonymes correspondent à la définition classique des n-uplets en mathématiques, alors que les produits étiquetés correspondent à la structure d'enregistrement que l'on trouve dans la plupart des langages de programmation.

Pour simplifier l'étude nous considérons les types couples, la généralisation aux n-uplets avec $n \geq 3$ étant immédiate.

6.1.1 Le type couple

On notera par $t \times t'$ le type des couples d'éléments de types respectifs t et t'. La grammaire abstraite des types est donc la suivante :

$$T \ \rightarrow \ \times (T,T) \ \mid \ \text{->} \ (T,T) \ \mid \ Nat \ \mid \ Bool$$

Syntaxe

Donnons tout d'abord la syntaxe abstraite des expressions de type couple :

$$E \ \rightarrow \ Couple(E,E) \ \mid \ Fst(E) \ \mid \ Snd(E) \ \mid \ \ldots$$

puis une syntaxe concrête avec les règles de calcul de la syntaxe abstraite :

$$\begin{array}{lcll} E & \rightarrow & (E,E) & \{Couple(E,E)\} \\ E & \rightarrow & fst(E) & \{Fst(E)\} \\ E & \rightarrow & snd(E) & \{Snd(E)\} \\ E & \rightarrow & \ldots & \{\ldots\} \end{array}$$

Les points de suspension figurent les règles étudiées au cours des chapitres antérieurs, qui peuvent toutes être récupérées telles quelles.

Typage

Nous aurons besoin de trois nouvelles règles de typage, une pour chaque nouvelle forme d'expression abstraite :

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash Couple(e_1, e_2) : \times(t_1, t_2}$$

$$\frac{\Gamma \vdash e : t_1 \times t_2}{Fst(e) : t_1}$$

$$\frac{\Gamma \vdash e : t_1 \times t_2}{Snd(e) : t_2}$$

Évaluation

Nous avons de même trois nouvelles règles d'évaluation :

$$\frac{\Gamma \vdash e_1 \Rightarrow v_1 \qquad \Gamma \vdash e_2 \Rightarrow v_2}{\Gamma \vdash Couple(e_1, e_2) \Rightarrow (v_1, v_2)}$$

$$\frac{\Gamma \vdash e \Rightarrow (v_1, v2)}{Fst(e) \Rightarrow v_1}$$

$$\frac{\Gamma \vdash e \Rightarrow (v_1, v_2)}{Snd(e) \Rightarrow v_2}$$

6.1.2 Les types enregistrements

Le type enregistrement permet de structurer des données de types variés que l'on peut ensuite retrouver via une clé d'accès. La grammaire des types devient la suivante:

$$\begin{array}{rcl} T & \rightarrow & TypeRecord(TR) \mid NomType(String) \\ TR & \rightarrow & TRvide \mid TRcons(String, T, TR) \end{array}$$

Par exemple, au type enregistrement (écrit en syntaxe concrète) `{x:int,y:int}` correspond la syntaxe abstraite $TRcons(\text{"}x\text{"}, \text{Int}, TRcons(\text{"}y\text{"}, \text{Int}, TRvide))$.

Donnons maintenant la syntaxe abstraite des expressions de type enregistrement :

$$\begin{array}{rcl} E & \rightarrow & ExprRecord(ER) \mid AccesRecord(E, String) \\ ER & \rightarrow & ERvide \mid TRcons(String, E, ER) \end{array}$$

Par exemple, à l'expression (écrite en syntaxe concrète) `{x=1,y=2}` correspond la syntaxe abstraite $ERcons(\text{"}x\text{"}, \text{Cte}(1), ERcons(\text{"}y\text{"}, \text{Cte}(2), ERvide))$

Pour des raisons qui seront explicitées dans le reste de ce paragraphe, un programme pourra posséder des déclarations de types. Nos programmes seront définis par la syntaxe concrète suivante, accompagnée de règles de calcul de la syntaxe abstraite :

$$
\begin{array}{rcll}
E & \to & \texttt{type}\ Id = T\ \texttt{in}\ E & \{Typedecl(Id, T, E)\} \\
E & \to & E.Id & \{AccesRecord(E, \mathrm{val}(Id_1))\} \\
E & \to & \{Id = E; ER & \{ExprREcord(ERcons(\mathrm{val}(Id_1), E_1, ER_1))\} \\
ER & \to & \} & \{ERvide\} \\
ER & \to & Id = E, ER & \{ERcons(\mathrm{val}(Id_1), E_1, ER_1)\} \\
T & \to & recordId : T, TR & \{TypeRecord(TRcons(\mathrm{val}(Id_1), T_1, TR_1))\} \\
T & \to & Id & \{Nomtype(\mathrm{val}(Id))\} \\
TR & \to & endrecord & \{TRvide\} \\
TR & \to & Id : T; TR & \{TRcons(\mathrm{val}(Id_1), T_1, TR)\}
\end{array}
$$

Ainsi, au programme

```
type point = record
               x:int ;
               y:int
             endrecord
in
let A : point = { x=3 ; y=4 }
in
let fun f(p:point):real =
      p.x * p.x + p.y * p.y
in f(A)
```

est associé la syntaxe abstraite

```
Typedecl(point,TypeRecord(TRcons(x,\Int,TRcons(y,\Int,TRvide))),
  Let(A,ExprRecord(ERcons(x,\Cte(3),ERcons(y,\Cte(4),ERvide))),
    Letfun(f,p,Nomtype(point),\Real,
        +(\times(AccesRecord(\Var(p),x),AccesRecord(\Var(p),x)),
          \times(AccesRecord(\Var(p),x),AccesRecord(\Var(p),x))),
      \App(f,\Var(A)))))
```

Typage

$$\frac{\Gamma, \Gamma_r \cdot \{x = t\} \ \vdash\ er : tr}{\Gamma, \Gamma_r \ \vdash\ Typedcl(x, tr, er)}$$

$$\frac{\Gamma, \Gamma_r \ \vdash\ er : tr}{\Gamma \ \vdash\ ExprRecord(er) : TypeRecord(tr)}$$

$$\frac{\Gamma \ \vdash\ E : TypeRecord(tr)}{\Gamma \ \vdash\ AccesRecord(E, x) : t}$$

6.2 Les types sommes

L'étude des types sommes est en quelque sorte duale de celle des types produits. Nous étudions en premier le cas de la somme anonyme de deux types, avant des regarder les sommes étiquetées quelconques.

6.2.1 Somme anonyme de deux types

On notera par $t \mid t'$ le type somme des types t et t'. La grammaire abstraite des types est donc la suivante :

$$T \rightarrow Union(T,T) \mid \texttt{->}\ (T,T) \mid Nat \mid Bool$$

Syntaxe

Le type couple possède une fonction de construction pour former un couple à partir de deux éléments, et deux fonctions d'accès pour accéder soit au premier, soit au second élément du couple. Pour un type somme de deux types t_1 et t_2, la situation est inversée : il y a deux fonctions de constructions que l'on notera arbitrairement $Premier$ (respectivement $Second$) permettant de construire un élement du type somme $t_1|t_2$ à partir d'un élement de type t_1 (respectivement t_2). La fonction d'accès correspondante se trouve être la construction bien connue de distinction de cas que l'on notera ici par `case ... of`. Voici par exemple une fonction qui prend en argument un entier ou un booléen et retourne le successeur si c'est un entier et 0 si c'est un booléen

```
let f = fun x : (Nat|Bool) ->
  case x of
    Premier(n) -> n+1
  | Second(b) -> 0
in
(f (Premier(1)))+(f (Second(true)))
```

La syntaxe abstraite des expressions de type somme est :

$$E \rightarrow Premier(E,E) \mid Second(E) \mid Cas(E, Id, E, Id, E) \mid \ldots$$

Donnons une syntaxe concrête avec les règles de calcul de la syntaxe abstraite :

E	$\rightarrow$	$Premier(E)$	$\{Premier(E_1)\}$
E	$\rightarrow$	$Second(E)$	$\{Second(E_1)\}$
E	$\rightarrow$	`case` E `of` $Premier(Id)$ `->` E `|` $Second(Id)$ `->` E `endcase`	$\{Cas(E_1, \mathrm{val}(Id_1), E_2, \mathrm{val}(Id_1), E_3)\}$
E	$\rightarrow$	$\ldots$	$\{\ldots\}$

Les points de suspension figurent les règles étudiées au cours des chapitres antérieurs, qui peuvent toutes être récupérées telles quelles.

Typage

Nous aurons besoin de trois nouvelles règles de typage, une pour chaque nouvelle forme d'expression abstraite :

$$\frac{\Gamma \vdash e_1 : t_1|t_2 \qquad \Gamma \vdash e_2 : t_3 \qquad \Gamma \vdash e_3 : t_3}{\Gamma \vdash Cas(e_1, x_1, e_2, x_2, e_3) : t_3}$$

$$\frac{\Gamma \vdash e : t_1}{Premier(e) : t_1|t_2}$$

$$\frac{\Gamma \vdash e : t_2}{Second(e) : t_1|t_2}$$

Noter que la règle du `case` exprime bien que les deux expressions des deux alternatives doivent être de même type.

Évaluation

Nous avons maintenant quatre nouvelles règles d'évaluation :

$$\frac{\Gamma \vdash e_1 \Rightarrow Premier(v_1) \qquad \Gamma \cdot \{x_1 = v_1\} \vdash e_2 \Rightarrow v_2}{\Gamma \vdash Cas(e_1, x_1, e_2, x_2, e_3) \Rightarrow v_2}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow Second(v_1) \qquad \Gamma \cdot \{x_2 = v_1\} \vdash e_3 \Rightarrow v_2}{\Gamma \vdash Cas(e_1, x_1, e_2, x_2, e_3) \Rightarrow v_2}$$

$$\frac{\Gamma \vdash e \Rightarrow v}{Premier(e) \Rightarrow Premier(v)}$$

$$\frac{\Gamma \vdash e \Rightarrow v}{Second(e) \Rightarrow Second(v)}$$

6.2.2 Sommes étiquetés

On trouve de tels types sous des dénominations variées dans les langages de programmation. En PASCAL, on trouve les enregistrements avec variantes qui permettent une définition de sommes. En langage C, il s'agit des types `union`. Dans les langages fonctionnels de la famille ML, il s'agit des types construits.

Chapitre 7

Sémantique d'un langage impératif

Ce chapitre s'intéresse à la sémantique des langages impératifs. Nous présentons un langage impératif avec affectations, instructions conditionnelles et boucles `while`. Nous définissons ensuite une traduction des programmes de ce langage impératif en des expressions fonctionnelles comme celles présentées dans le chapitre 4. Le but de cette traduction est de montrer comment les définitions formelles de sémantique d'un langage que nous avons données permettent de prouver de manière rigoureuse la correction d'une traduction d'un langage vers un autre.

7.1 Syntaxe concrète et abstraite

Notre langage impératif utilise seulement les types `bool` et `int`. Un programme commence avec une séquence (qui peut être vide) de déclarations des variables avec leurs valeurs initiales, suivi par une séquence d'instructions. Le seul type d'instruction simple est l'affectation; les instructions peuvent être enchaînées avec le symbole « `;` », et peuvent être composées par des « parenthèses » `begin...end`, les boucles `while` et l'instruction conditionnelle. Voilà un exemple d'un programme pour calculer la valeur de 5^3. Après l'exécution du programme, la variable `resultat` contiendra le résultat 125.

```
base:     int = 5;
exposant: int = 3;
resultat: int = base;
index:    int = 1;
while exposant > index do begin
  resultat := resultat * exposant;
  index := index+1;
end;
```

La syntaxe abstraite choisie pour notre langage impératif est donnée par la grammaire suivante. Nous utilisons pour les expressions arithmético-logiques la grammaire donnée dans la section 3.2 :

$$
\begin{array}{rcl}
P & \rightarrow & \mathrm{Prog}(D, I) \\
D & \rightarrow & \mathrm{Vide} \mid \mathrm{Declare}(\mathit{String}, T, E, D) \\
I & \rightarrow & \mathrm{Skip} \mid \mathrm{Cons}(I, I) \mid \mathrm{Affect}(\mathit{String}, E) \mid \mathrm{Cond}(E, I, I) \mid \mathrm{While}(E, I) \\
T & \rightarrow & \mathrm{Bool} \mid \mathrm{Int} \\
E & \rightarrow & \ldots \qquad \text{(voir section 3.2)}
\end{array}
$$

$$\frac{\Gamma \vdash i}{\Gamma \vdash \text{Prog}(\text{Vide}, i)} \qquad \frac{\Gamma \cdot \{s : t\} \vdash \text{Prog}(d, i) \qquad \Gamma \vdash e : t}{\Gamma \vdash \text{Prog}(\text{Declare}(s, t, e, d), i)} \quad \text{si } s \notin \Gamma$$

$$\frac{}{\Gamma \vdash \text{Skip}} \qquad \frac{\Gamma \vdash i_1 \qquad \Gamma \vdash i_2}{\Gamma \vdash \text{Cons}(i_1, i_2)}$$

$$\frac{\Gamma \vdash e : \text{Bool} \qquad \Gamma \vdash i}{\Gamma \vdash \text{While}(e, i)} \qquad \frac{\Gamma \vdash e : \text{Bool} \qquad \Gamma \vdash i_1 \qquad \Gamma \vdash i_2}{\Gamma \vdash \text{Cond}(e, i_1, i_2)}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \text{Affect}(s, e)} \quad \text{si } s : t \in \Gamma$$

FIG. 7.1 – *Typage des programmes impératifs.*

L'axiome de cette grammaire est P.

Pour la grammaire concrète non-ambiguë et les règles de calcul de la syntaxe abstraite nous employons également les règles données dans section 3.2:

P	$\rightarrow$	$DS\ IS$	$\{\text{Prog}(DS_1, IS_1)\}$
DS	$\rightarrow$	ε	$\{\text{Vide}\}$
DS	$\rightarrow$	$String : T = E\ ;\ DS$	$\{\text{Declare}(\text{val}(String_1), T_1, E_1, DS_1)\}$
IS	$\rightarrow$	ε	$\{\text{Skip}\}$
IS	$\rightarrow$	$I\ ;\ IS$	$\{\text{Cons}(I_1, IS_1)\}$
I	$\rightarrow$	$String$ `:=` E	$\{\text{Affect}(\text{val}(String_1), E_1)\}$
I	$\rightarrow$	`if` E `then` I `else` I	$\{\text{Cond}(E_1, I_1, I_2)\}$
I	$\rightarrow$	`while` E `do` I	$\{\text{While}(E_1, I_1)\}$
I	$\rightarrow$	`begin` IS `end`	$\{IS_1\}$
E	$\rightarrow$	... (voir section 3.2)	

La syntaxe abstraite correspondante au programme en syntaxe concrète donné au-dessus est:

$$\begin{array}{l}\text{Prog}(\text{Declare}(base, \text{Int}, 5, \text{Declare}(exposant, \text{Int}, 3,\\ \qquad \text{Declare}(resultat, \text{Int}, \text{Var}(base), \text{Declare}(index, \text{Int}, 1, \text{Vide})))),\\ \quad \text{Cons}(\text{While}(> (\text{Var}(exposant), \text{Var}(index)),\\ \qquad \text{Cons}(\text{Affect}(\text{Var}(resultat), \times(\text{Var}(resultat), \text{Var}(exposant))),\\ \qquad\quad \text{Cons}(\text{Affect}(\text{Var}(index), +(\text{Var}(index), \text{Cte}(1))),\\ \qquad\qquad \text{Skip}))),\\ \quad \text{Skip}))\end{array}$$

7.2 Typage

Les règles de typage sont données sur la figure 7.1. On utilise deux sortes de jugements :

1. des jugements $\Gamma \vdash e$ où e est une expression de la syntaxe abstraite de sorte P ou I, disant que e est bien typé dans l'environnement Γ ;

2. des jugements $\Gamma \vdash e : t$ où e est une expression de la syntaxe abstraite de la sorte E et t est un type, disant que e a dans l'environnement Γ le type t. Les règles pour les jugements de ce type sont données sur la figure 3.6.

$$\frac{\Gamma \vdash i \Rightarrow \Gamma'}{\Gamma \vdash \mathrm{Prog}(\mathrm{Vide}, i) \Rightarrow \Gamma'} \qquad \frac{\Gamma \vdash e \Rightarrow v \qquad \Gamma \cdot \{s = v\} \vdash \mathrm{Prog}(d, i) \Rightarrow \Gamma'}{\Gamma \vdash \mathrm{Prog}(\mathrm{Declare}(s, t, e, d), i) \Rightarrow \Gamma'}$$

$$\frac{}{\Gamma \vdash \mathrm{Skip} \Rightarrow \Gamma} \qquad \frac{\Gamma \vdash i_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash i_2 \Rightarrow \Gamma''}{\Gamma \vdash \mathrm{Cons}(i_1, i_2) \Rightarrow \Gamma''}$$

$$\frac{\Gamma \vdash e \Rightarrow \texttt{true} \qquad \Gamma \vdash i_1 \Rightarrow \Gamma'}{\Gamma \vdash \mathrm{Cond}(e, i_1, i_2) \Rightarrow \Gamma'} \qquad \frac{\Gamma \vdash e \Rightarrow \texttt{false} \qquad \Gamma \vdash i_2 \Rightarrow \Gamma'}{\Gamma \vdash \mathrm{Cond}(e, i_1, i_2) \Rightarrow \Gamma'}$$

$$\frac{\Gamma \vdash e \Rightarrow v}{\Gamma \vdash \mathrm{Affect}(s, e) \Rightarrow \Gamma \cdot \{s = v\}} \qquad \frac{\Gamma \vdash e \Rightarrow \texttt{false}}{\Gamma \vdash \mathrm{While}(e, i) \Rightarrow \Gamma}$$

$$\frac{\Gamma \vdash e \Rightarrow \texttt{true} \qquad \Gamma \vdash i \Rightarrow \Gamma' \qquad \Gamma' \vdash \mathrm{While}(e, i) \Rightarrow \Gamma''}{\Gamma \vdash \mathrm{While}(e, i) \Rightarrow \Gamma''}$$

FIG. 7.2 – *Règles d'évaluations des programmes impératifs.*

7.3 Évaluation

Au contraire des programmes fonctionnels définis dans le chapitre 4, les programmes de notre langage impératif ne disposent pas d'une notation pour leur résultat. Afin de définir un langage de programmation pratique il faudrait bien sûr ajouter, parmi d'autres, des instructions de sorties. Pour raison de simplicité nous nous contraindrons ici à un langage minimal qui nous permet d'étudier les constructions essentielles de la programmation impérative. Le résultat de l'exécution d'un programme impératif sera pour l'ensemble des valeurs de ses variables.

Nous définissons la sémantique des programmes impératifs à l'aide des jugements de la forme $\Gamma \vdash i \Rightarrow \Gamma'$, disant que l'exécution de l'instruction (ou du programme) i dans l'environnement Γ termine et produit l'environnement Γ'. Les règles d'évaluation sont données sur la figure 7.2. On utilise, comme pour les règles de typage, des jugements d'évaluation des expressions arithmético-logiques définis sur la figure 3.7.

7.4 Traduction des programmes impératifs en programmes fonctionnels récursifs

Nous allons maintenant traduire les programmes impératifs en des programmes fonctionnels comme définis dans la section 4.2, *en conservant la sémantique des programmes.* Le problème est que les sémantiques des deux langages sont définies par des formalismes différents : la sémantique d'un programme fonctionnel est définie comme une fonction qui renvoie, pour un environnement donné, une valeur. La sémantique d'un programme impératif, par contre, est une fonction qui renvoie un environnement. Parce que l'exécution d'un programme impératif donné peut seulement changer les valeurs des variables qui se trouvent dans le programme, (par exemple des variables $x_1, \ldots, x_n$), on peut préciser l'objectif de la traduction comme suit : on traduit un programme impératif p avec les variables $x_1, \ldots, x_n$ de types respectifs $t_1, \ldots, t_n$ en une expression fonctionnelle f du type $(t_1, \ldots, t_n)$ tel que si $v_1, \ldots, v_n$ sont les valeurs des variables $x_1, \ldots, x_n$ après l'exécution de p, alors l'évaluation de f renvoie le n-uplet $(v_1, \ldots, v_n)$.

Afin de ne pas alourdir la traduction par des détails, nous sommes obligés de nous contraindre ici au cas des programmes qui utilisent une seule variable. Nous supposons doré-

$$\frac{i \Rightarrow f}{\mathrm{Prog}(\mathrm{Vide}, i) \Rightarrow f} \qquad \frac{\mathrm{Prog}(d, i) \Rightarrow f}{\mathrm{Prog}(\mathrm{Declare}(x, \mathrm{Int}, e, d), i) \Rightarrow \mathrm{Let}(x, e, f)}$$

$$\frac{}{\mathrm{Skip} \Rightarrow x} \qquad \frac{i_1 \Rightarrow f_1 \qquad i_2 \Rightarrow f_2}{\mathrm{Cond}(e, i_1, i_2) \Rightarrow \mathrm{If}(e, f_1, f_2)}$$

$$\frac{i_1 \Rightarrow f_1 \qquad i_2 \Rightarrow f_2}{\mathrm{Cons}(i_1, i_2) \Rightarrow \mathrm{Let}(x, f_1, f_2)} \qquad \frac{}{\mathrm{Affect}(x, e) \Rightarrow e}$$

$$\frac{i \Rightarrow f}{\mathrm{While}(e, i) \Rightarrow \mathrm{LetFunRec}(l, x, \mathrm{Int}, \mathrm{Int}, \mathrm{If}(e, \mathrm{App}(l, f), \mathrm{Var}(x)), \mathrm{App}(l, (\mathrm{Var}(x))))}$$

FIG. 7.3 – *Traduction des programmes impératifs, avec une seule variable x du type* `int`*, aux programmes fonctionnels.*

navant que cette variable s'appelle x, et qu'elle a le type `int`. On pourrait généraliser cette traduction au cas des programmes impératifs utilisant un nombre quelconque de variables à l'aide des types structurés présentés dans le chapitre 6 (voir exercice 7.4).

On obtient, par exemple, pour le programme

```
x: int = 7;
while x > 1 do begin
  if 2*(x/2) = x
  then x := x/2
  else x := 3*x+1
end
```

le programme fonctionnel suivant:

```
let x = 7
in let rec l(x:int):int =
     if x > 1
     then l(if 2*(x/2)=x then x/2 else 3*x+1)
     else x
   in l(x)
```

Notons que la boucle `while` d'un programme impératif se traduit en une fonction récursive.

Les règles pour les jugements de la forme $i \Rightarrow f$, exprimant que le programme fonctionnel f est la traduction du programme impératif i, sont données par la figure 7.3.

Nous supposons que les identificateurs x et l n'apparaissent pas dans i. Notons que les expressions arithmético-logiques font partie du langage impératif et du langage fonctionnel, et que la traduction est par conséquent immédiate.

Théorème 7.1 *Soit p un programme impératif avec pour seul identificateur x. Soit f l'expression fonctionnelle traduction de p, c'est-à-dire telle que le jugement $p \Rightarrow f$ soit valide. Soit Γ, Γ' des environnements quelconques. On a la proposition suivante :*

$$\textit{Si} \quad \Gamma \vdash p \Rightarrow \Gamma' \quad \textit{alors} \quad \Gamma \vdash f \Rightarrow v' \quad \textit{avec} \quad x = v' \in \Gamma'$$

Autrement dit, la valeur de f est exactement la valeur de x dans l'environnement obtenu par exécution de p, ce qui est bien ce que l'on souhaite pour dire que la traduction est correcte.

Comment peut-on aborder la preuve du théorème 7.1? Une première idée est une preuve par récurrence sur la structure du programme impératif i. Regardons les règles de la figure 7.2, nous remarquons que pour toutes les constructions *sauf le While* les programmes dans les hypothèses des règles sont des sous-expressions du programme de la conclusion. Par conséquent, une récurrence sur la *taille* du programme, c'est-à-dire sur le nombre d'étiquettes abstraites qu'il contient, peut réussir dans tous les cas sauf le While. Pour le While, le programme de la conclusion se trouve également dans l'hypothèse dans le cas où la condition est satisfaite. Dans ce cas, nous somment donc obligés d'utiliser une technique appropriée à la nature de l'instruction While, c'est une récurrence sur le nombre d'itérations de la boucle. Ces deux idées sont réunies dans une *récurrence sur la taille de l'arbre de dérivation* du jugement $\Gamma \vdash i \Rightarrow \Gamma'$.

Preuve:

Avant de prouver le théorème 7.1 nous montrons une propriété analogue pour les instructions :

> Quelle que soit l'instruction i avec pour seul identificateur x, si $\Gamma \vdash i \Rightarrow \Gamma'$ avec $x \in \Gamma$ et $i \Rightarrow f$, alors $\Gamma \vdash f \Rightarrow v'$ avec $\{x = v'\} \in \Gamma'$.

Nous commençons avec les deux cas des instructions simples :

Cas : $i = \text{Skip}$

On a $\Gamma \vdash \text{Skip} \Rightarrow \Gamma'$, donc il faut par les règles d'évaluation des programmes impératifs que $\Gamma' = \Gamma$. Selon les règles des traduction, $\text{Skip} \Rightarrow \text{Var}(x)$.

Donc, il faut montrer que $\Gamma \vdash \text{Var}(x) \Rightarrow v$ où $\{x = v\} \in \Gamma$. C'est une conséquence triviale des règles d'évaluation des expressions arithmético-logiques et du fait que $\{x = v\} \in \Gamma$.

Cas : $i = \text{Affect}(x, e)$

On a $\Gamma \vdash \text{Affect}(x, e) \Rightarrow \Gamma'$, donc il faut par les règles d'évaluation des programmes impératifs qu'il y a un v tel que $\Gamma \vdash e \Rightarrow v$ et $\Gamma' = \Gamma \cdot \{x = v\}$. Selon les règles de traduction, $\text{Affect}(x, e) \Rightarrow e$.

Donc il faut montrer que $\Gamma \vdash e \Rightarrow v'$ où $\{x = v'\} \in \Gamma \cdot \{x = v\}$. C'est une conséquence triviale de l'hypothèse $\Gamma \vdash e \Rightarrow v$ et du fait que $v' = v$.

Dans les trois cas des instructions composées nous sommes amenés à faire une preuve par récurrence.

Cas : $i = \text{Cond}(e, j_1, j_2)$

On a $\Gamma \vdash \text{Cond}(e, j_1, j_2) \Rightarrow \Gamma'$. Selon les règles de traduction, il existe f_1 et f_2 tels que

$$\begin{aligned} j_1 &\Rightarrow f_1 \\ j_2 &\Rightarrow f_2 \\ \text{Cond}(e, j_1, j_2) &\Rightarrow \text{If}(e, f_1, f_2) \end{aligned}$$

Il faut montrer que $\Gamma \vdash \text{If}(e, f_1, f_2) \Rightarrow v'$ où $\{x = v'\} \in \Gamma'$.

Sous-cas : $\Gamma \vdash e \Rightarrow \texttt{true}$

Par les règles d'évaluation des programmes impératifs on obtient

$$\Gamma \vdash j_1 \Rightarrow \Gamma'$$

L'application de l'hypothèse de récurrence à j_1 donne

$$\Gamma \vdash f_1 \Rightarrow v'$$

Donc, on obtient par les règles d'évaluation des programmes fonctionnels :

$$\frac{\vdots \qquad \vdots}{\Gamma \vdash e \Rightarrow \texttt{true} \qquad \Gamma \vdash f_1 \Rightarrow v'} \\ \Gamma \vdash \mathrm{If}(e, f_1, f_2) \Rightarrow v'$$

Sous-cas : $\Gamma \vdash e \Rightarrow \texttt{false}$
Ce cas est analogue au cas précédent.

Cas : $i = \mathrm{Cons}(j_1, j_2)$
On a $\Gamma \vdash \mathrm{Cons}(j_1, j_2) \Rightarrow \Gamma'$. Selon les règles de traduction il existe f_1 et f_2 tels que

$$\begin{aligned} j_1 &\Rightarrow f_1 \\ j_2 &\Rightarrow f_2 \\ \mathrm{Cons}(j_1, j_2) &\Rightarrow \mathrm{Let}(x, f_1, f_2) \end{aligned}$$

Il faut montrer que $\Gamma \vdash \mathrm{Let}(x, f_1, f_2) \Rightarrow v'$ où $\{x = v'\} \in \Gamma'$. Selon les règles d'évaluation des programmes impératifs il existe Γ'' tel que

$$\begin{aligned} \Gamma &\vdash j_1 \Rightarrow \Gamma'' \\ \Gamma'' &\vdash j_2 \Rightarrow \Gamma' \end{aligned}$$

Soit $\{x = v''\} \in \Gamma''$. Puisque x est le seul identificateur de j_2, on obtient également

$$\Gamma \cdot \{x = v''\} \vdash j_2 \Rightarrow \Gamma'$$

L'application de l'hypothèse de récurrence à j_1 et j_2 donne

$$\begin{aligned} \Gamma &\vdash f_1 \Rightarrow v'' \\ \Gamma \cdot \{x = v''\} &\vdash f_2 \Rightarrow v' \end{aligned}$$

Donc, on obtient par les règles d'évaluation des programmes fonctionnels

$$\frac{\vdots \qquad \vdots}{\Gamma \vdash f_1 \Rightarrow v'' \qquad \Gamma \cdot \{x = v''\} \vdash f_2 \to v'} \\ \Gamma \vdash \mathrm{Let}(x, f_1, f_2) \vdash v'$$

Cas : $i = \mathrm{While}(e, j)$
On a $\Gamma \vdash \mathrm{While}(e, j) \Rightarrow \Gamma'$. Selon les règles de traduction il existe f tel que

$$\begin{aligned} j &\Rightarrow f \\ \mathrm{While}(e, j) &\Rightarrow \mathrm{LetFunRec}(l, x, \mathrm{Int}, \mathrm{Int}, \mathrm{If}(e, \mathrm{App}(l, f), \mathrm{Var}(x)), \mathrm{App}(l, (\mathrm{Var}(x)))) \end{aligned}$$

Il faut montrer que

$$\Gamma \vdash \mathrm{LetFunRec}(l, x, \mathrm{Int}, \mathrm{Int}, \mathrm{If}(e, \mathrm{App}(l, f), \mathrm{Var}(x)), \mathrm{App}(l, (\mathrm{Var}(x)))) \Rightarrow v'$$

où $x = v' \in \Gamma'$. Soit $x = v \in \Gamma$ et $\Gamma_1 = \Gamma \cdot \{l = (x, \mathrm{If}(e, \mathrm{App}(l, f), \mathrm{Var}(x)))\}$.

Sous-cas : $\Gamma \vdash e \Rightarrow \texttt{false}$
Puisque l n'apparaît pas dans e et puisque $\{x = v\} \in \Gamma$ on a aussi

$$\Gamma_1 \cdot \{x = v\} \vdash e \Rightarrow \texttt{false}$$

Par les règles de sémantique impérative il faut que $\Gamma = \Gamma'$, et par conséquent que $v = v'$. On construit la dérivation suivante:

$$\dfrac{\Gamma_1 \vdash \mathrm{Var}(x) \Rightarrow v \qquad \dfrac{\dfrac{\vdots}{\Gamma_1 \cdot \{x = v\} \vdash e \Rightarrow \texttt{false}} \quad \Gamma_1 \cdot \{x = v\} \vdash \mathrm{Var}(x) \Rightarrow v}{\Gamma_1 \cdot \{x = v\} \vdash \mathrm{If}(e, \mathrm{App}(l, f), \mathrm{Var}(x)) \Rightarrow v}}{\dfrac{\Gamma_1 \vdash \mathrm{App}(l, (\mathrm{Var}(x))) \Rightarrow v}{\Gamma \vdash \mathrm{LetFunRec}(l, x, \mathrm{Int}, \mathrm{Int}, \mathrm{If}(e, \mathrm{App}(l, f), \mathrm{Var}(x)), \mathrm{App}(l, (\mathrm{Var}(x)))) \Rightarrow v}}$$

Sous-cas : $\Gamma \vdash e \Rightarrow \texttt{true}$
Par les règles d'évaluation des programmes impératifs, il existe Γ'' tel que

$$\begin{aligned} \Gamma \vdash i &\Rightarrow \Gamma'' \\ \Gamma'' \vdash \mathrm{While}(e, i) &\Rightarrow \Gamma' \end{aligned}$$

Puisque x est le seul identificateur dans $\mathrm{While}(e, i)$ on obtient également

$$\Gamma_1 \cdot \{x = v\} \vdash \mathrm{While}(e, i) \Rightarrow \Gamma'$$

L'application de l'hypothèse de récurrence donne

$$\Gamma_1 \cdot \{x = v\} \vdash \mathrm{LetFunRec}(l, x, \mathrm{Int}, \mathrm{Int}, \mathrm{If}(e, \mathrm{App}(l, f), \mathrm{Var}(x)), \mathrm{App}(l, (\mathrm{Var}(x)))) \Rightarrow v'$$

Donc, selon les règles d'évaluation des programmes fonctionnels et puisque Γ_1 contient déjà la définition de l:

$$\Gamma_1 \cdot \{x = v\} \vdash \mathrm{App}(l, (\mathrm{Var}(x))) \Rightarrow v'$$

On construit donc la dérivation suivante:

$$\dfrac{\Gamma_1 \vdash \mathrm{Var}(x) \Rightarrow v \qquad \dfrac{\dfrac{\vdots}{\Gamma_1 \cdot \{x = v\} \vdash e \Rightarrow \texttt{true}} \quad \dfrac{\vdots}{\Gamma_1 \cdot \{x = v\} \vdash \mathrm{App}(l, f) \Rightarrow v'}}{\Gamma_1 \cdot \{x = v\} \vdash \mathrm{If}(e, \mathrm{App}(l, f), \mathrm{Var}(x)) \Rightarrow v'}}{\dfrac{\Gamma_1 \vdash \mathrm{App}(l, (\mathrm{Var}(x))) \Rightarrow v'}{\Gamma \vdash \mathrm{LetFunRec}(l, x, \mathrm{Int}, \mathrm{Int}, \mathrm{If}(e, \mathrm{App}(l, f), \mathrm{Var}(x)), \mathrm{App}(l, (\mathrm{Var}(x)))) \Rightarrow v'}}$$

Afin de compléter la preuve de théorème 7.1, il nous reste à traiter les deux cas des programmes de la forme $i = \mathrm{Prog}(\mathrm{Vide}, j)$ et $i = \mathrm{Prog}(\mathrm{Declare}(s, t, e, d), i)$. Ces deux preuves sont très similaires aux cas traités au-dessus, voir exercice 7.3.

□

7.5 Exercices

Exercice 7.1
On définit un deuxième système de règles pour l'évaluation des programmes impératifs, en remplaçant dans le système de la figure 7.2 les deux règles pour While par la règle

$$\dfrac{\Gamma \vdash \mathrm{Cond}(e, \mathrm{Cons}(i, \mathrm{While}(e, i)), \mathrm{Skip}) \Rightarrow \Gamma'}{\Gamma \vdash \mathrm{While}(e, i) \Rightarrow \Gamma'}$$

Montrer que les deux systèmes de règles définissent la même sémantique.

Exercice 7.2
Étendre la syntaxe concrète, la syntaxe abstraite et les règles de typage et d'évaluation par les boucles `repeat...until`. Donner une règle de traduction en programmes fonctionnelles.

Exercice 7.3
Completer la preuve de théorème 7.1 par les preuves des deux cas $i = \text{Prog}(\text{Vide}, j$ et $i = \text{Prog}(\text{Declare}(s,t,e,d),i)$.
Correction :

Cas: *$i = Prog(Vide, j)$*

Par notre hypothèse, $Prog(Vide, j) \Rightarrow f$ est vrai. La seule règle qui permet de déduire un jugement de cette forme est la première règle de figure 7.3. Si un sait que $Prog(Vide, j) \Rightarrow f$ est vrai et qu'on a appliqué la première règle pour le déduire, il faut que son hypothèse est aussi vrai. Donc, $f \Rightarrow j$ est vrai.

On applique un raisonnement similaire aux jugements d'évaluation: Par hypothèse, $\Gamma \vdash Prog(Vide, j) \Rightarrow \Gamma'$ est vrai. La seule règle qui permet de déduire un tel jugement est la première règle de la figure 7.2. Donc, sa hypothèse $\Gamma \vdash j \Rightarrow \Gamma'$ est aussi vrai.

On sait donc que $f \Rightarrow j$ et $\Gamma \vdash j \Rightarrow \Gamma'$ sont vrai. La longueur de la dérivation de $f \Rightarrow j$ doit être inférieure à la longueur de la dérivation de $Prog(Vide, j) \Rightarrow f$, donc nous appliquont la hypothèse d'induction et obtennont que $\Gamma \vdash f \Rightarrow \Gamma'(x)$ est vrai.

Cas: *$i = Prog(Declare(s,t,e,d),i)$*

Par les règles de sémantique impérative, il existe v tel que $\Gamma \vdash e \Rightarrow v$ et $\Gamma \cdot \{s = v\} \vdash Prog(d,i) \Rightarrow \Gamma'$. Par les règles de traduction, $f = Let(s,e,g)$ où $Prog(d,i) \Rightarrow g$. Nous obtenons par l'hypothèse d'induction $\Gamma \cdot \{s = v\} \vdash g \Rightarrow \Gamma'(x)$ et, par les règles de sémantique fonctionnelle, $\Gamma \vdash f \Rightarrow \Gamma'(x)$.

Exercice 7.4
Généraliser la traduction du chapitre 7.4 au cas de programmes avec un nombre quelconque de variables.

=======

Troisième partie

Études de cas

Chapitre 1

Généralités sur les études de cas

1.1 Le contenu des rapports

1.1.1 Manuel d'utilisation

Cette partie s'adresse aux utilisateurs de votre programme. Il faut qu'un utilisateur trouve dans cette partie toutes les informations nécessaires pour utiliser votre programme. En particulier, vous expliquerez dans cette partie :

1. les fonctionnalités de votre programme : brièvement, sans répéter l'énoncé, mais en expliquant tous les ajouts, choix ou options que vous avez éventuellement été amenés à faire ;

2. comment lancer votre programme ;

3. la syntaxe que votre programme accepte. **Cette syntaxe sera donnée en deux parties, une pour les textes HTML, une pour les expressions et les définitions.** Pour chacune des deux grammaires vous définirez les unités lexicales par des tokens associés à des expressions rationnelles, puis la syntaxe complète par une grammaire sur le vocabulaire formé par les tokens ;

4. les cas d'erreur possibles et le comportement que devra avoir votre programme dans ces cas d'erreur.

1.1.2 Structure du programme et découpage en modules

Dans une seconde partie de votre rapport, vous décrirez le raisonnement que vous avez été amené à faire pour l'analyse descendante du programme, et de quelle façon vous en avez déduit le découpage en modules qui vous paraissait le mieux adapté. Vous dessinerez le graphe de dépendances entre les modules.

1.1.3 Description des modules

Dans une troisième partie du rapport, et pour chacun des modules, dans un ordre compatible avec le graphe de dépendance, vous donnerez dans un premier temps les spécifications de ce module :

1. son rôle en général ;

2. les types qui y sont définis, décrits abstraitement, c.-à-d. par leur noms et les spécifications des fonctions qui permettent de créer des valeurs de ce type, et de les manipuler. **vous décrirez les types de données pour les documents HTML, les expressions et les définitions par des grammaires abstraites** ;

3. les spécifications des fonctions qui y sont définies.

(Vous ne décrirez que les types et fonctions de l'interface du module). Vous décrirez dans un second temps la réalisation de ce module :

1. les structures de données que vous avez choisies pour représenter les types, et les raisons de ces choix ;

2. les algorithmes que vous avez choisis pour les fonctions, *lorsque ceux-ci ne sont pas triviaux.*

Si vous avez été amenés à définir des types privés ou des fonctions locales au module, vous les spécifierez.

Les algorithmes seront décrits à un haut niveau. Les fonctions d'analyse syntaxique devront être décrites par des grammaires avec actions. **L'évaluation des cases des tableaux sera décrite par des règles sémantiques s'appliquant sur les jugements appropriés. Chaque type de jugement devra être clairement défini.**

1.1.4 Jeu de tests

Dans une quatrième partie, vous donnerez enfin un ensemble de fichiers de tests que vous aurez choisis pour tester toutes les fonctionnalités de votre programme, sans oublier les cas d'erreurs. Vous donnerez les traces d'exécution de vos fichiers de tests, et vous commenterez les résultats obtenus. Vous expliquerez en particulier les cas où les tests ne se passent pas comme prévu, en essayant de trouver pourquoi. Les jeux de tests sont notés sur l'exhaustivité (c.-à-d. si les tests couvrent tous les cas d'exécution du programme), et aussi sur les commentaires expliquant pourquoi ces tests ont été choisis.

Enfin, il est inutile de rendre un listing de votre programme, mais par contre vous fournirez une disquette avec les fichiers sources (`.ml` et `.mli`) et les fichiers de tests. **Mettez vos noms et le nom de l'enseignant de votre groupe sur la disquette**.

1.1.5 Soutenance

Une soutenance se déroulera lors de votre séance de TP entre le mercredi 3 juin et le mardi 9 juin. Vous préparerez une petite présentation des fonctionnalités de votre programme sur des exemples bien choisis (environ 5mn), puis vous répondrez **individuellement** aux questions qui vont seront posées.

Chapitre 2

Compilation d'un mini-Pascal

Chapitre 3

Un langage d'expressions rationnelles

Chapitre 4

Un langage pour la composition de documents

Dans cette étude de cas, nous allons écrire un programme de *composition* de documents, inspiré d'un logiciel déjà existant appelé LaTeX. Le but est de fournir en entrée une description de *haut niveau* d'un texte, et ce logiciel devra *mettre en page* le texte. Une description de haut niveau d'un texte consiste à décrire ces composants logiques : titre du document, titres et sous-titres de sections, énumérations, etc.; et la mise en page consiste alors à utiliser les caractères d'une fonte et d'une taille appropriée, couper les phrases et répartir les espaces entre les mots de manière à produire un texte avec des marges données, numéroter les titres et sous-titres de sections, numéroter les pages, choisir de changer de page aux endroits opportuns, etc.

4.1 Documents simples

Un fichier source type que l'on désire donner en entrée du programme est donné figure 4.1. Une sortie possible est alors donnée figure 4.2. Cette mise en page a été effectuée par un certain nombre de choix de style :

- la largeur du texte est de 80mm ;
- les paragraphes de texte doivent être « justifiés » (c.-à-d. marges alignées comme expliqué ci-dessus), dans la fonte Times de taille 3.5mm, le début de paragraphe devant être indenté de 7mm ;
- le titre du document doit être composé dans la fonte Helvetica, grasse, de taille 6mm ;
- un titre de section doit être composé dans la fonte Times, grasse, de taille 5mm ;
- un titre de sous-section doit être composé dans la fonte Times, oblique, de taille 4.2mm ;
- l'intervalle entre les lignes est de 1,2 fois la taille de la fonte ;
- etc.

Dans un premier temps, ces choix de style de composition pourront être codés directement dans le programme de composition. Nous proposerons à la section 4.1.5 une méthode pour paramétrer facilement le style.

Remarquer que les espaces entre les mots et les passages à la ligne du texte source n'ont aucune influence sur le texte de sortie : en effet les passages à la ligne et l'espacement entre

```
\title{Ceci est le titre du document}

Voici une première phrase introductive au document.

\section{Ceci est un titre de section}

 Ceci est   le premier   paragraphe de la première section. Il contient
   trois phrases.   Noter la façon dont les espaces entre les mots sont
ajustés  de  manière à  bien marquer les marges à droite et à
gauche.

Voici le deuxième   paragraphe.    Le changement de paragraphe est
matérialisé par une ligne vide.

\section{Voici la deuxième section}

\subsection{Et ceci est une sous-section}

Les sous-sections sont des sections de niveau deux, numérotées avec
deux nombres.

\subsection{Deuxième sous-section}

La numérotation des sections et sous-sections doit évidemment être
calculée automatiquement par le logiciel de composition.
```

FIG. 4.1 – *Exemple de fichier source simple*

les mots est calculé de manière à ce que les lignes soient contre les marges gauches et droite, en remplissant les lignes au maximum.

Il nous reste à parler de la façon de produire le texte de sortie : pour cela nous allons utiliser un autre langage appelé Postscript, qui est un langage de description de textes à imprimer. Il s'agit en fait du langage de commande pour certaines imprimantes. Comme il n'est pas question dans cette étude d'apprendre ce langage, nous fournissons un module PASCAL qui permet de produire des fichiers Postscript de manière simple. Ce module est décrit dans la section suivante.

4.1.3 Le module POSTSCR, et l'outil GhostView

Ce module fournit l'essentiel des procédures permettant de créer un fichier Postscript :

- définition des fontes possibles ;
- ouverture et fermeture d'un fichier Postscript ;
- écriture d'une chaîne de caractères dans une fonte donnée à une position donnée de la page ;
- détermination de la largeur d'une chaîne de caractères dans une fonte donnée (indispensable pour la mise en page) ;
- passage à la page suivante ;

L'interface PASCAL de ce module est donnée en annexe.

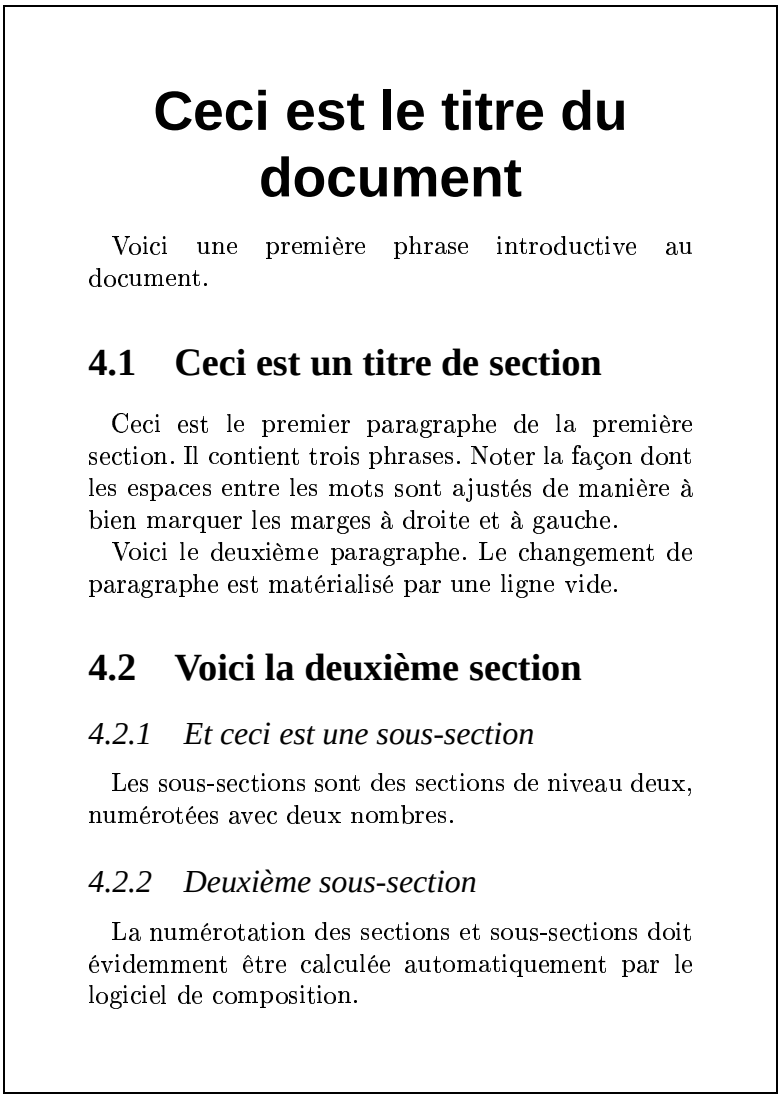


FIG. 4.2 – *Une page de sortie possible pour l'exemple*

Ce module permet donc facilement de produire des fichiers Postscript. Ensuite, nous souhaitons bien sûr pouvoir visualiser les pages produites à l'écran, et aussi imprimer ces pages. Pour cela, nous disposons de l'outil Ghostview. Il se lance sous Windows en cliquant sur l'icône GSView. Dans le menu qui vous est proposé, vous pouvez ouvrir un fichier Postscript (File/Open), et (après quelques secondes d'attente...) la première page est visualisée. La barre d'outil sur le côté gauche permet de passer à la page suivante, de zoomer, etc. Dans les menus est également proposé d'imprimer la page visualisée (File/Print) ou tout le fichier (File/Print File).

4.1.4 Les fichiers sources et leur syntaxe

Pour résumer, du point de vue pratique, le programme de composition devra opérer de la manière suivante :

1. analyse syntaxique du fichier d'entrée ;

2. le cas échéant, signaler une erreur de syntaxe et s'arrêter ;

3. si aucune erreur de syntaxe n'est détectée, construction d'une structure de données mémorisant le texte source ;

4. application d'algorithmes de composition sur la structure de données produite, la sortie étant un fichier Postscript construit à l'aide du module fourni.

Comme on a pu le remarquer sur l'exemple, il existe certains mots clés qui sont précédés par un caractère \ (backslash en anglais). En effet, les mots clés ou identificateurs que l'on trouve dans les langages de programmation classiques (`begin`, `end`, `if`, etc.) ne peuvent par être utilisés ici car ce sont des mots que l'on peut utiliser dans le texte. Aussi, dans cette étude, les mots-clés et identificateurs seront toujours formé d'un \ suivi d'une suite de lettres minuscules ou majuscules, ce qui donne ainsi `\title`, `\section` et `\subsection`. D'autre part, les parenthèses étant elles-mêmes des symboles de ponctuation habituels dans un texte, elles sont remplacées par les accolades.

La syntaxe des fichiers d'entrée doit suivre les règles suivantes :

- un texte est constitué d'une suite de *paragraphes* séparés par des lignes vides ;

- un paragraphe peut être soit un paragraphe *normal* formé par une séquence de *mots*, soit un paragraphe *spécial* (`title`, `section`, `subsection`) introduit par un mot clé puis une séquence de mots entre accolades.

- un mot est n'importe quelle suite de caractères non spéciaux (c.à-d. ni \, ni `{`, ni `}`, ni espace, tabulation ou retour à la ligne).

Comme montré sur l'exemple, les passages à la ligne du texte source n'ont pas d'influence sur les passages à la ligne du texte résultat. Aussi les espaces, tabulations et retour à la ligne pourront être considérés comme équivalents, et servent de séparateurs entre unités lexicales. Par contre, deux (ou plus) passages à la ligne successifs (éventuellement séparés par des espaces et tabulations) seront significatifs puisqu'ils séparent deux paragraphes : une telle séquence de caractères devra donc former une unité lexicale.

4.1.5 Extensions du langage

Le langage décrit jusqu'à présent permet de composer des documents très simples. Des constructions supplémentaires sont utiles pour des documents plus complexes.

Nous proposons ici quelques extensions, dans l'ordre de difficulté, et qui compterons pour quelques points dans la notation du rapport numéro 2.

Numérotation des pages et entêtes

Numéroter chaque page par un numéro placé en bas de page et centré. Introduire deux types de paragraphes supplémentaires :

`\lefthead{`*texte*`}`
`\righthead{`*texte*`}`

qui définissent des portions de texte à écrire sur chaque page, en haut à gauche pour `\lefthead`, en haut à droite pour `\righthead`.

α	'a'	\alpha	β	'b'	\beta	γ	'g'	\gamma
δ	'd'	\delta	ε	'e'	\epsilon	ζ	'z'	\zeta
η	'h'	\eta	θ	'q'	\theta	ι	'i'	\iota
κ	'k'	\kappa	λ	'l'	\lambda	μ	'm'	\mu
ν	'n'	\nu	ξ	'x'	\xi	o	'o'	\omicron
π	'p'	\pi	ρ	'r'	\rho	σ	's'	\sigma
τ	't'	\tau	υ	'u'	\upsilon	ϕ	'j'	\phi
χ	'c'	\chi	ψ	'y'	\psi	ω	'w'	\omega
Γ	'G'	\Gamma	Δ	'D'	\Delta	Θ	'Q'	\Theta
Λ	'L'	\Lambda	Ξ	'X'	\Xi	Π	'P'	\Pi
Σ	'S'	\Sigma	Υ	'U'	\Upsilon	Φ	'F'	\Phi
Ψ	'Y'	\Psi	Ω	'W'	\Omega			
$\geq$	179	\geq	$\leq$	163	\leq	$\neq$	185	\neq
$\pm$	177	\pm	$\times$	180	\times	$\div$	184	\div
$\sum$	229	\sum	$\int$	242	\int	$\prod$	213	\prod
∞	165	\infty	$\bullet$	183	\bullet	$\aleph$	192	\aleph
$\emptyset$	198	\emptyset	$\in$	206	\in	$\subset$	204	\subset
$\subseteq$	205	\subseteq	$\cup$	200	\cup	$\cap$	199	\cap
$\leftarrow$	172	\leftarrow	$\rightarrow$	174	\rightarrow	$\uparrow$	173	\uparrow
$\downarrow$	175	\downarrow	$\leftrightarrow$	171	\leftrightarrow			
$\Leftarrow$	220	\Leftarrow	$\Rightarrow$	222	\Rightarrow	$\Uparrow$	221	\Uparrow
$\Downarrow$	223	\Downarrow	$\Leftrightarrow$	219	\Leftrightarrow			
$\exists$	36	\exists	$\forall$	34	\forall	$\neg$	216	\neg
$\vee$	218	\vee	$\wedge$	217	\wedge			
$\spadesuit$	170	\spadesuit	$\heartsuit$	169	\heartsuit	$\diamondsuit$	168	\diamondsuit
$\clubsuit$	167	\clubsuit						

FIG. 4.3 – *Liste des symboles spéciaux et leurs identificateurs associés*

Symboles spéciaux

On pourra étendre la syntaxe de manière à pouvoir écrire à la place d'un mot un identificateur, désignant un symbole spécial dont la liste est donnée figure 4.3. Cette table donne pour chaque caractère spécial le caractère qui lui correspond dans la fonte Symbol de Postscript, ainsi que l'identificateur qui devra lui être associé.

On pourra ainsi écrire :

```
f(x)\rightarrow +\infty quand x\rightarrow a \Leftrightarrow \forall
A\geq 0, \exists \alpha>0, |x-a|<\alpha\Rightarrow f(x)>A
```

pour produire

$$f(x) \rightarrow +\infty \text{ quand } x \rightarrow a \Leftrightarrow \forall A \geq 0, \exists \alpha > 0, |x-a| < \alpha \Rightarrow f(x) > A$$

Paramétrage du style

Pour pouvoir changer le style d'un document, il est plus pratique de faire précéder le texte d'une commande additionnelle :

\documentstyle{*nom d'un fichier de style*}

où un fichier de style contiendra des déclarations de la forme suivante :

```
\textwidth{80}
```

```
\normalfontfamily{Times}
\normalfontsize{3.5}
\titlefontfamily{Helvetica}
\titlefontbold{true}
\titlefontsize{6}
```

Ces fichiers de style doivent pouvoir être lus avec le même analyseur lexical que les fichiers source LaTeX, mais avec un analyseur syntaxique différent.

4.1.6 Interface PASCAL du module POSTSCR

```
(**** fontes ************************************************************)

(* les familles de fontes postscript standard.                          *)
type
   font_family = (Times,Helvetica,Courier,Symbol) ;

(* description d'une fonte. size est la hauteur en mm.                  *)
(* toute utilisation d'une size <= 0 déclenche une erreur               *)
type
   fontdesc = record
                 family  : font_family;
                 oblique : boolean;
                 bold    : boolean;
                 size    : real
              end;

(**** fichiers ***********************************************************)

(* open_postscript(e,s) ouvre un fichier postscript en ecriture, nomme  *)
(* s.ps. Aucune operation d'ecriture n'est possible tant qu'aucun       *)
(* fichier n'est ouvert.                                                *)
(* declenche une erreur si l'ouverture ne reussit pas.                  *)
(* e donne quelle est le codage utilise pour les                        *)
(* lettres accentuees                                                   *)

type
   char_encoding = (ISO_8859, DOS, WINDOWS);

procedure open_postscript(e : char_encoding;
                          filename : string);

(* termine le fichier postscript en cours                               *)

procedure close_postscript;

(**** navigation *********************************************************)

(* new_page() enregistre la page courante dans le fichier               *)
(* postscript, et demarre une nouvelle page blanche                     *)

procedure new_page;

(**** text ***************************************************************)

(* output_text(f,x,y,s) affiche le texte donne par la chaine s sur la   *)
(* page courante, a la position (x,y) qui sera le coin inferieur        *)
```

```
(* gauche du texte, ceci dans la police de caracteres f.                *)
(* x et y sont en millimetres, l'origine etant situee approximativement *)
(* a 2cm du bord gauche et du bas de la feuille                         *)
(* Declenche une erreur si la position (x,y) déborde de la              *)
(* feuille (mais ne verifie pas si le texte lui-meme ne déborde pas !)  *)

procedure output_text(f   : fontdesc ;
                      x,y : real;
                      s   : string);

(* text_width(f,s) retourne la largeur de la chaine s dans la fonte f, *)
(* en millimetres                                                      *)

function text_width(f : fontdesc ;
                    s : string):real;
```

À titre d'exemple, voici la suite des instructions PASCAL à exécuter pour créer un fichier Postscript d'une page, contenant un petit texte :

```
uses postscr;

var
   f : fontdesc;
   w : real;

begin
   open_postscript(DOS,'c:\result');

   f.family := Times;
   f.oblique := false;
   f.bold := false;
   f.size := 10.0 ;

   output_text(f,0.0,230.0,'Voici un texte qui se pour');
   w := text_width(f,'Voici un texte qui se pour');
   output_text(f,w,220.0,'suit ici.');

   close_postscript;
end;
```

4.2 Les formules mathématiques

Comme première extension, nous souhaitons pouvoir saisir des formules mathématiques qui seront automatiquement composées par le programme.

4.2.1 Syntaxe des formules

Il y aura en fait deux types de formules : les formules simples, dont la hauteur est la même que le texte, qui pourront faire partie du texte d'un paragraphe normal, et les formules complexes, qui formeront un paragraphe à part entière, centré. Les formules simples sont :

- les constantes entières, réelles, par exemple « 23 » , « 3.1416 » ;
- les variables, qui seront formées par une lettre majuscule ou minuscule, par exemple « x », « A » ; remarquer que celles-ci seront composées en italique ;

+	75	+	−	77	-	(vide)		*
/	79	/						
±	177	\pm	×	180	\times	÷	184	\div
>	94	>	<	92	<	=	93	=
≥	179	>=	≤	163	<=	≠	185	<>
∈	206	\in	⊂	204	\subset	⊆	205	\subseteq
∪	200	\cup	∩	199	\cap			

FIG. 4.4 – *Liste des opérations et leurs identificateurs associés*

- les lettres grecques, par exemple « π », « Γ », que l'on saisira `\pi`, `\Gamma` dans le texte source ;
- une opération entre deux formules, par exemple « $x + 1$ » (écrit `x+1` dans le source) , « $E \cup F$ » (écrit `E\cup F`) ;
- une fonction appliquée à une formule ou plusieurs formules, par exemple « $f(x)$ » qui sera écrit simplement `f(x)`, ou bien « $\sin(x)$ » qui sera écrit `\sin(x)` ; noter que la fonction sin est écrite en caractères droits et pas en italique ;
- une formule entre parenthèses « $(x + 1)$ », entre crochets « $[x + y]$ », entre barres « $|a + b|$ » ;
- une formule puissance une autre formule, par exemple « x^2 », ce qui sera noté `x^2` dans le texte source ;
- une formule indice une autre formule, par exemple « x_i », ce qui sera noté `x_i` dans le texte source ;
- une formule puissance une autre formule et indice une troisième, par exemple « x_i^2 », ce qui sera noté `x_i^2` dans le texte source ;

La liste des opérations est donnée figure 4.4, chacune d'elles étant accompagnée du numéro du caractère correspondant dans la fonte Symbol et de sa syntaxe dans le source. Noter que dans la mesure où les parenthèses correspondent à un affichage effectif, les accolades pourront être utilisées pour « parenthéser » les formules, on écrira ainsi `2^{x+y}` pour produire 2^{x+y} et `x_{i+j}` pour produire x_{i+j}, ou encore `x_{i^2}` pour obtenir x_{i^2}.

Les formules complexes seront formées à partir des formules simples par

- des fractions, par exemple
$$\frac{\pi}{2}$$
qui sera noté `\frac{\pi}{2}` dans le source ;
- des signes d'intégration, par exemple
$$\int_0^{+\infty} f(x)\mathrm{d}x$$
qui sera noté `\int{0}{+\infty}{f(x)*\d x}` dans le source (remarque : si `x` est une variable, alors `\d x` sera une formule valide, composée dx) ;
- des signes de sommation, par exemple
$$\sum_{k=1}^{n} \frac{1}{k}$$
qui sera noté `\sum{k=1}{n}{\frac{1}{k}}` dans le source.

Il faut remarquer que dans le fichier source, les formules ne respectent pas les mêmes conventions lexicographiques que le texte normal. Ceci oblige à analyser le texte des formules avec un autre analyseur lexical. Nous prendrons la convention suivante: dans le texte, les formules seront écrites encadrées par deux caractères $. Pour les formules complexes, celles-ci formant un paragraphe entier, nous écrirons `\formula{$...$}`. Ainsi le texte suivant:

```
La formule suivante permet de calculer la somme des $n$ premiers
entiers :

\formula{$\sum{k=1}{n}k = \frac{n*(n+1)}{2}$}
```

produira la sortie suivante:

La formule suivante permet de calculer la somme des n premiers entiers :

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

Enfin, noter que le symbole d'intégrale est le caractère numero 242 de la fonte Symbol, et le caractère de sommation est le caractère numéro 229 ; ils devront être écrit dans une taille 3/2 fois plus grande que la taille normale de la fonte des formules, et à une hauteur inférieure de 1/4 fois cette taille. Pour les traits de fractions, vous utiliserez la procédure `draw_line` fournie dans le module `postscr`. Comme les formules en indice ou en exposant, les formules formant les bornes d'intégrale ou de sommes, ou encore les numérateurs et dénominateurs de fractions seront composées dans une fonte de taille 0.7 fois inférieure à la taille normale.

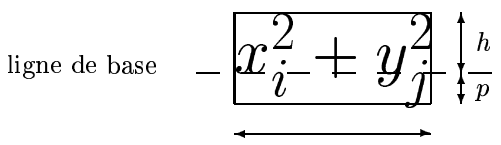
Noter également que ces constructions doivent pouvoir être imbriquées, comme par exemple

$$\frac{1}{\frac{1}{x}+\frac{1}{y}} \text{ ou bien } \frac{\sum_{i=1}^{N} x_i^2}{\left(\sum_{i=1}^{N} x_i\right)^2} \text{ ou encore } \int_{\frac{1}{4}}^{\frac{1}{2}} \frac{e^x}{x^2}\mathrm{d}x$$

4.2.2 Indications pour la composition des formules

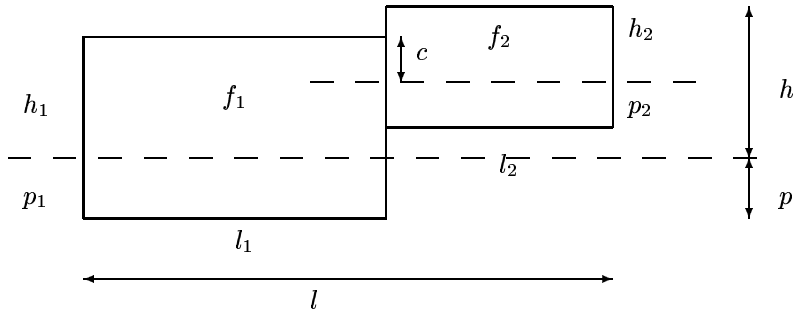
La difficulté de la composition des formules provient de deux choses : en présence d'indice ou d'exposant, les symboles ne sont pas tous écrits à la même hauteur ; et dans les formules complexes, une formule peut être de hauteur arbitraire.

Pour composer les formules, nous procéderons en deux étapes : d'abord nous calculerons la taille de la formule à composer, ce qui demande récursivement de calculer les tailles des formules qui la constitue. La taille d'une formule sera en fait donnée par trois nombres : la *hauteur*, la *profondeur* et la *largeur* ; la hauteur et la profondeur donnant les limites de la formules par rapport à une *ligne de base*, comme indiqué par la figure ci-dessous :



Un *jugement de taille* est un jugement de la forme $t \vdash f \Rightarrow h, p, l$ où le contexte t est un réel donnant la taille courante de la fonte (en mm), f une formule et h, p et l la hauteur, la profondeur et la largeur de la formule f. L'algorithme de calcul de la taille d'une formule sera alors décrit par des règles sémantiques portant sur ce jugement. À titre d'exemple, le

calcul de la taille d'une formule de la forme $f_1 \hat{} f_2$ sera calculée à partir des tailles de f_1 et f_2 d'après le schéma suivant :



La distance c entre le haut de la formule f_1 et la ligne de base de f_2 sera calculée par un certain coefficient k_1 multiplié par la hauteur h_2 de f_2. k_1 sera un paramètre de style, qui pourra être de 0.9 par défaut. La taille de fonte dans laquelle la formule f_2 doit être composée doit être plus petite que la taille de fonte de f_1, on choisira comme taille de fonte pour f_2 un certain coefficient k_2 multiplié par la taille de fonte de f_1. k_2 sera un paramètre de style qui vaudra 0.7 par défaut.

Un principe analogue doit être appliqué pour les formules en indice : la ligne de base de la formule en indice de hauteur h_2 doit être de $k_3 \times h_2$ en dessous de la ligne de base de f_1. k_3 pourra être choisi à 0.5 par défaut. Pour les autres formules complexes, vous devez déterminer des principes analogues.

Enfin, pour les formules simples que sont les variables et les constantes, vous prendrez comme hauteur la hauteur de la fonte, et 0 comme profondeur. la largeur sera calculée à l'aide de la fonction `text_width`.

Ce jugement de taille permet ensuite de définir des règles pour un *jugement de composition* d'une formule, de la forme $x, y, t \vdash f \Rightarrow i$ où dans le contexte, t donne la taille de la fonte et x et y sont les coordonnées d'origine où la formule f doit être composée. i sera une liste d'instructions de composition, qui sont essentiellement des appels à la fonction `output_text`, et occasionnellement à `draw_line`.

4.3 Les tableaux

4.3.1 Syntaxe

Un tableau à m lignes et n colonnes sera décrit dans le texte source sous la forme suivante :

```
\tabular{lcr...}
{case 1,1 & case 1,2 & ... & case 1,n}
...
{case m,1 & case 1,2 & ... & case m,n}
```

Il constitue un paragraphe à part entière, il se termine donc par une ligne vide. Le premier argument de `\tabular` est une séquence d'exactement n lettres parmi `l`, `c` et `r` indiquant que la colonne correspondante sera composée respectivement à gauche, centrée et à droite. le caractère `&` sert de séparateur entre les cases. Une erreur à la composition devra être déclenchée si le nombre de cases données dans l'une des listes de cases n'est pas exactement n. Le nombres de lignes est déterminé uniquement par le nombre de lignes dans le source, et ne peut donc pas donner lieu à une erreur.

Par exemple, la description de haut niveau suivante:

```
\tabular{lcr}
```

```
{Année & semestre & Nombre d'étudiants }
{96-97 & 1 & 27}
{96-97 & 2 & 110}
{97-98 & 1 & 50}
```

devra donner :

Année	semestre	Nombre d'étudiants
96-97	1	27
96-97	2	110
97-98	1	50

4.3.2 Indications pour la composition

Le programme de composition devra composer un tableau en deux étapes : d'abord le calcul de la largeur de chaque colonne et de la hauteur totale du tableau, puis la composition proprement dite. Remarquer que le tableau lui-même devra être centré sur la page. La distance entre le texte des cases et les lignes tracées est un paramêtre de style supplémentaire, par défaut 3mm. L'épaisseur des traits est également un nouveau paramètre de style, par défaut 1mm.

Vous devez donc définir un jugement de taille pour les tableaux, et les règles sémantiques associées, ainsi qu'un jugement de composition.

4.4 Calculs automatiques

Pour réaliser la fonction d'un « tableur », nous désirons permettre à l'utilisateur de saisir des formules à calculer au moment de la composition.

4.4.1 Syntaxe

Par convention, ces formules seront tapées entre deux #. Voici un exemple de source souhaité

```
% moy(y) donne la moyenne des 2e et 3e case de la ligne y
\def{#moy(y):=(C(2,y)+C(3,y))/2#}
% sumcol(y) donne la somme des cases c(x,2)+..+c(x,y) ou x est le
% numero de colonne courante
\def{#sumcol(y):=if y<2 then 0 else C(X,y)+sumcol(y-1)#}
% moycol(y) donne la moyenne des cases c(x,2),..,c(x,y-1)
\def{#moycol(y):=sumcol(y-1)/(y-2)#}

\tabular{lrrr}
{Élève & devoir 1 & devoir 2 & moyenne }
{Agnan & #19# & #19# & #moy(Y)# }
{Alceste & #1# & #2# & #moy(Y)# }
{Clotaire & #0.5# & #0# & #moy(Y)# }
{Eudes & #5# & #8# & #moy(Y)# }
{Rufus & #9# & #10# & #moy(Y)# }
{moyennes & #moycol(Y)# & #moycol(Y)# & #moy(Y)#}
```

et la sortie voulue :

Élève	devoir 1	devoir 2	moyenne
Agnan	19	19	19
Alceste	1	2	1.5
Clotaire	0.5	0	0.25
Eudes	5	8	6.5
Rufus	9	10	9.5
moyennes	6.9	7.8	7.35

Comme pour les formules à composer, les formules à calculer seront analysées avec un analyseur lexical particulier. Le nombre de décimales affichées pour les réels sera un paramètre de style, 2 par défaut.

4.4.2 Sémantique

La sémantique de ces expressions entre # est informellement définie de la façon suivante :

- dans une instruction `\def`, on définit soit une constante, soit une fonction à argument, par une expression arithmético-logique, éventuellement récursive pour les fonctions ;
- dans un tableau, toute expression entre # est remplacée par sa valeur, qui est calculée dans un environnement où sont définies :
  1. les fonctions précédemment définies par des `\def` ;
  2. les constantes `X` et `Y` qui sont respectivement les numéros de colonne et de ligne de la case en cours de calcul (numérotées en commençant à 1) ;
  3. la fonction spéciale `C` à deux arguments donnant pour chaque (x, y) la valeur de la case $C(x, y)$ du tableau en cours de composition.

  Les expressions autorisées seront les expressions arithmético-logiques avec le `if` et le `let` pour les variables locales. Le programme de composition devra bien entendu vérifier que ces expressions sont bien typées.

Pour éviter des boucles dans le calcul des cases, on considèrera que pour calculer la valeur d'une case, on ne peut utiliser que les valeurs des cases d'ordonnée inférieure, ou bien de même ordonnée et d'abscisse inférieure. Les valeurs de chaque case seront donc calculées de haut en bas et de gauche à droite.

D'autre part, on s'interdit de redéfinir une constante ou une fonction, et pour éviter des conflits avec X, Y ou C, on n'autorise comme identificateur de constante ou de fonction uniquement des séquences non vides de lettres minuscules.

Dans votre deuxième rapport, la sémantique informelle décrite ci-dessus devra être formalisée par des jugements de typage et d'évaluation, et des règles sémantiques associées.

Chapitre 5

Un navigateur hypertexte

Dans cette étude de cas, nous allons écrire un programme de navigation dans un *hypertexte*. Un hypertexte est un ensemble de pages de textes (autrement dit des fichiers) qui sont reliées entre elles par des *ancres*, une ancre étant une marque sur un mot du texte qui permet, par sélection de ce mot avec la souris, de faire afficher une autre page de texte.

Un exemple d'hypertexte est donné par les fichiers d'aide de Windows. Un autre exemple à la mode est donné par les fameuses pages du *World Wide Web*, dont la particularité est que les liens peuvent référer à des pages se trouvant physiquement sur une machine située n'importe où dans le monde, à la condition que celle-ci soit reliée au réseau mondial *Internet*.

Sur la toile (mot français pour le *World Wide Web*), les fichiers d'hypertexte doivent respecter un standard international pour que ces pages puissent être visualisées sur n'importe quel type d'ordinateur (PC, Macintosh, station de travail sous UNIX, etc.). Ce standard s'appelle le langage HTML (*HyperText Markup Language*), et dans cette étude nous allons réaliser un outil de visualisation de pages décrites dans ce langage, et de navigation d'une page à l'autre.

Bien sûr, il n'est pas de question d'implanter le langage HTML complet qui comporte trop de fonctionnalités différentes, nous allons implanter les fonctionnalités de base, qui nous permettrons de voir les principes et méthodes à utiliser pour implanter toutes les autres fonctionnalités. Nous commençons dans cette première partie avec une version très basique qui sera à étendre dans la deuxième et la troisième partie.

5.1 Le langage HTML de base

Ce langage permet de donner une description de *haut niveau* d'une page de texte : il décrit la structure du texte, c'est-à-dire pour la première partie de notre étude, comment le texte est découpé en paragraphes, mais il ne dit absolument pas où doivent être placés les sauts de lignes, ce sera le travail du programme de visualisation de déterminer ces sauts de ligne.

Un exemple de fichier HTML est donné figure 5.7.

Le programme de visualisation devra afficher le texte de ce fichier en ouvrant une fenêtre graphique et afficher une page comme montré figure 5.8. La chose importante à remarquer est que les espaces entre les mots et les passages à la ligne du texte source n'ont aucune influence sur le texte de sortie : en effet les passages à la ligne sont calculées de manière à ce que les lignes soient contre les marges gauches, en remplissant les lignes au maximum.

Le texte de l'exemple contient également une ancre, qui est définie par la ligne

```
<a href="fichier.htm">ancres</a>
```

```
<p>
Ceci est un paragraphe de texte écrit dans le langage HTML. il commence
par un << p >> écrit entre les caractères << inférieur >> et << supérieur
>>. il se termine par << /p >> écrit entre ces mêmes caractères.
</p>

<p>
Remarquer que les passages à la ligne du texte
initial n'influent pas du tout sur les passages à la ligne du texte
affiché, ainsi que le nombre    d'espaces     entre     les
      mots      qui sont toujours remplacés par un seul espace
dans le texte affiché.
</p>


<p>
Un fichier HTML peut également contenir des
<a href="fichier.htm">ancres</a> pour pouvoir
naviguer entre les pages de texte.
</p>
```

FIG. 5.1 – *Un exemple d'un fichier HTML.*

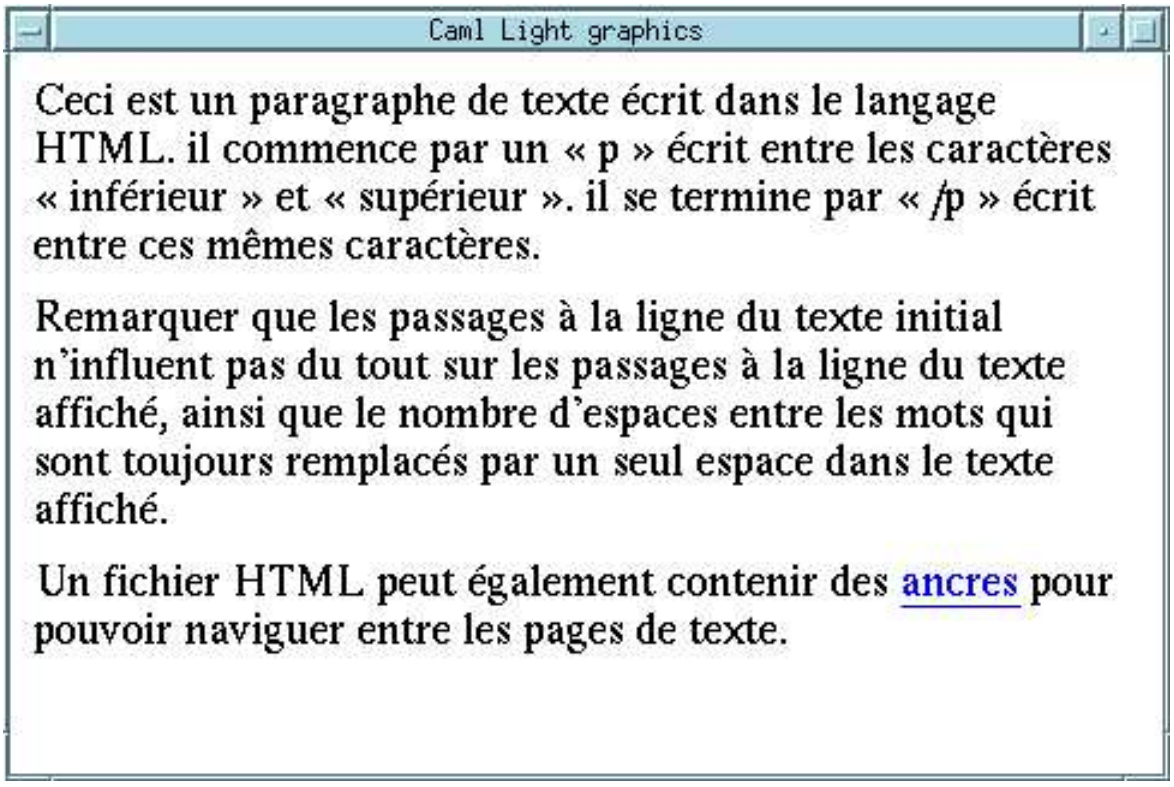


FIG. 5.2 – *Le fichier HTML comme affiché par le navigateur.*

Dans le texte affiché, le mot sur lequel l'ancre est placé, c.-à-d. « ancres » dans l'exemple, sera affiché d'une manière spéciale de manière à ce que le lecteur voit que ce mot est marqué par une ancre : il sera souligné et, sur un écran couleur, écrit dans une autre couleur, par exemple bleue. Le *lien* (`"fichier.htm"` dans l'exemple) n'est pas visible sur la page affichée, mais quand l'utilisateur placera la souris sur le texte de l'ancre et cliquera sur le bouton gauche de la souris, le contenu de la fenêtre graphique sera effacé et la page indiquée par le lien de l'ancre sera affichée.

Le programme se terminera quand l'utilisateur pressera la touche `q`. Un exemple de navigateur est à votre disposition pour mieux comprendre les fonctionnalités voulues du programme :

- pour le DEUG MIAS : dans `f:\caml\m3mias\html`
- pour la MIAGE : dans `~treinen/exemplaire/partie1`

Description précise du langage

Nous donnons maintenant une description précise du langage. Pour votre premier rapport, vous devrez traduire cette description en une grammaire formelle.

Un fichier HTML consiste en une suite (éventuellement vide) de *paragraphes*. Chaque paragraphe commence par `<p>` et finit par `</p>`. Un paragraphe consiste en une suite (éventuellement vide) *d'éléments*, séparés par un ou plusieurs caractères *d'espacement* (espace, tabulation ou saut de ligne). Un élément d'un paragraphe peut être :

- soit un *mot*, formé d'une suite non vide de caractères autres que les caractères d'espacements, `<`, `>` et `"` ;
- soit une *ancre*, qui est de la forme `<a href=`*fichier*`>`*mot*`</a>`, où *fichier* est un nom de fichier entre guillemets.

On appelera *fichier* le *lien* et *mot* le *texte* de cette ancre. Le nom de fichier pourra être formé de n'importe quel caractères sauf `"`. On autorisera aussi plusieurs caractères d'espacement entre `<a`, `href`, `=` et le nom du fichier. Les « mots-clés » `a`, `p` et `href` devront être acceptés aussi bien en minuscules qu'en majuscules.

Pour la taille de la fenêtre graphique, des marges et l'espacement vertical entre les paragraphes on choisira des valeurs raisonnables. Quand un fichier est trop long pour être affiché dans la fenêtre graphique, l'affichage s'arrêtera quand la fenêtre est pleine.

Il y a, par contre, des erreurs plus importantes : un fichier peut ne pas exister, ou bien ce fichier peut ne pas être conforme à la syntaxe HTML. Dans ce cas, le navigateur superposera sur la dernière page affichée un petit rectangle d'une couleur de fond différente du texte normal, et contenant le message d'erreur. Le rectangle avec le message d'erreur sera alors supprimé quand l'utilisateur pressera une touche du clavier.

5.2 Énumérations et tableaux

Cette deuxième partie de l'étude est une suite de la première partie. Vous utiliserez pour cette partie les outils CAMLLEX et CAMLYACC présentés en cours, qui nous permettent d'aborder un langage HTML plus riche qu'il n'était possible pour une analyse lexicale et syntaxique programmée avec des automates standard. Pour cette partie, le programme devra obligatoirment être écrits avec des vrais modules compilés.

Le langage HTML considéré dans la première partie de l'étude ne contenait qu'un seul type de paragraphe : une liste de mots et d'ancres, délimités par `<p>` et `</p>`. Le langage HTML étendu de cette deuxième partie de l'étude comprend maintenant deux nouveaux types de paragraphes : les tableaux et les énumérations.

5.2.1 Les tableaux

Un exemple de fichier contenant un tableau est donné figure 5.3, et l'affichage attendu est donné figure 5.4.

De manière générale, un tableau commence par une déclaration de la forme `<table cols="`w`">` et se termine par `</table>`, où le mot w est la spécification des alignements de chaque colonne. Cette spécification ne doit contenir que les lettres `l`, `c` et `r`, indiquant respectivement une justification à gauche, centrée ou à droite de la colonne correspondante.

Le contenu du tableau est donné ligne par ligne, une ligne commençant par `<tr>`, et dans chaque ligne, chaque case commence par `<td>`. Le contenu des cases est une liste quelconque de mots ou d'ancres.

Une condition sémantique est que toutes les lignes doivent avoir le même nombre de cases, et ce nombre doit être égal à la longueur du mot qui spécifie les alignements des colonnes.

5.2.2 Les énumérations

Une énumération est une liste numérotée de portions de texte, où la numérotation dans le bon ordre doit être réalisée automatiquement par le navigateur.

Un exemple de fichier contenant une énumération est donné figure 5.5, et l'affichage attendu est donné figure 5.6.

Les énumérations peuvent donc être imbriquées et elles peuvent contenir également des tableaux. Les différents points d'une énumération sont numérotés par défaut à partir de 1. Il est aussi possible de spécifier le premier numéro n utilisé par une énumération, dans ce cas-là l'énumération commence par `<ol start=`n`>`.

D'une manière générale, une énumération commence donc avec `<ol>` ou bien `<ol start=`n`>` et se termine par `</ol>`. Chaque point d'une énumération commence par `<li>`. Le contenu de chaque point est une suite quelconque de paragraphes simples, tableaux ou énumérations.

5.2.3 Remarques générales sur le langage

Pour simplifier, on n'autorise maintenant les mots clefs (`href`, `start`, etc.) qu'en minuscule. Rappelons que les énumérations et les tableaux peuvent, comme les paragraphes de textes, contenir des ancres qui sont à gérer comme les autres.

Un exemple de fichier HTML montrant tous les extensions considérées dans cette deuxième partie de l'étude est donné figure 5.7, et le résultat attendu est donné figure 5.8.

```
<p>Le paragraphe qui suit est un tableau :</p>

<table cols="llrc">
<tr>
  <td>Nom du film
  <td>Réalisateur
  <td>Année
  <td>Distribution
<tr>
  <td>L'Atalante
  <td>Jean <a href="vigo.htm">Vigo</a>
  <td>1934
  <td>Michel Simon, Jean Dasté, Dita Parlo
<tr>
  <td>Les 400 coups
  <td>François <a href="truffaut.htm">Truffaut</a>
  <td>1959
  <td>Jean-Pierre Léaud, Claire Maurier
</table>
```

FIG. 5.3 – *Un exemple de tableau.*

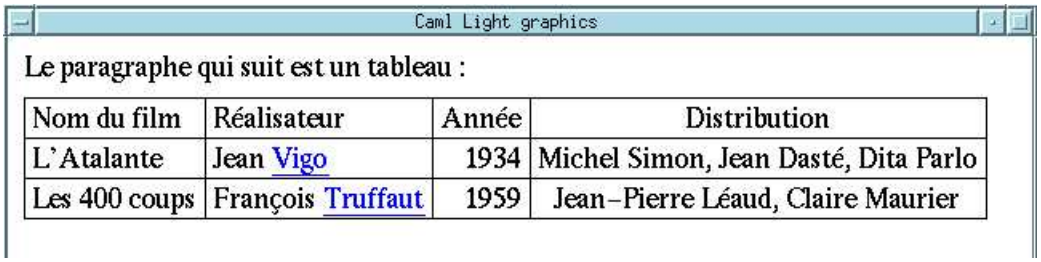


FIG. 5.4 – *Le tableau comme affiché par le navigateur.*

```
<p>Le paragraphe qui suit est une énumération</p>

<ol>

<li><p>Ceci est le premier point de l'énumération.</p>

<li><p>Voilà le deuxième point de l'énumération. Noter que la marge
gauche de tout le texte de chaque point de l'énumération est décalé
sur la droite.</p>

<p>Chaque point d'une énumeration peut contenir plusieurs paragraphes,
ainsi que des <a href="ancre.htm">ancres.</a></p>

<li><p>Il est également possible de faire des énumérations imbriquées
</p>

<ol>
<li><p>Le premier point de la sous-énumération.</p>
<li><p>Le deuxième point de la sous-énumération.</p>
</ol>

<li><p>la numérotation reprend automatiquement à l'endroit où elle
s'était arrêtée.</p>

</ol>
```

FIG. 5.5 – *Un exemple d'énumération.*

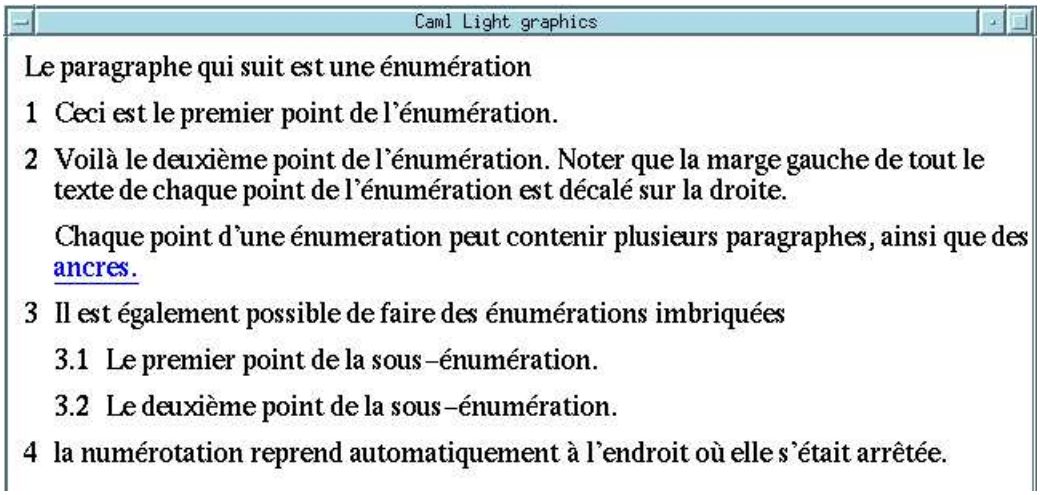


FIG. 5.6 – *L'énumération comme affichée par le navigateur.*

```
<p>Quelques données sur les cours d'Informatique du semestre M3 du
DEUG MIAS</p>

<ol>

<li><p>Le nouveau programme a commencé en septembre 1996</p>

<li><p>Le tableau suivant indique l'évolution du nombre d'étudiants :</p>

<table cols="lcr">
<tr>
<td>Année <td>Semestre <td>Nombre d'étudiants
<tr>
<td>96-97    <td>1         <td>27
<tr>
<td>96-97    <td>2        <td>110
<tr>
<td>97-98    <td>1         <td>50
<tr>
<td>97-98    <td>2         <td>70
</table>
</ol>

<p>On peut faire les remarques suivantes :</p>

<ol start=1997>
<li><p>beaucoup plus d'étudiants au second semestre qu'au premier</p>
<li><p>beaucoup plus équilibré</p>
</ol>
```

FIG. 5.7 – *Un exemple d'un fichier HTML avec toutes les extensions.*

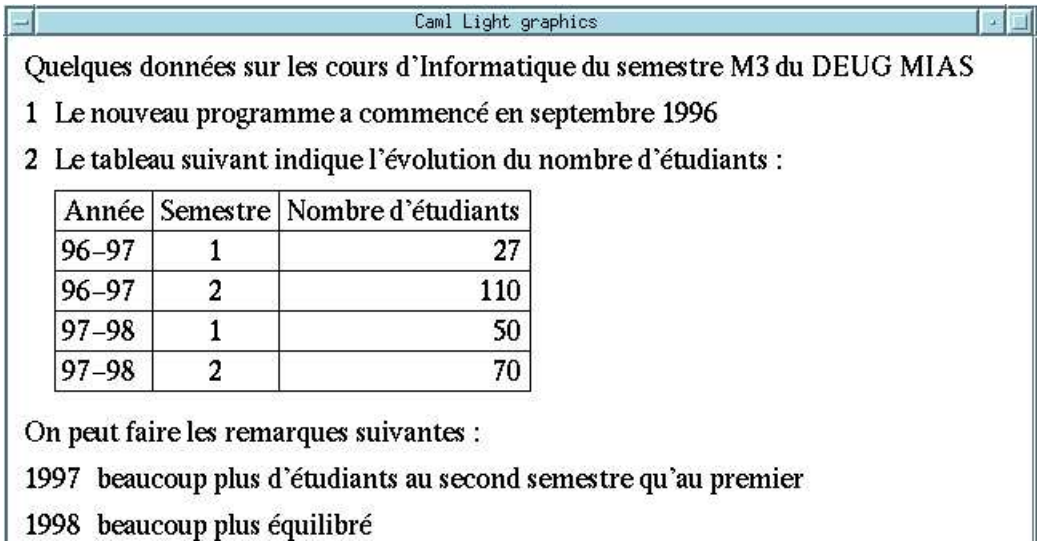


FIG. 5.8 – *Le fichier HTML complet comme affiché par le navigateur.*

=======

Table des figures

Index