Une analyse syntaxique d'ASN.1 : 1990 en Caml Light

Christian Rinderknecht

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

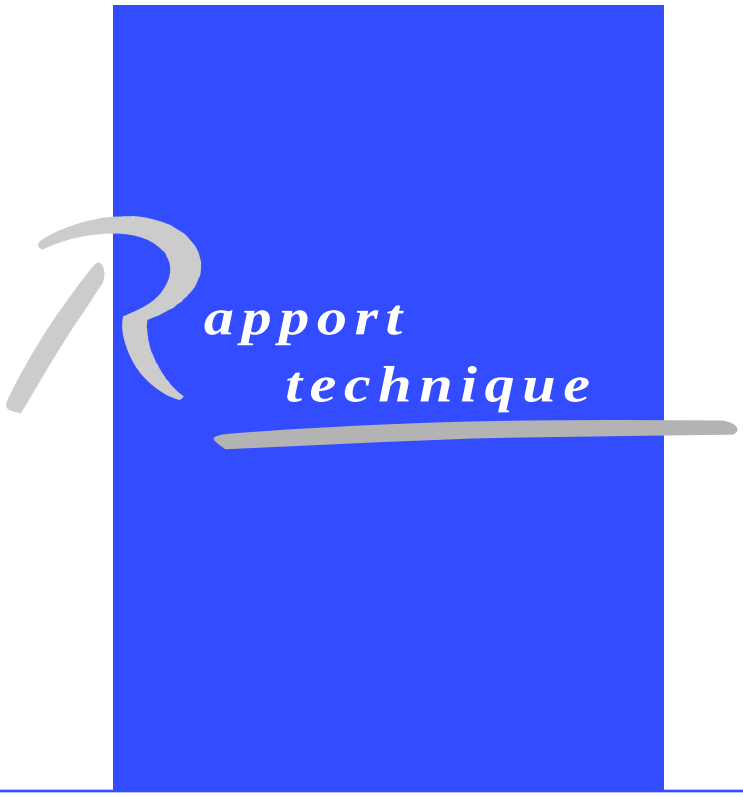
Une analyse syntaxique d'ASN.1 :1990 en Caml Light

Christian Rinderknecht

N ° 171
Avril 1995

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel



ISSN 0249-6399

Une analyse syntaxique d'ASN.1 :1990 en Caml Light

Christian Rinderknecht

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet CRISTAL

Rapport technique n ° 171 — Avril 1995 — 228 pages

Résumé : ASN.1 est un langage de spécification de protocoles normalisé par l'ISO et utilisé fréquemment dans les télécommunications. Il permet de décrire et de regrouper en modules les types et les valeurs que sont susceptibles d'échanger des applications. L'ambiguïté de la grammaire d'ASN.1, dans sa version 1990, avait jusqu'à présent contraint les concepteurs de compilateurs à prendre des libertés avec la norme. Nous montrons dans ce document comment il est possible de transformer la grammaire en une grammaire équivalente LL(1). Un analyseur syntaxique a été réalisé en Caml Light, un langage fonctionnel typé de la famille ML. Il apporte la sécurité du typage statique fort et l'expressivité de la pleine fonctionnalité, deux atouts majeurs pour la construction d'un arbre de syntaxe abstraite et le traitement des macros « à la volée », conduisant ainsi à *un analyseur en une passe*. Une méthode pour réaliser des analyseurs syntaxiques en Caml Light est de plus présentée en détail, ainsi que le code abondamment commenté de celui réalisé pour ASN.1.

Mots-clé : ASN.1, langages de spécification, protocoles ISO, Caml, ML, langages fonctionnels, analyse syntaxique.

(Abstract: pto)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Parsing ASN.1 :1990 with Caml Light

Abstract: ASN.1 is a specification language for network protocols, normalized by the ISO and frequently used in telecommunications. It allows the modular description of the types and values that may be exchanged between two applications. The ambiguity of the ASN.1 grammar, in its 1990 version, led the compilers conceptors to compromise with the normative document. We show in this work how to transform the grammar in an LL(1) equivalent one. A parser has been implemented in Caml Light, a typed functional language of the ML family. It offers the security of strong static type-checking and the expressivity of full functionality, very useful for abstract-syntax tree building and on-the-fly macro-processing, leading to a *one-pass parser*. A method for parsers writing in Caml Light is also given, and the fully commented code of the ASN.1 parser.

Key-words: ASN.1, specification languages, ISO protocols, Caml, ML, functional languages, parsing.

MAÎTRE DE PHILOSOPHIE. — On les peut mettre [les paroles] premièrement comme vous avez dit : *Belle Marquise, vos beaux yeux me font mourir d'amour*, ou bien : *D'amour mourir me font, belle Marquise, vos beaux yeux.* Ou bien : *Vos yeux beaux d'amour me font, belle Marquise, mourir.* Ou bien : *Mourir vos beaux yeux, belle Marquise, d'amour me font.* Ou bien : *Me font vos yeux beaux mourir, belle Marquise, d'amour.*

MONSIEUR JOURDAIN. — Mais de toutes ces façons-là laquelle est la meilleure ?

MAÎTRE DE PHILOSOPHIE. — Celle que vous avez dite : *Belle Marquise, vos beaux yeux me font mourir d'amour.*

Molière. *Le bourgeois gentilhomme,* Acte II, Scène IV.

Remerciements

Ce travail a été mené à bien en collaboration avec Olivier Dubuisson, Joaquín Keller et Frédéric Hugot, du Centre National d'Études des Télécommunications de France Télécom (CNET) à Lannion. Olivier et Joaquín m'ont accueilli chaleureusement et fait profiter de leur expérience par leurs conseils avisés.

Bernard Lorho m'a honoré de son intérêt constant pour cette réalisation et Michel Mauny, qui en a suivi le déroulement, m'a toujours fait confiance et laissé une grande liberté de manœuvre.

L'aide de Daniel de Rauglaudre, grand dompteur de flux devant l'Éternel, m'a été précieuse.

Valérie Ménissier-Morain et Émilie Sayag m'ont apporté leur soutien amical et technique (dans les labyrinthes de TEX, Emacs, Unix...). L'imprimeur de grammaires d'Émilie m'a fait gagner un temps précieux dans les versions préliminaires de ce document et l'expérience de Valérie a été un atout hors pair tous azimuts.

La qualité typographique de ce document doit énormément à l'indenteur de code Caml Light mis au point par Michel Mauny. Il permet de marier Caml et TEX avec élégance et souplesse.

Mis amigos Manuel Castro y Oscar Barrientos me apoyaron con paciencia, cordura y tacto en aquellos momentos difíciles.

Last but not least, *quiero agradecer a Javier Barón por darle un pleno significado a la palabra amistad, y hacérmelo compartir día tras día. Soy muy dichoso.*

Table des matières

Introduction

Les spécifications de services et protocoles OSI[1] sont la base de la conception et de l'implantation de systèmes informatiques hétérogènes distribués. Ces normes de communication OSI sont spécifiées en langage naturel, les données transmises sur les couches 6 et 7 du modèle sont décrites à l'aide de la notation ASN.1.

La notation de syntaxe abstraite numéro un (*Abstract Syntax Notation One*, en anglais) est une norme conjointe de l'ISO [3] et du CCITT (X.208) apportant une description standardisée de types de données et de valeurs. En ce sens elle peut être comparée aux définitions de types et de valeurs dans des langages de programmation comme Pascal ou C. Elle dispose d'un certain nombre de types prédéfinis tels que les entiers, les réels, les booléens, etc., et permet de définir des types structurés tels des enregistrements, des ensembles ordonnés ou non d'éléments de même type, ou bien encore des listes de types alternatifs (correspondant aux enregistrements variables en Pascal).

Une étiquette (*tag* en anglais) peut être utilisée pour distinguer différents éléments alternatifs ou optionnels dans un ensemble ; fondamentalement elle identifie le type de chaque valeur et chaque valeur est transmise sur la ligne de communication conjointement avec son étiquette (de type).

Ce rapport présente le travail réalisé sur la grammaire du langage de spécification ASN.1, dans sa version ISO 1990 [3]. Nous montrons que l'on peut obtenir une grammaire ASN.1 ayant la propriété LL(1)[2], grâce à une suite de transformations à partir de la forme normalisée, sans faire aucun compromis avec celle-ci. Une grammaire peut être ici considérée comme étant un ensemble d'équations rationnelles ayant pour (plus petite) solution un langage non contextuel. Chaque transformation est donc décrite en fonction d'opérations rationnelles (union, produit et étoile) garantissant l'invariance du langage engendré par la grammaire résultante.
Les macros ASN.1 sont traitées mais pas dans toute leur généralité, étant donné que cela est irréalisable. Une analyse au vol est présentée, ainsi que son intégration orthogonale à l'analyseur de base (qui ne gère pas les macros). L'analyseur complet fonctionne donc en *une passe*.

Tout d'abord nous présentons une grammaire LL(1) pour le lexique d'ASN.1, obtenue à partir des indications en langue naturelle du document ISO. Ensuite vient l'étude de la syntaxe d'ASN.1. La contrainte qui guidait le choix des transformations était de rendre analysable de façon descendante la grammaire finale, ce qui impliquait qu'elle devait être non ambiguë. Or, la question de l'ambiguïté d'une grammaire étant un problème indécidable, il a fallu faire des choix explicites *ad hoc* pour lever les ambiguïtés rencontrées. De plus, nous ne savions même pas *a priori*, en supposant éliminées ces ambiguïtés, s'il existait

[1] *Open Systems Interconnection*
[2] Cf. section 4

une grammaire LL(1) pour ASN.1. Ces deux raisons rendirent nécessaire un travail « à la main ». D'autre part, étant données deux grammaires, le problème de l'égalité des langages engendrés par celles-ci est aussi indécidable [1]. Par conséquent, la preuve de cette égalité se fera par la présentation *in extenso*, étape par étape, de chaque transformation syntaxique. Pour faciliter la compréhension, la grammaire initiale sera découpée en sections dont nous suivrons l'évolution. La preuve de la propriété LL(1) est donnée ensuite. Les errata[3] de la norme ont été pris en compte dès les étapes initiales des transformations, leur validité étant rétroactive. Ils sont signalés dès leur première apparition.

Ce travail a abouti à la réalisation d'un analyseur syntaxique complet, entièrement écrit dans le langage de programmation *Caml Light*, issu de la recherche au sein de l'INRIA [8, 5], et disponible en *ftp* anonyme. C'est un langage fonctionnel de la famille ML, dont le typage statique fort garantit la bonne construction et la cohérence des valeurs lors de l'exécution. Sa compilation est un gage de rapidité du code produit. Sa grande portabilité (des versions pour PC et Macintosh sont disponibles) permet une large diffusion des applications. D'autre part il permet d'écrire des analyseurs syntaxiques grâce à une structure de données spéciale : les *flux* [7]. Les flux se comportent comme des listes paresseuses[4] avec une sémantique destructive. Cela signifie que lorsque l'on accède à l'élément de tête, celui-ci est alors évalué (d'où la paresse) et est ensuite obligatoirement ôté (d'où la destruction). Un mécanisme de filtrage spécifique pour ces flux permet de décrire ce qui se « passe » lors de l'analyse syntaxique, et ce avec une syntaxe Caml Light proche d'une spécification de grammaire standard. L'avantage par rapport à des outils comme YACC est que nous continuons à profiter de la pleine fonctionnalité et d'une sémantique formelle (« nous savons ce *qui* est analysé et calculé, *quand* c'est analysé et calculé et *comment* c'est analysé et calculé »). L'inconvénient par rapport à YACC étant que la grammaire doit être analysable de façon prédictive et descendante (cf. section 5).

Une méthode systématique pour écrire des analyseurs syntaxiques en Caml Light à partir d'une grammaire LL(1) est de plus présentée. L'analyseur ainsi produit n'est en rien une « boîte noire », la syntaxe particulièrement lisible des filtres de flux et l'uniformité du code y contribuant grandement.

Toutes les caractéristiques précitées de Caml Light en font un atout sérieux de sécurité pour la validation de spécifications de protocoles.

[3] *Technical corrigenda.*
[4] donc potentiellement infinies

1 Notations

La notation pour les grammaires adoptée dans ce document est *grosso modo* la BNF[5], plus quelques ajouts (notamment d'opérateurs rationnels).

1.1 Lexique ASN.1

- Les identificateurs de terminaux apparaissent en minuscules.
- Les mots-clefs sont en majuscules.
- Les identificateurs de non-terminaux sont la concaténation d'identificateurs en minuscules, débutant chacun par une majuscule.
- Les extraits de syntaxe concrète (terminaux) sont entre guillemets — si ces derniers font partie de la syntaxe concrète, alors ils sont précédés d'une contre-oblique[6].

1.2 Grammaires

Nous appellerons « règle de production » (ou en abrégé « règle »), le couple formé par un non-terminal et tous les mots qui peuvent en être dérivés en une étape.

Nous nommerons « membre droit d'une règle de production » (ou en abrégé « membre droit ») un des mots (constitué de terminaux ou non) que l'on peut dériver en une étape. Par exemple, $X_0\ X_1\ \ldots X_n$ est un membre droit de la règle de production X dans :
$X \rightarrow X_0\ X_1\ \ldots X_n \mid \ldots$

Nous baptiserons « production » l'ensemble des membres droits d'une règle de production.

Structure La grammaire est découpée en sections qui sont divisées en sous-sections séparées par une ligne. Le but de ce découpage est de regrouper des règles qui ont une sémantique voisine ou qui contribuent à une même sémantique. Une sous-section A séparée par une ligne double d'une sous-section B signifie que B est une transformée de A. L'ordre des règles au sein d'une même sous-section est significatif : c'est un parcours en largeur à partir d'une règle d'entrée, en considérant les productions dans l'ordre d'écriture. L'ordre entre les règles d'entrée n'est pas significatif.

Les règles d'entrée sont des règles qui sont appelées en dehors de la sous-section où elles sont définies. Si les appels sont restreints à la section courante, la règle est dite *locale* (à la section), s'ils sont limités au dehors de la section définissante, elle est dite *globale*, et s'il existe des appels à la fois dans la section courante et dans les autres sections, elle est qualifiée de *mixte*. Dans le premier cas, l'identificateur du non-terminal définissant

[5] *Backus-Naur Form*
[6] *backslash*

apparaîtra en italique, dans le deuxième en souligné, et dans le dernier en italique souligné. L'axiome sera en gras. Il peut y avoir plusieurs règles d'entrée dans une sous-section.

Commentaires Dans la présentation des grammaires intermédiaires, les commentaires seront dans des boîtes, après les règles. Il est impératif de les lire dans l'ordre d'apparition dans la sous-section, car ils peuvent décrire des transformations composées (d'où le séquencement). Pour la même raison, il faut lire les sections dans l'ordre d'apparition. D'autre part, les règles créées par une transformation seront données entre parenthèses.

Conventions En transformant une règle se créent de nouvelles règles pour lesquelles il est difficile de trouver un identificateur pertinent suggérant sa sémantique. Dans ce cas, chacune de ces règles aura pour identificateur la concaténation de celui de la règle appelante (parfois abrégé), plus un préfixe et/ou un suffixe, souvent numéroté. Dans les exemples contenus dans ce document, nous adopterons en outre les conventions lexicographiques supplémentaires suivantes :

- La production vide est notée ε.
- Une lettre latine minuscule représente toujours un terminal.
- Une lettre latine majuscule représente toujours un non-terminal.
- Une lettre grecque minuscule représente toujours une concaténation quelconque de terminaux et de non-terminaux.
- Nous noterons $\mathrm{A} \Longrightarrow \alpha$ la relation « A produit α ».

1.3 Opérateurs rationnels

Les opérateurs rationnels suivants ont été rajouté à la notation BNF, pour gagner en compacité et lisibilité (il ne s'agit pas d'un gain d'expressivité) : α^*, α^+, $[\alpha]$, $\{\ \mathrm{A\ a} \ldots\}^*$, $\{\ \mathrm{A\ a} \ldots\}^+$. Chaque occurrence de ces expressions peut être remplacée par un non-terminal dont la règle définissante est spécifique. Comme nous le verrons dans la partie consacrée à la vérification de la propriété LL(1) (cf. 4), nous pourrons même considérer qu'il n'y a pas de partage de ces règles. Voici le tableau donnant leur sémantique.

Notation	Définition	Contrainte de validité
$\mathrm{X} \to \alpha^*$	$\mathrm{X} \to \alpha\ \mathrm{X} \mid \varepsilon$	$\neg(\alpha \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \to \alpha^+$	$\mathrm{X} \to \alpha\alpha^*$	$\neg(\alpha \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \to [\alpha]$	$\mathrm{X} \to \alpha \mid \varepsilon$	$\neg(\alpha \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \to \{\mathrm{A\ a} \ldots\}^*$	$\mathrm{X} \to \varepsilon \mid \mathrm{A\ (a\ A)}^*$	$\neg(\mathrm{A} \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \to \{\mathrm{A\ a} \ldots\}^+$	$\mathrm{X} \to \mathrm{A\ (a\ A)}^*$	$\neg(\mathrm{A} \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \to \{[\mathrm{A}]\ \mathrm{a} \ldots\}$	$\mathrm{X} \to \varepsilon \mid \mathrm{A\ (a\ [A])}^* \mid \mathrm{(a\ [A])}^+$	$\neg(\mathrm{A} \overset{*}{\Longrightarrow} \varepsilon)$

Les définitions choisie sont LL(1) et les contraintes de validité permettent une définition directe des fonctions d'analyse en Caml Light (type des arguments adéquat).

2 Transformations de grammaires

Ce sont des applications de propriétés de base des expressions rationnelles, exprimées avec des notations un peu différentes dans le cadre des grammaires.

La factorisation forme des préfixes ou des suffixes (ou les deux à la fois) d'un ensemble de productions d'une même règle.

Préfixe $X \to \beta\gamma_1 \mid \beta\gamma_2 \mid \ldots \mid \beta\gamma_n \mid B$ *devient* $\begin{array}{lcl} X & \to & \beta\ Y \mid B \\ Y & \to & \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n \end{array}$

Suffixe $X \to \alpha_1\beta \mid \alpha_2\beta \mid \ldots \mid \alpha_n\beta \mid B$ *devient* $\begin{array}{lcl} X & \to & Y\ \beta \mid B \\ Y & \to & \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \end{array}$

Bifixe $X \to \alpha\beta_1\gamma \mid \alpha\beta_2\gamma \mid \ldots \mid \alpha\beta_n\gamma \mid B$ *devient* $\begin{array}{lcl} X & \to & \alpha\ Y\ \gamma \mid B \\ Y & \to & \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{array}$

Nous pouvons appliquer ces factorisations si $n = 1$, ou sur une partie seulement des productions factorisables.

La réduction regroupe des productions d'une même règle pour former un nouveau non-terminal :

$X \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ *devient* $\begin{array}{lcl} X & \to & Y \mid \alpha_{i+1} \mid \alpha_{i+2} \mid \ldots \mid \alpha_n \\ Y & \to & \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_i \end{array}$

L'élimination Toute règle devenue inutile (typiquement à la suite d'une expansion globale) peut être supprimée. Une règle d'entrée globale ou mixte, devenue inutile au sein d'une section, n'y figurera plus par la suite — ce qui ne signifie pas son élimination de la grammaire ! Nous pouvons aussi éliminer les productions redondantes au sein d'une même règle pour n'en conserver qu'une.

Remarque 1 La transformation suivante n'est *pas* une élimination :
$X \to X \mid A$ *devient* $X \to A$
Cf. le paragraphe consacré au lemme d'Arden (2.1).

Remarque 2 L'élimination suivante est licite :
$X \to \texttt{lower}_{val} \mid \texttt{lower}_{id}$ *devient* $X \to \texttt{lower}$

Le renommage Pour des raisons de lisibilité, les identificateurs peuvent être renommés globalement dans toute la grammaire. Nous tâcherons d'éviter cette manipulation en choisissant dès le départ des noms adaptés pour éviter les collisions.

L'expansion peut être totale, partielle, préfixe, suffixe ou globale.

Totale Elle consiste à substituer textuellement tout le membre droit d'une règle à *une* occurrence du non-terminal correspondant. Par exemple :

```
A → a B C          A → a b A C
  | B                | a b C
B → b A     devient  | B
  | b              B → b A
C → c                | b
                   C → c
```

Ce sera l'expansion par défaut, en l'absence de qualificatif.

Globale C'est une expansion **totale** d'une règle pour toutes les occurrences possibles dans toute la grammaire (cette règle peut alors être éliminée). La présentation des transformations se faisant par modules, le lecteur pourrait être dérouté par cette portée globale. Concrètement, une expansion globale équivaudra à une expansion totale appliquée à toute la sous-section, suivie d'une élimination ; sinon une note sera fournie.

Partielle C'est la composition d'une expansion **globale**, d'une factorisation bifixe partielle et d'un renommage. En résumé :

$$\begin{array}{lcl} Z & \to & \alpha\ A\ \beta \\ A & \to & \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n \end{array} \quad \textit{devient} \quad \begin{array}{lcl} Z & \to & \alpha\gamma_1\beta \mid \alpha\gamma_2\beta \mid \ldots \mid \alpha\gamma_{i-1}\beta \mid \alpha\ A\ \beta \\ A & \to & \gamma_i \mid \gamma_{i+1} \mid \ldots \mid \gamma_n \end{array}$$

Préfixe C'est la composition d'une expansion **globale** d'une règle factorisable à gauche, d'une factorisation bifixe et d'un renommage. Schématiquement :

$$\begin{array}{lcl} Z & \to & \alpha\ A\ \beta \\ A & \to & \gamma\alpha_1 \mid \gamma\alpha_2 \mid \ldots \mid \gamma\alpha_n \end{array} \quad \textit{devient} \quad \begin{array}{lcl} Z & \to & \alpha\gamma\ A\ \beta \\ A & \to & \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \end{array}$$

Suffixe C'est la composition d'une expansion **globale** d'une règle factorisable à droite, d'une factorisation bifixe et d'un renommage. C'est-à-dire :

$$\begin{array}{lcl} Z & \to & \alpha\ A\ \beta \\ A & \to & \alpha_1\gamma \mid \alpha_2\gamma \mid \ldots \mid \alpha_n\gamma \end{array} \quad \textit{devient} \quad \begin{array}{lcl} Z & \to & \alpha\ A\ \gamma\beta \\ A & \to & \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \end{array}$$

L'option C'est le cas particulier de l'expansion **partielle** d'une production vide, suivie d'une « mise entre crochets ». Par exemple :

```
A → a B C          A → a [B] C
  | B                | [B]
B → b A     devient B → b A
  | ε              C → c
C → c
```

De même, nous assimilerons à une option (par raccourci) la transformation :

```
X → A                      X → [A]
  | b                        | b
A → { B "," ... }*  devient A → { B "," ... }+
  | [a B c]                  | a B c
  | a                        | a
```

Dans le but de faciliter la preuve de la propriété LL(1) de la grammaire finale, nous appliquerons l'option chaque fois que cela sera possible. Ainsi, la forme résultante ne contiendra plus de productions vides *explicites* (attention : nous ne faisons *pas* disparaître ces productions !). En d'autres termes, nous n'autorisons aucune règle à produire explicitement ε.

Notons que les réciproques des factorisations, des réductions, des expansions et des options sont aussi des transformations licites.

2.1 Le lemme d'Arden

Supposons que nous ayons une équation rationnelle de langages de la forme
$X = \alpha\ X + \beta$, avec $\varepsilon \notin \alpha$, et où $+$ représente la disjonction (|). Alors le lemme d'Arden affirme que $X = \alpha^*\beta$ est la solution unique de l'équation. Si $\varepsilon \in \alpha$, alors $X = \alpha^*(\beta + \gamma)$ est solution, quelque soit γ (qui peut même être un langage contextuel). Nous avons donc une infinité de solutions, et nous choisirons dans ce cas le plus petit langage solution («le point fixe minimal») : $X = \alpha^*\beta$. En résumé, nous autoriserons toujours la transformation : $X \to \alpha\ X \mid \beta$ *devient* $X \to \alpha^*\beta$.
Nous avons ainsi le cas particulier : $X \to X \mid \beta$ *devient* $X \to \beta$. Nous nommerons « Arden » toutes les transformations élémentaires qui impliquent l'application du lemme d'Arden :

$X \to \alpha\ X \mid \varepsilon$ *ou* $X \to X\ \alpha \mid \varepsilon$	*devient*	$X \to \alpha^*$
$X \to \alpha\ X \mid \alpha$ *ou* $X \to X\ \alpha \mid \alpha$	*devient*	$X \to \alpha^+$
$X \to A \mid A\ a\ X$	*devient*	$X \to \{A\ a \ldots\}^+$
$X \to \varepsilon \mid A \mid A\ a\ X$	*devient*	$X \to \{A\ a \ldots\}^*$

Remarque 1 Nous avons : $X \to X\ \alpha \mid \beta$ *devient* $X \to \beta\alpha^*$.

Remarque 2 Le lemme d'Arden permet d'écrire des égalités non triviales, comme par exemple : $(a + cb^*d)^* = a^* + a^*c(b + da^*c)^*da^*$.

2.2 Ambiguïtés syntaxiques

L'ambiguïté d'une grammaire est toujours un problème épineux pour l'analyse du langage qu'elle engendre. Une grammaire est ambiguë si et seulement si on peut construire deux

dérivations différentes pour un même mot du langage. Une condition suffisante d'ambiguïté est la double récursivité d'une règle.
Quand la double récursivité est au sein d'une même production, nous pouvons parfois régler ce problème en appliquant le lemme d'Arden. Un cas typique où apparaît cette double récursivité est celui des grammaires d'expressions arithmétiques, où des opérateurs sont binaires et infixes : E → E "+" E | E "∗" E | "(" E ")" | **id**.
Si la double récursivité est à gauche et à droite, sur deux productions, nous pouvons procéder comme suit. Supposons : Z → A Z | Z B | C
Le lemme d'Arden donne ici : Z → A Z B^* | C B^*
Or il est trivial que : X → α X β | γ *devient* $\forall n \geq 0$.X → $\alpha^n \gamma \beta^n$
D'où : $\forall n \geq 0$.Z → A^n (C B^*) $(B^*)^n$ *qui se simplifie en* Z → A^* C B^*
En réécrivant Z → A^+ C B^* | C B^* *il vient* Z → A Z | C B^*
En résumé, nous retiendrons la transformation suivante :

Z → A Z | Z B | C *devient* Z → A Z | C B^*

3 Transformations de la grammaire d'ASN.1 :1990

3.1 Modules

3.1.1 Étape 0

Nous présentons d'abord la partie correspondant à la spécification des modules ASN.1. Nous introduisons le découpage et les regroupements de règles.

ModuleDefinition	→	ModuleIdentifier DEFINITIONS TagDefault “::=” BEGIN ModuleBody END

ModuleIdentifier	→	`modulereference` AssignedIdentifier
AssignedIdentifier	→	ObjectIdentifierValue
	|	ε
ObjectIdentifierValue	→	“{” ObjIdComponentList “}”
	|	“{” DefinedValue ObjIdComponentList “}”
ObjIdComponentList	→	ObjIdComponent
	|	ObjIdComponent ObjIdComponentList
ObjIdComponent	→	NameForm
	|	NumberForm
	|	NameAndNumberForm
NameForm	→	`identifier`
NumberForm	→	`number`
	|	DefinedValue
NameAndNumberForm	→	`identifier` “(” NumberForm “)”

TagDefault	→	EXPLICIT TAGS
	|	IMPLICIT TAGS
	|	ε

ModuleBody	→	Exports Imports AssignmentList
	|	ε
Exports	→	EXPORTS SymbolsExported ";"
	|	ε
Imports	→	IMPORTS SymbolsImported ";"
	|	ε
SymbolsExported	→	SymbolList
	|	ε
SymbolsImported	→	SymbolsFromModuleList
	|	ε
SymbolList	→	Symbol
	|	Symbol "," SymbolList
SymbolsFromModuleList	→	SymbolsFromModule
	|	SymbolsFromModuleList SymbolsFromModule
Symbol	→	`typereference`
	|	`valuereference`
SymbolsFromModule	→	SymbolList FROM ModuleIdentifier

AssignmentList	→	Assignment
	|	AssignmentList Assignment
Assignment	→	TypeAssignment
	|	ValueAssignment
TypeAssignment	→	`typereference` "::=" Type
ValueAssignment	→	`valuereference` Type "::=" Value

DefinedType	→	Externaltypereference
	|	`typereference`
DefinedValue	→	Externalvaluereference
	|	`valuereference`
Externaltypereference	→	`modulereference` "." `typereference`
Externalvaluereference	→	`modulereference` "." `valuereference`

3.1.2 Étape 1

Nous substituons les identificateurs de terminaux (lexicalement) ambigus, en conservant si possible l'information sur leur sémantique originelle (qui apparaît alors en indice). D'autre part, nous appliquons l'option jusqu'à faire disparaître tous les ε, et nous utilisons le lemme d'Arden pour introduire les opérateurs rationnels.

ModuleDefinition → ModuleIdentifier
DEFINITIONS
[TagDefault TAGS]
“::=”
BEGIN
[ModuleBody]
END

Cf. ‘TagDefault’ et ‘ModuleBody’.

ModuleIdentifier → $\texttt{upper}_{mod}$ [“{” ObjIdComponent$^+$ “}”]
ObjIdComponent → $\texttt{lower}_{id}$
| NumberForm
| $\texttt{lower}_{id}$ “(” NumberForm “)”
NumberForm → `number`
| DefinedValue

Option puis expansion globale de ‘AssignedIdentifier’.
Arden de ‘ObjIdComponentList’.
Expansion globale de ‘NameForm’.
Expansion globale de ‘NameAndNumberForm’.
Élimination de la seconde production de ‘ObjectIdentifierValue’ car :
ObjIdComponentList ⇒ ObjIdComponent ObjIdComponentList
⇒ NumberForm ObjIdComponentList ⇒ DefinedValue ObjIdComponentList.
Expansion globale de ‘ObjIdComponentList’.
Expansion globale de ‘*ObjectIdentifierValue*’.
Attention ! Nous conservons ObjectIdentifierValue → “{” ObjIdComponent$^+$ “}”
dans la section (3.3).

TagDefault	→	EXPLICIT
	|	IMPLICIT

Option puis expansion suffixe de 'TagDefault'.

ModuleBody	→	[Exports] [Imports] AssignmentList
Exports	→	EXPORTS [SymbolList] ";"
Imports	→	IMPORTS [SymbolsFromModuleList] ";"
SymbolList	→	{Symbol "," ...}$^+$
SymbolsFromModuleList	→	SymbolsFromModule$^+$
Symbol	→	`typereference`
	|	`valuereference`
SymbolsFromModule	→	SymbolList FROM ModuleIdentifier

Option de 'Exports' et de 'Imports'.
Option puis expansion globale de 'SymbolsExported'.
Option puis expansion globale de 'SymbolsImported'.
Arden de 'SymbolList' et 'SymbolsFromModuleList'.

AssignmentList	→	Assignment$^+$
Assignment	→	`upper`$_{typ}$ "::=" Type
	|	`lower`$_{val}$ Type "::=" Value

Arden de *'AssignmentList'*.
Expansion globale de 'TypeAssignment' et 'ValueAssignment'.

DefinedType	→	`upper`$_{mod}$ "." `upper`$_{typ}$
	|	`upper`$_{typ}$
DefinedValue	→	`upper`$_{mod}$ "." `lower`$_{val}$
	|	`lower`$_{val}$

Expansion globale de 'ExternalTypeReference' et 'ExternalValueReference'.

3.1.3 Étape 2

ModuleDefinition → ModuleIdentifier
DEFINITIONS
[TagDefault TAGS]
" ::= "
BEGIN
[ModuleBody]
END

ModuleIdentifier → `upper`$_{mod}$ [" {" ObjIdComponent$^+$ " }"]
ObjIdComponent → `number`
| `upper`$_{mod}$ "." `lower`$_{val}$
| `lower` [" (" ClassNumber ")"]

> Expansion totale de '*DefinedValue*'.
> Factorisation préfixe (lower).
> Élimination de 'NumberForm' car 'NumberForm' = 'ClassNumber' (Cf. section 3.2).

TagDefault → EXPLICIT
| IMPLICIT

ModuleBody → [Exports] [Imports] Assignment$^+$
Exports → EXPORTS {Symbol "," ...}* " ;"
Imports → IMPORTS SymbolsFromModule* " ;"
SymbolsFromModule → {Symbol "," ...}$^+$ FROM ModuleIdentifier
Symbol → `upper`$_{typ}$
| `lower`$_{val}$

> Expansion globale de '*AssignmentList*'.
> Expansion globale de 'SymbolList'.
> Expansion globale de 'SymbolsFromModuleList'.

Assignment → `upper`$_{typ}$ " ::= " Type
| `lower`$_{val}$ Type " ::= " Value

> 'DefinedType' est déplacée dans la section (3.2).
> '*DefinedValue*' est déplacée dans les sections 3.2 et (3.3).

3.2 Types

3.2.1 Étape 0

Nous donnons d'abord ici la forme normalisée de la section consacrée à la spécification des types ASN.1, après restructuration. Notons que nous récupérons les règles *'DefinedType'* et *'DefinedValue'* en provenance de la section (3.1). Les règles *'Subtype'* et *'ParentType'* sont présentes dans cette section, et non pas dans la section 3.4, pour faciliter la lecture. *'SubtypeSpec'* est présente dans la section (3.4) . Nous noterons aussi que *'ClassNumber'* est une règle d'entrée mixte (Cf. 3.1.3).

Type → DefinedType
| BuiltInType
| Subtype

DefinedType → `upper` ["." $\texttt{upper}_{typ}$]
DefinedValue → [$\texttt{upper}_{mod}$ "."] $\texttt{lower}_{val}$

Factorisation préfixe de *'DefinedType'*.
Factorisation suffixe de *'DefinedValue'*.

BuiltInType → BooleanType
| IntegerType
| BitStringType
| OctetStringType
| NullType
| SequenceType
| SequenceOfType
| SetType
| SetOfType
| ChoiceType
| SelectionType
| TaggedType
| AnyType
| ObjectIdentifierType
| CharacterStringType
| UsefulType
| EnumeratedType
⊏

	⊏	
	|	RealType

NamedType	→	`identifier` Type
	|	Type
	|	SelectionType

Subtype	→	ParentType SubtypeSpec
	|	SET SizeConstraint OF Type
	|	SEQUENCE SizeConstraint OF Type
ParentType	→	Type
SizeConstraint	→	SIZE SubtypeSpec

IntegerType	→	INTEGER
	|	INTEGER "{" NamedNumberList "}"
NamedNumberList	→	NamedNumber
	|	NamedNumberList "," NamedNumber
NamedNumber	→	`identifier` "(" SignedNumber ")"
	|	`identifier` "(" DefinedValue ")"
SignedNumber	→	["-"] `number`

BooleanType	→	BOOLEAN

EnumeratedType	→	ENUMERATED "{" Enumeration "}"
Enumeration	→	NamedNumber
	|	Enumeration "," NamedNumber

RealType	→	REAL

BitStringType	→	BIT STRING
	|	BIT STRING "{" NamedBitList "}"
NamedBitList	→	NamedBit
	|	NamedBitList "," NamedBit
NamedBit	→	`identifier` "(" `number` ")"
	|	`identifier` "(" DefinedValue ")"

OctetStringType	→	OCTET STRING

SequenceOfType	→	SEQUENCE OF Type
	|	SEQUENCE
SetOfType	→	SET OF Type
	|	SET

NullType	→	NULL

SequenceType	→	SEQUENCE "{" ElementTypeList "}"
	|	SEQUENCE "{" "}"
SetType	→	SET "{" ElementTypeList "}"
	|	SET "{" "}"
ElementTypeList	→	ElementType
	|	ElementTypeList "," ElementType
ElementType	→	NamedType
	|	NamedType OPTIONAL
	|	NamedType DEFAULT Value
	|	COMPONENTS OF Type

ChoiceType	→	CHOICE "{" AlternativeTypeList "}"
AlternativeTypeList	→	NamedType
	|	AlternativeTypeList "," NamedType

SelectionType	→	`identifier` "<" Type

TaggedType	→	Tag Type
	|	Tag IMPLICIT Type
	|	Tag EXPLICIT Type
Tag	→	"[" Class ClassNumber "]"
ClassNumber	→	`number`
	|	DefinedValue
Class	→	UNIVERSAL
	|	APPLICATION
	|	PRIVATE
	|	ε

AnyType	→	ANY
	|	ANY DEFINED BY `identifier`

ObjectIdentifierType	→	OBJECT IDENTIFIER

UsefulType	→	EXTERNAL
	|	"UTCTime"
	|	"GeneralizedTime"
	|	"ObjectDescriptor"

CharacterStringType	→	"NumericString"
	|	"PrintableString"
	|	"TeletexString"
	|	"T61String"
	|	"VideotexString"
	|	"VisibleString"
	|	"ISO646String"
	|	"IA5String"
	|	"GraphicString"
	|	"GeneralString"

3.2.2 Étape 1

NamedType → `lower`$_{id}$ Type
| Type

Élimination de la production 'SelectionType' car :
Type ⇒ BuiltInType ⇒ SelectionType

Type → `upper` [“.” `upper`$_{typ}$]
| BuiltInType
| SetType
| SequenceType
| SetOfType
| SequenceOfType
| SelectionType
| TaggedType
| NullType
| Type SubtypeSpec
| SET SIZE SubtypeSpec OF Type
| SEQUENCE SIZE SubtypeSpec OF Type

Expansion totale de *'DefinedType'*.
Expansion partielle des productions *'SetType'*, *'SequenceType'*, *'SetOfType'*, *'SequenceOfType'*, *'NullType'*, *'SelectionType'* et *'TaggedType'* de la règle *'BuiltInType'*.
Expansion globale de '*Subtype*' après l'expansion globale de 'ParentType' :
Subtype ⇒ ParentType SubtypeSpec ⇒ Type SubtypeSpec
Le but est de factoriser à gauche la règle *'Type'* et de faire apparaître la double récurrence, cause d'ambiguïté.
Expansion globale de 'SizeConstraint' qui est déplacée dans la section (3.4).

BuiltInType → BOOLEAN
| INTEGER [“{” {NamedNumber “,” ...}$^+$ “}”]
| BIT STRING [“{” {NamedBit “,” ...}$^+$ “}”]
| OCTET STRING
| CHOICE “{” {NamedType “,” ...}$^+$ “}”
| ANY [DEFINED BY `lower`$_{id}$]
| OBJECT IDENTIFIER
| ENUMERATED “{” {NamedNumber “,” ...}$^+$ “}”
| REAL
⊏

⊏
| "NumericString"
| "PrintableString"
| "TeletexString"
| "T61String"
| "VideotexString"
| "VisibleString"
| "ISO646String"
| "IA5String"
| "GraphicString"
| "GeneralString"
| EXTERNAL
| "UTCTime"
| "GeneralizedTime"
| "ObjectDescriptor"

Expansion globale de *'BooleanType'*, *'IntegerType'*, *'BitStringType'*, *'OctetStringType'*, *'TaggedType'*, *'AnyType'*, *'ObjectIdentifierType'*, *'UsefulType'*, *'CharacterStringType'*, *'RealType'*.
Arden de 'Enumeration' et expansion globale, puis expansion globale de *'EnumeratedType'*.
Arden de 'AlternativeTypeList' et expansion globale, puis expansion globale de *'ChoiceType'*.
Cf. *'Type'*, *'NamedNumber'*, *'NamedBit'*.

NamedNumber → `lower`$_{id}$ "(" AuxNamedNum ")"
AuxNamedNum → SignedNumber
| DefinedValue

Arden de 'NamedNumberList' et expansion globale.
Factorisation préfixe de *'IntegerType'* puis expansion globale.
Factorisation bifixe de 'NamedNumber' ('AuxNamedNum').

NamedBit → `lower`$_{id}$ "(" ClassNumber ")"

Factorisation bifixe : nous reconnaissons la règle 'ClassNumber'.

SequenceOfType → SEQUENCE [OF Type]
SetOfType → SET [OF Type]

Factorisations préfixes.

SequenceType → SEQUENCE “{” {ElementType “,” ...}* “}”
SetType → SET “{” {ElementType “,” ...}* “}”
ElementType → NamedType [ElementTypeSuf]
| COMPONENTS OF Type
ElementTypeSuf → OPTIONAL
| DEFAULT Value

Factorisation bifixe de *'SequenceType'* et *'SetType'*.
Arden de 'ElementTypeList' et expansion globale.
Factorisation préfixe de 'ElementType'.

SelectionType → `lower`$_{id}$ “<” Type

NullType → NULL

TaggedType → “[” [Class] ClassNumber “]” [TagDefault] Type
Class → UNIVERSAL
| APPLICATION
| PRIVATE

Factorisation bifixe de *'TaggedType'* : nous reconnaissons l'option de la règle *'TagDefault'* (Cf. 3.1).
Expansion globale de 'Tag'.
Option de 'Class'.

ClassNumber → `number`
| [`upper`$_{mod}$ “.”] `lower`$_{val}$

Expansion totale de *'DefinedValue'*.

3.2.3 Étape 2

NamedType → `lower`$_{id}$ Type
| Type

Type → `upper` [“.” `upper`$_{typ}$]
| BuiltInType
| SET “{” {ElementType “,” ...}* “}”
| SEQUENCE “{” {ElementType “,” ...}* “}”
| SET [OF Type]
| SEQUENCE [OF Type]
| `lower`$_{id}$ “<” Type
| “[” [Class] ClassNumber “]” [TagDefault] Type
| NULL
| Type SubtypeSpec
| SET SIZE SubtypeSpec OF Type
| SEQUENCE SIZE SubtypeSpec OF Type

Expansion globale de *'SetType'*, *'SequenceType'*, *'SetOfType'*, *'SequenceOfType'*, *'SelectionType'*, *'TaggedType'* et *'NullType'*.

BuiltInType → BOOLEAN
| INTEGER [“{” {NamedNumber “,” ...}+ “}”]
| BIT STRING [“{” {NamedBit “,” ...}+ “}”]
| OCTET STRING
| CHOICE “{” {NamedType “,” ...}+ “}”
| ANY [DEFINED BY `lower`$_{id}$]
| OBJECT IDENTIFIER
| ENUMERATED “{” {NamedNumber “,” ...}+ “}”
| REAL
| “NumericString”
| “PrintableString”
| “TeletexString”
| “T61String”
| “VideotexString”
| “VisibleString”
| “ISO646String”
| “IA5String”
| “GraphicString”
| “GeneralString”
⊏

	⊏	
	|	EXTERNAL
	|	"UTCTime"
	|	"GeneralizedTime"
	|	"ObjectDescriptor"

NamedNumber	→	$\texttt{lower}_{id}$ "(" AuxNamedNum ")"
AuxNamedNum	→	["-"] `number`
	|	[$\texttt{upper}_{mod}$ "."] $\texttt{lower}_{val}$

Expansion totale de 'SignedNumber' et *'DefinedValue'* dans 'AuxNamedNum'.
Déplacement de 'SignedNumber' et *'DefinedValue'* dans la section (3.3).

NamedBit	→	$\texttt{lower}_{id}$ "(" ClassNumber ")"

ElementType	→	NamedType [ElementTypeSuf]
	|	COMPONENTS OF Type
ElementTypeSuf	→	OPTIONAL
	|	DEFAULT Value

Class	→	UNIVERSAL
	|	APPLICATION
	|	PRIVATE
ClassNumber	→	`number`
	|	[$\texttt{upper}_{mod}$ "."] $\texttt{lower}_{val}$

3.2.4 Étape 3

Nous supprimons ici l'ambiguïté de la règle *'Type'*, en appliquant la transformation vue à la section (2.2). Nous rendrons ensuite LL(1) la règle *'NamedType'*.

Type	→	`lower`$_{id}$ "<" Type
	|	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	SetSeq [SIZE SubtypeSpec] OF Type
	|	Type SubtypeSpec
	|	NULL
	|	BuiltInType
	|	`upper` ["." `upper`$_{typ}$]
	|	SetSeq ["{" {ElementType "," ...}* "}"]
SetSeq	→	SET
	|	SEQUENCE

Factorisations suffixes ('SetSeq').

Type	→	`lower`$_{id}$ "<" Type
	|	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	SetSeq [SIZE SubtypeSpec] OF Type
	|	NULL SubtypeSpec*
	|	BuiltInType SubtypeSpec*
	|	`upper` ["." `upper`$_{typ}$] SubtypeSpec*
	|	SetSeq ["{" {ElementType "," ...}* "}"] SubtypeSpec*
SetSeq	→	SET
	|	SEQUENCE

Application de la transformation vue à la section (2.2).

Type	→	`lower`$_{id}$ "<" Type
	|	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	NULL SubtypeSpec*
	|	BuiltInType SubtypeSpec*
	|	`upper` ["." `upper`$_{typ}$] SubtypeSpec*
	|	SetSeq TypeSuf
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	["{" {ElementType "," ...}* "}"] SubtypeSpec*
	|	[SIZE SubtypeSpec] OF Type

Factorisation préfixe ('TypeSuf').

Type	→	`lower`$_{id}$ "<" Type
	|	`upper` ["." `upper`$_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType
AuxType	→	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	BuiltInType SubtypeSpec*
	|	SetSeq [TypeSuf]
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	SubtypeSpec$^+$
	|	"{" {ElementType "," ...}* "}" SubtypeSpec*
	|	[SIZE SubtypeSpec] OF Type

Option de 'TypeSuf'.
Réduction de '*Type*' ('AuxType'). On comprendra pourquoi à l'étape 4 de la section (3.3).

NamedType → `lower`$_{id}$ Type
| Type

NamedType → `lower`$_{id}$ Type
| `lower`$_{id}$ "<" Type
| `upper` ["." `upper`$_{typ}$] SubtypeSpec*
| NULL SubtypeSpec*
| AuxType

Expansion totale de '*Type*'.

NamedType → `lower`$_{id}$ ["<"] Type
| `upper` ["." `upper`$_{typ}$] SubtypeSpec*
| NULL SubtypeSpec*
| AuxType

Factorisation bifixe.

3.2.5 Étape 4

Voici la grammaire finale de la section (3.2).

Type	→	`lower`$_{id}$ "<" Type
	|	`upper` ["." `upper`$_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType
AuxType	→	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	BuiltInType SubtypeSpec*
	|	SetSeq [TypeSuf]
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	SubtypeSpec$^+$
	|	"{" {ElementType "," ...}* "}" SubtypeSpec*
	|	[SIZE SubtypeSpec] OF Type

BuiltInType	→	BOOLEAN
	|	INTEGER ["{" {NamedNumber "," ...}$^+$ "}"]
	|	BIT STRING ["{" {NamedBit "," ...}$^+$ "}"]
	|	OCTET STRING
	|	CHOICE "{" {NamedType "," ...}$^+$ "}"
	|	ANY [DEFINED BY `lower`$_{id}$]
	|	OBJECT IDENTIFIER
	|	ENUMERATED "{" {NamedNumber "," ...}$^+$ "}"
	|	REAL
	|	"NumericString"
	|	"PrintableString"
	|	"TeletexString"
	|	"T61String"
	|	"VideotexString"
	|	"VisibleString"
	|	"ISO646String"
	|	"IA5String"
	|	"GraphicString"
	|	"GeneralString"
	⊏	

	⊏	
	|	EXTERNAL
	|	"UTCTime"
	|	"GeneralizedTime"
	|	"ObjectDescriptor"

NamedType	→	`lower`$_{id}$ ["<"] Type
	|	`upper` ["." `upper`$_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType

NamedNumber	→	`lower`$_{id}$ "(" AuxNamedNum ")"
AuxNamedNum	→	["-"] `number`
	|	[`upper`$_{mod}$ "."] `lower`$_{val}$

NamedBit	→	`lower`$_{id}$ "(" ClassNumber ")"

ElementType	→	NamedType [ElementTypeSuf]
	|	COMPONENTS OF Type
ElementTypeSuf	→	OPTIONAL
	|	DEFAULT Value

Class	→	UNIVERSAL
	|	APPLICATION
	|	PRIVATE
ClassNumber	→	`number`
	|	[`upper`$_{mod}$ "."] `lower`$_{val}$

3.3 Valeurs

3.3.1 Étape 0

Voici la forme normalisée de la grammaire des valeurs ASN.1. Notons que nous récupérons ici les règles *'ObjectIdentifierValue'*, en provenance de la section 3.1, *'DefinedValue'* et *'SignedNumber'*, en provenance de la section (3.2).

Value	→	BuiltInValue
	|	DefinedValue
DefinedValue	→	[`upper`$_{mod}$ "."] `lower`$_{val}$
SignedNumber	→	["-"] `number`
BuiltInValue	→	BooleanValue
	|	IntegerValue
	|	BitStringValue
	|	OctetStringValue
	|	NullValue
	|	SequenceValue
	|	SequenceOfValue
	|	SetValue
	|	SetOfValue
	|	ChoiceValue
	|	SelectionValue
	|	TaggedValue
	|	AnyValue
	|	ObjectIdentifierValue
	|	CharacterStringValue
	|	EnumeratedValue
	|	RealValue
NamedValue	→	`identifier` Value
	|	Value

BooleanValue	→	TRUE
	|	FALSE
IntegerValue	→	SignedNumber
	|	`identifier`
EnumeratedValue	→	`identifier`
RealValue	→	NumericRealValue
	|	SpecialRealValue
NumericRealValue	→	"{" Mantissa "," Base "," Exponent "}"
	|	"0"
Mantissa	→	SignedNumber
Base	→	"2"
	|	"10"
Exponent	→	SignedNumber
SpecialRealValue	→	PLUS-INFINITY
	|	MINUS-INFINITY
OctetStringValue	→	`bstring`
	|	`hstring`
BitStringValue	→	`bstring`
	|	`hstring`
	|	"{" IdentifierList "}"
	|	"{" "}"
IdentifierList	→	`identifier`
	|	IdentifierList "," `identifier`
NullValue	→	NULL

SequenceValue → "{" ElementValueList "}"
| "{" "}"
SetValue → "{" ElementValueList "}"
| "{" "}"
ElementValueList → NamedValue
| ElementValueList "," NamedValue

ChoiceValue → [`identifier` ":"] Value

Et non pas : *ChoiceValue* → NamedValue (**Erratum**).

SelectionValue → Value

Et non pas : *SelectionValue* → NamedValue (**Erratum**).

SequenceOfValue → "{" ValueList "}"
| "{" "}"
SetOfValue → "{" ValueList "}"
| "{" "}"
ValueList → Value
| ValueList "," Value

TaggedValue → Value

AnyValue → Type ":" Value

Et non pas : *AnyValue* → Type Value (**Erratum**).

ObjectIdentifierValue → "{" ObjIdComponent$^+$ "}"

CharacterStringValue → `cstring`

3.3.2 Étape 1

On prendra garde à ce que les terminaux `bstring` et `hstring` sont fusionnés en `basednumber`. Cf. (6.2). D'autre part, les productions en doublon ne sont pas immédiatement fusionnées pour des raisons de clarté. On notera aussi que les terminaux "0", "2" et "10" deviennent le terminal `number`. (C'est ce que fait normalement un analyseur lexical.)

Value → BuiltInValue
| IntegerValue
| NullValue
| ChoiceValue
| SelectionValue
| TaggedValue
| AnyValue
| EnumeratedValue
| [`upper`$_{mod}$ "."] `lower`$_{val}$

Expansion partielle de *'IntegerValue'*, *'NullValue'*, *'ChoiceValue'*, *'SelectionValue'*, *'TaggedValue'*, *'AnyValue'* et *'EnumeratedValue'* de la règle *'BuiltInValue'*.
Expansion globale de *'DefinedValue'*.

BuiltInValue → TRUE
| FALSE
| `basednum`
| "{" {`lower`$_{id}$ "," ...}* "}"
| `basednum`
| "{" {NamedValue "," ...}* "}"
| "{" {Value "," ...}* "}"
| "{" {NamedValue "," ...}* "}"
| "{" {Value "," ...}* "}"
| "{" ObjIdComponent^{+} "}"
| `string`
| "{" ["-"] `number` "," `number` "," ["-"] `number` "}"
| `number`
| PLUS-INFINITY
| MINUS-INFINITY

Expansion globale de *'BooleanValue'*, *'ObjectIdentifierValue'*, *'CharacterStringValue'* et *'OctetStringValue'*.
Arden de 'IdentifierList' et expansion globale, puis factorisation bifixe dans *'BitStringValue'*.
Expansion globale de *'BitStringValue'*.
Arden de 'ElementValueList' et expansion globale, puis factorisation bifixe dans *'SequenceValue'* et *'SetValue'*.
Expansion globale de *'SequenceValue'* et *'SetValue'*.
Arden de 'ValueList' et expansion globale, puis factorisation bifixe dans *'SequenceOfValue'* et *'SetOfValue'*.
Expansion globale de *'SequenceOfValue'* et *'SetOfValue'*.
Sous-section *'RealValue'*: expansion totale de *'SignedNumber'* dans 'Mantissa' et 'Exponent'.
Expansion globale de 'Mantissa', 'Base' (Base → number) et 'Exponent'.
Expansion globale de 'NumericRealValue', 'SpecialRealValue' et *'RealValue'*.

NamedValue → `lower`$_{id}$ Value
| Value

IntegerValue → ["-"] `number`
| `lower`$_{id}$

Expansion globale de *'SignedNumber'*.

EnumeratedValue → `lower`$_{id}$

NullValue → NULL

ChoiceValue → [`lower`$_{id}$ ":"] Value

SelectionValue → Value

TaggedValue → Value

AnyValue → Type ":" Value

3.3.3 Étape 2

Value → BuiltInValue
| ["-"] `number`
| `lower`$_{id}$
| NULL
| [`lower`$_{id}$ ":"] Value
| Value
| Value
| Type ":" Value
| `lower`$_{id}$
| [`upper`$_{mod}$ "."] `lower`$_{val}$
| `number`

Expansion globale de *'IntegerValue'*, *'NullValue'*, *'ChoiceValue'*, *'SelectionValue'*, *'TaggedValue'*, *'AnyValue'*, et *'EnumeratedValue'*.
Expansion partielle de *BuiltInValue* → number

BuiltInValue → TRUE
| FALSE
| PLUS-INFINITY
| MINUS-INFINITY
| `basednum`
| `string`
| "{" BetBraces "}"

BetBraces → {NamedValue "," ...}*
| ObjIdComponent^{+}
| ["-"] `number` "," `number` "," ["-"] `number`

Lemme d'Arden (Élimination des redondances Value → Value | ...).
Élimination de la production "{" {lower$_{id}$ "," ...}* "}"
car "{" {Value "," ...}* "}" ⇒ "{" {lower$_{id}$ "," ...}* "}"
Élimination de la production "{"{Value "," ...}* "}"
car "{" {NamedValue "," ...}* "}" ⇒ "{"{Value "," ...}* "}"
Factorisation bifixe ('BetBraces').

NamedValue → `lower`$_{id}$ Value
| Value

3.3.4 Étape 3

Value	→	BuiltInValue
	|	["-"] `number`
	|	NULL
	|	`lower` [":" Value]
	|	Type ":" Value
	|	$\texttt{upper}_{mod}$ "." $\texttt{lower}_{val}$

Élimination des redondances.
Arden de *'Value'*.
Factorisation préfixe de lower.

BuiltInValue	→	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" [BetBraces] "}"
BetBraces	→	{NamedValue "," ...}$^+$
	|	ObjIdComponent$^+$

Option de 'BetBraces'.
Élimination de la troisième production de 'BetBraces' car :
{NamedValue "," ...}$^+ \stackrel{+}{\Longrightarrow}$ ["-"] number "," number "," ["-"] number

NamedValue	→	$\texttt{lower}_{id}$ Value
	|	Value

3.3.5 Étape 4

Nous rendons LL(1) la règle *'Value'*, puis *'NamedValue'*.

Value	→	BuiltInValue
	|	["-"] `number`
	|	NULL
	|	`lower` [":" Value]
	|	$\texttt{upper}_{mod}$ "." $\texttt{lower}_{val}$
	|	$\texttt{lower}_{id}$ "<" Type ":" Value
	|	`upper` ["." $\texttt{upper}_{typ}$] SubtypeSpec* ":" Value
	|	NULL SubtypeSpec* ":" Value
	|	AuxType ":" Value

Expansion totale de *'Type'*.

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	["-"] `number`
AuxVal0	→	BuiltInValue
	|	AuxType ":" Value
	|	NULL [SubtypeSpec* ":" Value]
AuxVal1	→	["." $\texttt{upper}_{typ}$] SubtypeSpec* ":" Value
	|	"." $\texttt{lower}_{val}$
AuxVal2	→	["<" Type] ":" Value

Factorisations préfixes.
Réduction ('AuxVal0').
La raison est donnée en (3.3.6).

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	[“-”] `number`
AuxVal0	→	BuiltInValue
	|	AuxType “:” Value
	|	NULL [SpecVal]
AuxVal1	→	SpecVal
	|	“.” AuxVal11
AuxVal2	→	[“<” Type] “:” Value
AuxVal11	→	$\texttt{upper}_{typ}$ SpecVal
	|	$\texttt{lower}_{val}$
SpecVal	→	SubtypeSpec* “:” Value

Expansion inverse (‘SpecVal’).
Factorisation préfixe de ‘AuxVal1’ (‘AuxVal11’).

NamedValue	→	$\texttt{lower}_{id}$ Value
	|	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	[“-”] `number`

Expansion *‘Value’*.

NamedValue	→	`lower` [NamedValSuf]
	|	`upper` AuxVal1
	|	[“-”] `number`
	|	AuxVal0
NamedValSuf	→	Value
	|	AuxVal2

Factorisation préfixe.

3.3.6 Étape 5

Nous rendons maintenant LL(1) la règle 'BetBraces'.

BetBraces	→	NamedValue [“,” {NamedValue “,” ...}$^+$]
	|	ObjIdComponent ObjIdComponent*

Opérations rationnelles.

BetBraces	→	`lower` [NamedValSuf] [AuxNamed]
	|	`upper` AuxVal1 [AuxNamed]
	|	[“-”] `number` [AuxNamed]
	|	AuxVal0 [AuxNamed]
	|	`number` ObjIdComponent*
	|	`upper`$_{mod}$ “.” `lower`$_{val}$ ObjIdComponent*
	|	`lower` [“(” ClassNumber “)”] ObjIdComponent*
AuxNamed	→	“,” {NamedValue “,” ...}$^+$

Expansion inverse ('AuxNamed').
Expansion totale de *'NamedValue'*.
Expansion totale de *'ObjIdComponent'*.
Cf. (3.1.3).

BetBraces	→	AuxVal0 [AuxNamed]
	|	“-” `number` [AuxNamed]
	|	`lower` [AuxBet1]
	|	`upper` AuxBet2
	|	`number` [AuxBet3]
AuxBet1	→	“(” ClassNumber “)” ObjIdComponent*
	|	ObjIdComponent$^+$
	|	NamedValSuf [AuxNamed]
	|	AuxNamed
AuxBet2	→	AuxVal1 [AuxNamed]
	|	“.” `lower`$_{val}$ ObjIdComponent*
AuxBet3	→	ObjIdComponent$^+$
	|	AuxNamed

Factorisations préfixes et développement de 'AuxBet1'.

AuxBet1	→	"(" ClassNumber ")" ObjIdComponent*
	|	AuxNamed
	|	ObjIdComponent ObjIdComponent*
	|	ObjIdComponent ObjIdComponent*
	|	ObjIdComponent ObjIdComponent*
	|	Value [AuxNamed]
	|	AuxVal2 [AuxNamed]
AuxBet2	→	SpecVal [AuxNamed]
	|	"." $\texttt{upper}_{typ}$ SpecVal [AuxNamed]
	|	"." $\texttt{lower}_{val}$ [AuxNamed]
	|	"." $\texttt{lower}_{val}$ ObjIdComponent*

Opération rationnelle dans 'AuxBet1'.
Expansion totale de 'NamedValSuf'.
Expansion totale de 'AuxVal1' dans 'AuxBet2', puis de 'AuxVal11'.

AuxBet1	→	"(" ClassNumber ")" ObjIdComponent*
	|	AuxNamed
	|	AuxVal2 [AuxNamed]
	|	`number` ObjIdComponent*
	|	$\texttt{upper}_{mod}$ "." $\texttt{lower}_{val}$ ObjIdComponent*
	|	`lower` ["(" ClassNumber ")"] ObjIdComponent*
	|	AuxVal0 [AuxNamed]
	|	`upper` AuxVal1 [AuxNamed]
	|	`lower` [AuxVal2] [AuxNamed]
	|	["-"] `number` [AuxNamed]
AuxBet2	→	SpecVal [AuxNamed]
	|	"." AuxBet21
AuxBet21	→	$\texttt{upper}_{typ}$ SpecVal [AuxNamed]
	|	$\texttt{lower}_{val}$ [AuxBet3]

Expansion totale de 'ObjectIdComponent'.
Expansion totale de '*Value*'.
Factorisation préfixe de 'AuxBet2'.
  Nous reconnaissons 'AuxBet3'.

AuxBet1 → "(" ClassNumber ")" ObjIdComponent*
| AuxNamed
| AuxVal2 [AuxNamed]
| "-" `number` [AuxNamed]
| AuxVal0 [AuxNamed]

| `lower` [AuxBet11]
| `number` [AuxBet3]
| `upper` AuxBet2

AuxBet11 → "(" ClassNumber ")" ObjIdComponent*
| ObjIdComponent^{+}
| AuxVal2 [AuxNamed]
| AuxNamed

Factorisations préfixes dans 'AuxBet1' ('AuxBet11') où nous reconnaissons 'AuxBet3' et 'AuxBet2'.
Nous développons 'AuxBet11'.
NOTA : Nous n'avons pas fait figurer ici les règles 'AuxBet2' et 'AuxBet21'.

3.3.7 Étape 6

Nous rappelons le bilan des transformations.

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	["-"] `number`
AuxVal0	→	BuiltInValue
	|	AuxType ":" Value
	|	NULL [SpecVal]
AuxVal1	→	SpecVal
	|	"." AuxVal11
AuxVal2	→	["<" Type] ":" Value
AuxVal11	→	$\texttt{upper}_{typ}$ SpecVal
	|	$\texttt{lower}_{val}$
SpecVal	→	SubtypeSpec* ":" Value

BuiltInValue	→	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" [BetBraces] "}"

BetBraces	→	AuxVal0 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	`lower` [AuxBet1]
	|	`upper` AuxBet2
	|	`number` [AuxBet3]
AuxBet1	→	"(" ClassNumber ")" ObjIdComponent*
	|	AuxNamed
	|	AuxVal2 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	AuxVal0 [AuxNamed]
	|	`lower` [AuxBet11]
	|	`number` [AuxBet3]
	|	`upper` AuxBet2
AuxBet2	→	SpecVal [AuxNamed]
	|	"." AuxBet21
AuxBet3	→	ObjIdComponent+
	|	AuxNamed
AuxBet11	→	"(" ClassNumber ")" ObjIdComponent*
	|	ObjIdComponent+
	|	AuxVal2 [AuxNamed]
	|	AuxNamed
AuxBet21	→	$\texttt{upper}_{typ}$ SpecVal [AuxNamed]
	|	$\texttt{lower}_{val}$ [AuxBet3]
AuxNamed	→	"," {NamedValue "," ...}+
NamedValue	→	`lower` [NamedValSuf]
	|	`upper` AuxVal1
	|	["-"] `number`
	|	AuxVal0
NamedValSuf	→	Value
	|	AuxVal2

3.4 Sous-types

3.4.1 Étape 0

Nous présentons ici la forme normalisée de la grammaire des sous-types ASN.1, avec un découpage en sous-sections. Notons que nous récupérons ici la règle 'SizeConstraint' en provenance de la section (3.2).

SubtypeSpec	→	"(" SubtypeValueSet SubtypeValueSetList ")"
SubtypeValueSetList	→	"|" SubtypeValueSet SubtypeValueSetList
	|	ε
SubtypeValueSet	→	SingleValue
	|	ContainedSubtype
	|	ValueRange
	|	PermittedAlphabet
	|	SizeConstraint
	|	InnerTypeConstraints

SingleValue	→	Value

ContainedSubtype	→	INCLUDES Type

ValueRange	→	LowerEndPoint ".." UpperEndPoint
LowerEndPoint	→	LowerEndValue
	|	LowerEndValue "<"
UpperEndPoint	→	UpperEndValue
	|	"<" UpperEndValue
LowerEndValue	→	Value
	|	MIN
UpperEndValue	→	Value
	|	MAX

SizeConstraint	→	SIZE SubtypeSpec

PermittedAlphabet	→	FROM SubtypeSpec

InnerTypeConstraints	→	WITH COMPONENT SingleTypeConstraint
	|	WITH COMPONENTS MultipleTypeConstraints
SingleTypeConstraint	→	SubtypeSpec
MultipleTypeConstraints	→	FullSpecification
	|	PartialSpecification
FullSpecification	→	"{" TypeConstraints "}"
PartialSpecification	→	"{" "..." "," TypeConstraints "}"
TypeConstraints	→	NamedConstraint
	|	NamedConstraint "," TypeConstraints
NamedConstraint	→	`identifier` Constraint
	|	Constraint
Constraint	→	ValueConstraint PresenceConstraint
ValueConstraint	→	SubtypeSpec
	|	ε
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL
	|	ε

3.4.2 Étape 1

SubtypeSpec → "(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet → SingleValue
| ContainedSubtype
| ValueRange
| PermittedAlphabet
| SizeConstraint
| InnerTypeConstraints

Arden de 'SubtypeValueSetList', puis expansion globale.

SingleValue → Value

ContainedSubtype → INCLUDES Type

ValueRange → LowerEndValue ["<"] ".." ["<"] UpperEndValue
LowerEndValue → Value
| MIN
UpperEndValue → Value
| MAX

Factorisation préfixe de 'LowerEndPoint'.
Factorisation suffixe de 'UpperEndPoint'.
Expansion globale de 'LowerEndPoint' et 'UpperEndPoint'.

SizeConstraint → SIZE SubtypeSpec

PermittedAlphabet → FROM SubtypeSpec

InnerTypeConstraints	→	WITH InnerTypeSuf
InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	"{" ["..." ","] TypeConstraints "}"
TypeConstraints	→	{NamedConstraint "," ...}$^+$
NamedConstraint	→	[$\texttt{lower}_{id}$] Constraint
Constraint	→	[SubtypeSpec] [PresenceConstraint]
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL

Expansion globale de 'SingleTypeConstraint'.
Factorisation préfixe de *'InnerTypeConstraints'* ('InnerTypeSuf').
Expansion globale de 'FullSpecification' et 'PartialSpecification'.
Factorisation bifixe de 'MultipleTypeConstraints'.
Arden de 'TypeConstraints'.
Factorisation suffixe de 'NamedConstraint'.
Option de 'ValueConstraint', puis expansion globale.
Option de 'PresenceConstraint'.

3.4.3 Étape 2

SubtypeSpec	→	"(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet	→	Value
	|	INCLUDES Type
	|	LowerEndValue ["<"] ".." ["<"] UpperEndValue
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
LowerEndValue	→	Value
	|	MIN
UpperEndValue	→	Value
	|	MAX
InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	"{" ["..." ","] {[NamedConstraint] "," ...} "}"
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	|	SubtypeSpec [PresenceConstraint]
	|	PresenceConstraint
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL

Expansion globale de 'SingleValue'.
Expansion globale de 'ContainedSubtype'.
Expansion globale de 'ValueRange'.
Expansion globale de 'PermittedAlphabet'.
Expansion globale de 'SizeConstraint'.
Expansion globale de 'InnerTypeConstraints'.
Expansion globale de 'TypeConstraints'.
Expansion globale de 'Constraint' puis option de 'NamedConstraint'.

3.4.4 Étape 3

SubtypeSpec	→	"(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet	→	Value [SubValSetSuf]
	|	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue
UpperEndValue	→	Value
	|	MAX
InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	"{" ["..." ","] {[NamedConstraint] "," ...} "}"
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	|	SubtypeSpec [PresenceConstraint]
	|	PresenceConstraint
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL

Expansion globale de 'LowerEndValue'.
Factorisation préfixe de 'SubtypeValueSet' ('SubValSetSuf').

3.4.5 Étape 4

Nous voyons apparaître un problème plus subtil qui empêche la grammaire d'être LL(1). En effet, nous avions obtenu précédemment (Cf. 3.3) :

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	["-"] `number`
AuxVal2	→	["<" Type] " :" Value

La troisième production de *'Value'* implique que le terminal "<" ne doit pas être un symbole possible après une occurrence de *'Value'*, sinon nous ne vérifions pas la troisième condition définissant une grammaire LL(1) — cf. (4). Or nous venons de former :

SubtypeValueSet	→	Value [SubValSetSuf]
	|	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue

Ce qui implique que "<" *est* un symbole possible après *'Value'*...
Nous allons montrer que nous pouvons éliminer cet obstacle. L'idée consiste à faire apparaître le sous-mot 'Value [SubValSetSuf]' partout où c'est possible à l'aide d'expansions, et à en former une règle. Nous tenterons ensuite de transformer cette règle pour en obtenir une définition ne produisant jamais un mot contenant 'Value [SubValSetSuf]' : à chaque occurrence (dans les règles dépendantes ou dans la règle elle-même) nous placerons un appel à cette règle. Il faut noter qu'il n'était pas certain *a priori* qu'une telle solution existait.

SubtypeValueSet	→	SVSAux
	|	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue
SVSAux	→	Value [SubValSetSuf]

Factorisation inverse ('SVSAux').

```
SVSAux    →  AuxVal0 [SubValSetSuf]
          |  upper AuxVal1 [SubValSetSuf]
          |  lower [AuxVal2] [SubValSetSuf]
          |  ["-"] number [SubValSetSuf]
```

Expansion totale de '*Value*'.

```
SVSAux    →  BuiltInValue [SubValSetSuf]
          |  AuxType ":" Value [SubValSetSuf]
          |  NULL [SpecVal] [SubValSetSuf]
          |  upper SVSAux1
          |  lower [SVSAux2]
          |  ["-"] number [SubValSetSuf]

SVSAux1   →  AuxVal1 [SubValSetSuf]

SVSAux2   →  AuxVal2 [SubValSetSuf]
          |  SubValSetSuf
```

Expansion totale de 'AuxVal0'.
Expansions inverses ('SVSAux1' et 'SVSAux2').

```
SVSAux    →  BuiltInValue [SubValSetSuf]
          |  AuxType ":" SVSAux
          |  NULL [SVSAux3]
          |  upper SVSAux1
          |  lower [SVSAux2]
          |  ["-"] number [SubValSetSuf]
SVSAux1   →  SpecVal [SubValSetSuf]
          |  "." AuxVal11 [SubValSetSuf]
SVSAux2   →  ["<" Type] ":" Value [SubValSetSuf]
          |  ["<"] ".." ["<"] UpperEndValue
SVSAux3   →  SpecVal [SubValSetSuf]
          |  SubValSetSuf
```

Expansion inverse ('SVSAux3').
Expansion totale de 'AuxVal1'.
Expansion totale de 'AuxVal2' et 'SubValSetSuf'.
Expansions inverses ('SVSAux1' et 'SVSAux2').

SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType ":" SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	["-"] `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec* ":" Value [SubValSetSuf]
	|	"." SVSAux11
SVSAux2	→	":" SVSAux
	|	".." ["<"] UpperEndValue
	|	"<" SVSAux21
SVSAux3	→	SubtypeSpec* ":" Value [SubValSetSuf]
	|	SubValSetSuf
SVSAux11	→	$\texttt{upper}_{typ}$ SpecVal [SubValSetSuf]
	|	$\texttt{lower}_{val}$ [SubValSetSuf]
SVSAux21	→	Type ":" SVSAux
	|	".." ["<"] UpperEndValue

Nous reconnaissons 'SVSAux' dans 'SVSAux2', puis factorisation préfixe ('SVSAux21').
Expansion totale de 'SpecVal' dans 'SVSAux1' et 'SVSAux3'.
Expansion totale inverse de 'SVSAux11' ('SVSAux11'):
SVSAux11 → AuxVal11 [SubValSetSuf]
puis expansion totale de 'AuxVal11' dans 'SVSAux11'.

SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType ":" SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	["-"] `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec* ":" SVSAux
	|	"." SVSAux11
SVSAux2	→	":" SVSAux
	|	".." ["<"] UpperEndValue
	|	"<" SVSAux21
SVSAux3	→	SubtypeSpec* ":" SVSAux
	|	SubValSetSuf
SVSAux11	→	$\texttt{upper}_{typ}$ SubtypeSpec* ":" SVSAux
	|	$\texttt{lower}_{val}$ [SubValSetSuf]
SVSAux21	→	Type ":" SVSAux
	|	".." ["<"] UpperEndValue

Expansion totale de 'SpecVal' dans 'SVSAux11'.
Nous reconnaissons 'SVSAux' dans 'SVSAux1', 'SVSAux3' et 'SVSAux11'.

3.4.6 Étape 5

Nous présentons ici le bilan des transformations.

SubtypeSpec	→	"(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet	→	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
	|	SVSAux

InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	"{" ["..." ","] {[NamedConstraint] "," ...} "}"
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	|	SubtypeSpec [PresenceConstraint]
	|	PresenceConstraint
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL

SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue
UpperEndValue	→	Value
	|	MAX

SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType ":" SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	["-"] `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec* ":" SVSAux
	|	"." SVSAux11
SVSAux2	→	":" SVSAux
	|	".." ["<"] UpperEndValue
	|	"<" SVSAux21
SVSAux3	→	SubtypeSpec* ":" SVSAux
	|	SubValSetSuf
SVSAux11	→	`upper`$_{typ}$ SubtypeSpec* ":" SVSAux
	|	`lower`$_{val}$ [SubValSetSuf]
SVSAux21	→	Type ":" SVSAux
	|	".." ["<"] UpperEndValue

3.5 Nouvelle grammaire ASN.1 :1990

Nous donnons ici la nouvelle grammaire ASN.1 obtenue par les transformations précédentes. *Elle décrit exactement le même langage que la grammaire normalisée par l'ISO.*

MODULES

ModuleDefinition	→	ModuleIdentifier DEFINITIONS [TagDefault TAGS] “::=” BEGIN [ModuleBody] END
ModuleIdentifier	→	$\texttt{upper}_{mod}$ [“{” ObjIdComponent$^+$ “}”]
ObjIdComponent	→	`number` | $\texttt{upper}_{mod}$ “.” $\texttt{lower}_{val}$ | `lower` [“(” ClassNumber “)”]
TagDefault	→	EXPLICIT | IMPLICIT
ModuleBody	→	[Exports] [Imports] Assignment$^+$
Exports	→	EXPORTS {Symbol “,” ...}* “;”
Imports	→	IMPORTS SymbolsFromModule* “;”
SymbolsFromModule	→	{Symbol “,” ...}$^+$ FROM ModuleIdentifier
Symbol	→	$\texttt{upper}_{typ}$ | $\texttt{lower}_{val}$
Assignment	→	$\texttt{upper}_{typ}$ “::=” Type | $\texttt{lower}_{val}$ Type “::=” Value

TYPES

Type	→	`lower`$_{id}$ "<" Type
	|	`upper` ["." `upper`$_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType
AuxType	→	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	BuiltInType SubtypeSpec*
	|	SetSeq [TypeSuf]
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	SubtypeSpec$^+$
	|	"{" {ElementType "," ...}* "}" SubtypeSpec*
	|	[SIZE SubtypeSpec] OF Type

BuiltInType	→	BOOLEAN
	|	INTEGER ["{" {NamedNumber "," ...}$^+$ "}"]
	|	BIT STRING ["{" {NamedBit "," ...}$^+$ "}"]
	|	OCTET STRING
	|	CHOICE "{" {NamedType "," ...}$^+$ "}"
	|	ANY [DEFINED BY `lower`$_{id}$]
	|	OBJECT IDENTIFIER
	|	ENUMERATED "{" {NamedNumber "," ...}$^+$ "}"
	|	REAL
	|	"NumericString"
	|	"PrintableString"
	|	"TeletexString"
	|	"T61String"
	|	"VideotexString"
	|	"VisibleString"
	|	"ISO646String"
	|	"IA5String"
	|	"GraphicString"
	|	"GeneralString"
	|	EXTERNAL
	|	"UTCTime"
	|	"GeneralizedTime"
	|	"ObjectDescriptor"

NamedType	→	`lower`$_{id}$ [“<”] Type
	|	`upper` [“.” `upper`$_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType
NamedNumber	→	`lower`$_{id}$ “(” AuxNamedNum “)”
AuxNamedNum	→	[“-”] `number`
	|	[`upper`$_{mod}$ “.”] `lower`$_{val}$
NamedBit	→	`lower`$_{id}$ “(” ClassNumber “)”
ElementType	→	NamedType [ElementTypeSuf]
	|	COMPONENTS OF Type
ElementTypeSuf	→	OPTIONAL
	|	DEFAULT Value
Class	→	UNIVERSAL
	|	APPLICATION
	|	PRIVATE
ClassNumber	→	`number`
	|	[`upper`$_{mod}$ “.”] `lower`$_{val}$

VALEURS

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	[“-”] `number`
AuxVal0	→	BuiltInValue
	|	AuxType “:” Value
	|	NULL [SpecVal]
AuxVal1	→	SpecVal
	|	“.” AuxVal11
AuxVal2	→	[“<” Type] “:” Value
AuxVal11	→	`upper`$_{typ}$ SpecVal
	|	`lower`$_{val}$
SpecVal	→	SubtypeSpec* “:” Value

BuiltInValue	→	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" [BetBraces] "}"

BetBraces	→	AuxVal0 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	`lower` [AuxBet1]
	|	`upper` AuxBet2
	|	`number` [AuxBet3]
AuxBet1	→	"(" ClassNumber ")" ObjIdComponent*
	|	AuxNamed
	|	AuxVal2 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	AuxVal0 [AuxNamed]
	|	`lower` [AuxBet11]
	|	`number` [AuxBet3]
	|	`upper` AuxBet2
AuxBet2	→	SpecVal [AuxNamed]
	|	"." AuxBet21
AuxBet3	→	ObjIdComponent+
	|	AuxNamed
AuxBet11	→	"(" ClassNumber ")" ObjIdComponent*
	|	ObjIdComponent+
	|	AuxVal2 [AuxNamed]
	|	AuxNamed
AuxBet21	→	$\texttt{upper}_{typ}$ SpecVal [AuxNamed]
	|	$\texttt{lower}_{val}$ [AuxBet3]
AuxNamed	→	"," {NamedValue "," ...}+
NamedValue	→	`lower` [NamedValSuf]
	|	`upper` AuxVal1
	|	["-"] `number`
	|	AuxVal0
NamedValSuf	→	Value
	|	AuxVal2

SOUS-TYPES

SubtypeSpec	→	"(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet	→	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
	|	SVSAux
SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue
UpperEndValue	→	Value
	|	MAX
InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	"{" ["..." ","] {[NamedConstraint] "," ...} "}"
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	|	SubtypeSpec [PresenceConstraint]
	|	PresenceConstraint
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL
SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType ":" SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	["-"] `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec* ":" SVSAux
	|	"." SVSAux11
SVSAux2	→	":" SVSAux
	|	".." ["<"] UpperEndValue
	|	"<" SVSAux21
SVSAux3	→	SubtypeSpec* ":" SVSAux
	|	SubValSetSuf
SVSAux11	→	`upper`$_{typ}$ SubtypeSpec* ":" SVSAux
	|	`lower`$_{val}$ [SubValSetSuf]
SVSAux21	→	Type ":" SVSAux
	|	".." ["<"] UpperEndValue

4 Preuve LL(1)

Nous apportons ici la preuve que la grammaire précédente obtenue après moult transformations est LL(1). Nous rappelons préalablement la définition exacte de la propriété LL(1).

4.1 Définition de la propriété LL(1)

Les grammaires LL(1) sont les grammaires analysables de façon descendante avec un lexème de prévision, sans reprise arrière (*Left to right scanning of the input, building a Leftmost derivation with one token of look-ahead*). Pour les définir formellement, il est nécessaire d'introduire au préalable deux fonctions. Nous noterons N l'ensemble des non-terminaux et Σ l'ensemble des terminaux.

4.1.1 La fonction *Premiers*

La fonction $\mathcal{P}$ [7] est la suivante :

$$\forall \mathrm{A} \in N,\quad \mathcal{P}(\mathrm{A}) = \{x \in \Sigma \cup \{\varepsilon\} \mid \mathrm{A} \stackrel{*}{\Longrightarrow} x\alpha \quad et \quad x \neq \varepsilon \quad ou \quad \mathrm{A} \stackrel{*}{\Longrightarrow} x\}$$

4.1.2 La fonction *Suivants*

La fonction $\mathcal{S}$ [8] est définie par :

$$\forall \mathrm{A} \in N,\quad \mathcal{S}(\mathrm{A}) = \{x \in \Sigma \mid \exists \mathrm{B} \in N, \mathrm{B} \stackrel{*}{\Longrightarrow} \alpha \mathrm{A} x \beta \quad ou \quad \mathrm{B} \stackrel{*}{\Longrightarrow} \alpha \mathrm{A} x\}$$

4.1.3 Définition LL(1)

Nous notons ici l'implication par $\models$, pour éviter toute confusion avec la relation $\Longrightarrow$. Une grammaire est LL(1) si et seulement si elle vérifie les propriétés suivantes :

$$\forall \mathrm{A} \in N,\quad \neg(\mathrm{A} \stackrel{*}{\Longrightarrow} \mathrm{A}\alpha) \tag{P1}$$

$$\mathrm{A} \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad \models \quad \bigcap_{i=1}^{n} \mathcal{P}(\alpha_i) = \emptyset \tag{P2}$$

$$\mathrm{A} \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad et \quad \alpha_1 \stackrel{*}{\Longrightarrow} \varepsilon \quad \models \quad \forall i \in [1, n],\ \mathcal{P}(\alpha_i) \cap \mathcal{S}(\mathrm{A}) = \emptyset \tag{P3}$$

[7] Prononcez « premiers »
[8] Prononcez « suivants »

4.1.4 Extension aux opérateurs rationnels

Nous vîmes au chapitre 1.3 la définition des opérateurs rationnels utilisés dans cette grammaire ASN.1. Nous indiquions que nous pouvions considérer les expressions rationnelles comme étant produites par une règle spécifique. Nous allons étendre ici les fonctions $\mathcal{P}$ et $\mathcal{S}$ à ces expressions et en donner une définition récursive et algorithmique.

Nouvelle définition de $\mathcal{P}$:

$$\begin{cases} \mathcal{P}(\varepsilon) = \{\varepsilon\} \\ \mathcal{P}(x\gamma) = \{x\} \\ \mathcal{P}(\mathrm{B}\gamma) = \mathcal{P}(\mathrm{B}) \\ \mathcal{P}([\beta]\gamma) = \mathcal{P}(\beta) \cup \mathcal{P}(\gamma) \\ \mathcal{P}(\{\mathrm{B\ b}\ldots\}^*\gamma) = \mathcal{P}(\mathrm{B}) \cup \mathcal{P}(\gamma) \\ \mathcal{P}(\{\mathrm{B\ b}\ldots\}^+\gamma) = \mathcal{P}(\mathrm{B}) \\ \mathcal{P}(\{[\mathrm{B}]\ \mathrm{b}\ldots\}\gamma) = \mathcal{P}(\mathrm{B}) \cup \{\mathrm{b}\} \cup \mathcal{P}(\gamma) \\ \mathcal{P}(\beta^*\gamma) = \mathcal{P}(\beta) \cup \mathcal{P}(\gamma) \\ \mathcal{P}(\beta^+\gamma) = \mathcal{P}(\beta) \\ \mathcal{P}(\mathrm{A}) = \bigcup_{i=1}^{n} \mathcal{P}(\alpha_i) \quad si \quad \mathrm{A} \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \end{cases}$$

Nous signalons que, pour faciliter la définition, nous avons étendu $\mathcal{P}$ à ε (bien que ε n'apparaisse jamais explicitement dans la grammaire) et γ peut valoir ε.

Nouvelle définition de $\mathcal{S}$:

$\mathrm{X} \to \ldots \mid \alpha\mathrm{AB}\beta$	$\models$	$\mathcal{S}(\mathrm{A}) = \mathcal{P}(\mathrm{B})$
$\mathrm{X} \to \ldots \mid \alpha\mathrm{A}\{\mathrm{B\ b}\ldots\}^+\beta$	$\models$	*Idem.*
$\mathrm{X} \to \ldots \mid \alpha\mathrm{A}[\beta]\gamma$	$\models$	$\mathcal{S}(\mathrm{A}) = \begin{cases} si \quad \gamma \overset{*}{\Longrightarrow} \varepsilon & alors \quad \mathcal{P}(\beta) \cup (\mathcal{P}(\gamma) \backslash \{\varepsilon\}) \cup \mathcal{S}(\mathrm{X}) \\ & sinon \quad \mathcal{P}(\beta) \cup \mathcal{P}(\gamma) \end{cases}$
$\mathrm{X} \to \ldots \mid \alpha\mathrm{A}\beta^*\gamma$	$\models$	*Idem.*
$\mathrm{X} \to \ldots \mid \alpha\mathrm{A}$	$\models$	$\mathcal{S}(\mathrm{A}) = \mathcal{S}(\mathrm{X})$
$\mathrm{X} \to \ldots \mid \alpha\mathrm{A}x\beta$	$\models$	$\mathcal{S}(\mathrm{A}) = \{x\}$
$\mathrm{X} \to \ldots \mid \alpha\mathrm{A}\beta^+\gamma$	$\models$	$\mathcal{S}(\mathrm{A}) = \mathcal{P}(\beta)$
$\mathrm{X} \to \ldots \mid \alpha\mathrm{A}\{\mathrm{B\ b}\ldots\}^*\beta$	$\models$	$\mathcal{S}(\mathrm{A}) = \begin{cases} si \quad \beta \overset{*}{\Longrightarrow} \varepsilon & alors \quad \mathcal{P}(\mathrm{B}) \cup (\mathcal{P}(\beta) \backslash \{\varepsilon\}) \cup \mathcal{S}(\mathrm{X}) \\ & sinon \quad \mathcal{P}(\mathrm{B}) \cup \mathcal{P}(\beta) \end{cases}$
$\mathrm{X} \to \ldots \mid \alpha\mathrm{A}\{[\mathrm{B}]\ \mathrm{b}\ldots\}\gamma$	$\models$	$\mathcal{S}(\mathrm{A}) = \begin{cases} si \quad \gamma \overset{*}{\Longrightarrow} \varepsilon & alors \quad \mathcal{P}(\mathrm{B}) \cup \{\mathrm{b}\} \cup (\mathcal{P}(\gamma) \backslash \{\varepsilon\}) \cup \mathcal{S}(\mathrm{X}) \\ & sinon \quad \mathcal{P}(\mathrm{B}) \cup \{\mathrm{b}\} \cup \mathcal{P}(\gamma) \end{cases}$
$\mathrm{X} \to \ldots \mid \alpha\{\mathrm{A\ a}\ldots\}^*\beta$	$\models$	$\mathcal{S}(\mathrm{A}) = \begin{cases} si \quad \beta \overset{*}{\Longrightarrow} \varepsilon & alors \quad \{\mathrm{a}\} \cup (\mathcal{P}(\beta) \backslash \{\varepsilon\}) \cup \mathcal{S}(\mathrm{X}) \\ & sinon \quad \{\mathrm{a}\} \cup \mathcal{P}(\beta) \end{cases}$
$\mathrm{X} \to \ldots \mid \alpha\{\mathrm{A\ a}\ldots\}^+\beta$	$\models$	*Idem.*
$\mathrm{X} \to \ldots \mid \alpha\{[\mathrm{A}]\ \mathrm{a}\ldots\}\beta$	$\models$	*Idem.*

Partout où cela est autorisé, nous pouvons lire le tableau précédent en remplaçant A par [A], A^* ou A^+.
Ceci était la définition générale, mais dans cette étude les grammaires ne possèdent pas de productions vides explicites (Cf. 2). Nous pouvons alors remplacer l'équation P3 par l'algorithme suivant : à chaque occurrence des expressions rationnelles α^*, α^+, $[\alpha]$, $\{\text{A a} \ldots\}^*$, et $\{\text{A a} \ldots\}^+$, les considérer comme étant produites par une règle dédiée (sans partage) et vérifier les contraintes associées suivantes :

Règle rationnelle	Contrainte
$\text{X} \to \alpha^*$	$\mathcal{P}(\alpha) \cap \mathcal{S}(X) = \emptyset$
$\text{X} \to \alpha^+$	$\mathcal{P}(\alpha) \cap \mathcal{S}(X) = \emptyset$
$\text{X} \to [\alpha]$	$\mathcal{P}(\alpha) \cap \mathcal{S}(X) = \emptyset$
$\text{X} \to \{\text{A a} \ldots\}^*$	$(\mathcal{P}(A) \cup \{a\}) \cap \mathcal{S}(X) = \emptyset$
$\text{X} \to \{\text{A a} \ldots\}^+$	$\{a\} \cap \mathcal{S}(X) = \emptyset$
$\text{X} \to \{[\text{A}]\ \text{a} \ldots\}$	$(\mathcal{P}(A) \cup \{a\}) \cap \mathcal{S}(X) = \emptyset$

Le fait de ne pas considérer de partage facilite le travail s'il est réalisé « à la main » : le coût est identique, mais la tâche aura été plus localisée à chaque étape. Un programme pourrait se passer de ce choix, bien entendu.

4.2 Vérification de la propriété LL(1) de la nouvelle grammaire ASN.1 :1990

4.2.1 Équation P1

Nous présentons dans le tableau ci-dessous les non-terminaux qui sont produits en tête de chaque production. Nous vérifions aisément, par fermeture transitive, qu'il n'y a pas de récursivité gauche qui apparaît.

Règle	Tête
ModuleDefinition	ModuleIdentifier
ModuleIdentifier	
ObjIdComponent	
TagDefault	
ModuleBody	Exports, Imports, Assignment
Exports	
Imports	
SymbolsFromModule	Symbol
Symbol	
Assignment	

Règle	Tête
Type	AuxType
AuxType	BuiltInType, SetSeq
SetSeq	
TypeSuf	SubtypeSpec
BuiltInType	
NamedType	AuxType
NamedNumber	
AuxNamedNum	
NamedBit	
ElementType	NamedType
ElementTypeSuf	
Class	
ClassNumber	
Value	AuxVal0
AuxVal0	BuiltInValue, AuxType
AuxVal1	SpecVal
AuxVal2	
AuxVal11	
SpecVal	SubtypeSpec
BuiltInValue	
BetBraces	AuxVal0
AuxBet1	AuxNamed, AuxVal2, AuxVal0
AuxBet2	SpecVal
AuxBet3	ObjIdComponent, AuxNamed
AuxBet11	ObjIdComponent, AuxVal2, AuxNamed
AuxBet21	
AuxNamed	
NamedValue	AuxVal0
NamedValSuf	Value, AuxVal2
SubtypeSpec	
SubtypeValueSet	SVSAux
SubValSetSuf	
UpperEndValue	Value
InnerTypeSuf	
MultipleTypeConstraints	
NamedConstraint	SubtypeSpec, PresenceConstraint
PresenceConstraint	
SVSAux	BuiltInValue, AuxType
SVSAux1	SubtypeSpec
SVSAux2	
SVSAux3	SubtypeSpec, SubValSetSuf
SVSAux11	
SVSAux21	Type

4.2.2 Équation P2

Pour toutes les règles nous écrivons la contrainte imposée par l'équation P2, puis calculons les ensembles $\mathcal{P}(\alpha_i)$ nécessaires pour les résoudre. Pour faciliter la lecture nous ne ferons pas figurer la contrainte si tous les α_i sont des terminaux (trivial). De même, nous réduirons directement les $\mathcal{P}(\alpha_i)$, où α_i est une expression rationnelle spéciale, à leur forme sans opérateur (Cf. 4.1.4).

Règle	Contrainte
ModuleDefinition	
ModuleIdentifier	
ObjIdComponent	
TagDefault	
ModuleBody	
Exports	
Imports	
SymbolsFromModule	
Symbol	
Assignment	
Type	$\{\texttt{lower}, \texttt{upper}, \text{NULL}\} \cap \mathcal{P}(\text{AuxType}) = \emptyset$
AuxType	$\{\text{“[”}\} \cap \mathcal{P}(\text{BuiltInType}) \cap \mathcal{P}(\text{SetSeq}) = \emptyset$
SetSeq	
TypeSuf	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“\{”, SIZE, OF}\} = \emptyset$
BuiltInType	
NamedType	$\{\texttt{lower}, \texttt{upper}, \text{NULL}\} \cap \mathcal{P}(\text{AuxType}) = \emptyset$
NamedNumber	
AuxNamedNum	
NamedBit	
ElementType	$\mathcal{P}(\text{NamedType}) \cap \{\text{COMPONENTS}\} = \emptyset$
ElementTypeSuf	
Class	
ClassNumber	

Règle	Contrainte
Value	$\mathcal{P}(\text{AuxVal0}) \cap \{\texttt{upper}, \texttt{lower}, \text{“-”}, \texttt{number}\} = \emptyset$
AuxVal0	$\mathcal{P}(\text{BuiltInValue}) \cap \mathcal{P}(\text{AuxType}) \cap \{\text{NULL}\} = \emptyset$
AuxVal1	$\mathcal{P}(\text{SpecVal}) \cap \{\text{“.”}\} = \emptyset$
AuxVal2	
AuxVal11	
SpecVal	
BuiltInValue	
BetBraces	$\mathcal{P}(\text{AuxVal0}) \cap \{\text{“-”}, \texttt{lower}, \texttt{upper}, \texttt{number}\} = \emptyset$
AuxBet1	$\mathcal{P}(\text{AuxNamed}) \cap \mathcal{P}(\text{AuxVal2}) \cap \mathcal{P}(\text{AuxVal0})$ $\cap \{\text{“(”}, \text{“-”}, \texttt{lower}, \texttt{upper}, \texttt{number}\} = \emptyset$
AuxBet2	$\mathcal{P}(\text{SpecVal}) \cap \{\text{“.”}\} = \emptyset$
AuxBet3	$\mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{P}(\text{AuxNamed}) = \emptyset$
AuxBet11	$\{\text{“(”}\} \cap \mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{P}(\text{AuxVal2})$ $\cap \mathcal{P}(\text{AuxNamed}) = \emptyset$
AuxBet21	
AuxNamed	
NamedValue	$\mathcal{P}(\text{AuxVal0}) \cap \{\text{“-”}, \texttt{lower}, \texttt{upper}, \texttt{number}\} = \emptyset$
NamedValSuf	$\mathcal{P}(\text{Value}) \cap \mathcal{P}(\text{AuxVal2}) = \emptyset$
SubtypeSpec	
SubtypeValueSet	$\{\text{INCLUDES, MIN, FROM, SIZE, WITH}\} \cap \mathcal{P}(\text{SVSAux}) = \emptyset$
SubValSetSuf	
UpperEndValue	$\mathcal{P}(\text{Value}) \cap \{\text{MAX}\} = \emptyset$
InnerTypeSuf	
MultipleTypeConstraints	
NamedConstraint	$\{\texttt{lower}\} \cap \mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{P}(\text{PresenceConstraint}) = \emptyset$
PresenceConstraint	
SVSAux	$\mathcal{P}(\text{BuiltInValue}) \cap \mathcal{P}(\text{AuxType})$ $\cap \{\text{NULL}, \texttt{upper}, \texttt{lower}, \text{“-”}, \texttt{number}\} = \emptyset$
SVSAux1	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“.”}, \text{“ :”}\} = \emptyset$
SVSAux2	
SVSAux3	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“ :”}\} \cap \mathcal{P}(\text{SubValSetSuf}) = \emptyset$
SVSAux11	
SVSAux21	$\mathcal{P}(\text{Type}) \cap \{\text{“..”}\} = \emptyset$

Finalement, après simplification, le système d'équations suivant doit être satisfait :

$$\begin{cases}
(1) & \{\text{“[”}\} \cap \mathcal{P}(\text{BuiltInType}) \cap \mathcal{P}(\text{SetSeq}) = \emptyset \\
(2) & \mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“\{”, “.”, “ :”, SIZE, OF}\} = \emptyset \\
(3) & \mathcal{P}(\text{NamedType}) \cap \{\text{COMPONENTS}\} = \emptyset \\
(4) & \mathcal{P}(\text{SpecVal}) \cap \{\text{“.”}\} = \emptyset \\
(5) & \mathcal{P}(\text{AuxNamed}) \cap \mathcal{P}(\text{AuxVal2}) \cap \mathcal{P}(\text{AuxVal0}) \cap \{\text{“(”, “-”, } \texttt{lower}, \texttt{upper}, \texttt{number}\} = \emptyset \\
(6) & \{\text{“(”}\} \cap \mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{P}(\text{AuxVal2}) \cap \mathcal{P}(\text{AuxNamed}) = \emptyset \\
(7) & \mathcal{P}(\text{Value}) \cap \mathcal{P}(\text{AuxVal2}) = \emptyset \\
(8) & \mathcal{P}(\text{Value}) \cap \{\text{MAX}\} = \emptyset \\
(9) & \{\text{INCLUDES, MIN, FROM, SIZE, WITH}\} \cap \mathcal{P}(\text{SVSAux}) = \emptyset \\
(10) & \{\texttt{lower}\} \cap \mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{P}(\text{PresenceConstraint}) = \emptyset \\
(11) & \mathcal{P}(\text{BuiltInValue}) \cap \mathcal{P}(\text{AuxType}) \cap \{\text{NULL}, \texttt{upper}, \texttt{lower}, \text{“-”}, \texttt{number}\} = \emptyset \\
(12) & \mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“ :”}\} \cap \mathcal{P}(\text{SubValSetSuf}) = \emptyset \\
(13) & \mathcal{P}(\text{Type}) \cap \{\text{“..”}\} = \emptyset
\end{cases}$$

Or :

$$\begin{aligned}
\mathcal{P}(\text{BuiltInType}) &= \{\ \text{BOOLEAN, INTEGER, BIT, OCTET, CHOICE, ANY,} \\
&\quad \text{OBJECT, ENUMERATED, REAL, EXTERNAL,} \\
&\quad \text{“NumericString”, “PrintableString”, “TeletexString”,} \\
&\quad \text{“T61String”, “VideotexString”, “VisibleString”,} \\
&\quad \text{“ISO646String”, “IA5String”, “GraphicString”,} \\
&\quad \text{“GeneralString”, “UTCTime”, “GeneralizedTime”,} \\
&\quad \text{“ObjectDescriptor” }\} \\
\mathcal{P}(\text{SetSeq}) &= \{\ \text{SET, SEQUENCE }\}
\end{aligned}$$

Donc l'équation (1) est vérifiée.

$$\mathcal{P}(\text{SubtypeSpec}) = \{\ \text{“(” }\}$$

Donc l'équation (2) est vérifiée.

$$
\begin{array}{lcl}
\mathcal{P}(\text{NamedType}) & = & \{\texttt{lower}, \texttt{upper}, \text{NULL}\} \cup \mathcal{P}(\text{AuxType}) \\
\mathcal{P}(\text{AuxType}) & = & \{\text{“[”}\} \cup \mathcal{P}(\text{BuiltInType}) \cup \mathcal{P}(\text{SetSeq}) \\
 & = & \{\text{ “[”, SET, SEQUENCE, BOOLEAN, INTEGER, BIT,} \\
 & & \text{OCTET, CHOICE, ANY, OBJECT, ENUMERATED, REAL,} \\
 & & \text{EXTERNAL, “NumericString”, “PrintableString”,} \\
 & & \text{“TeletexString”, “T61String”, “VideotexString”,} \\
 & & \text{“VisibleString”, “ISO646String”, “IA5String”,} \\
 & & \text{“GraphicString”, “GeneralString”, “UTCTime”,} \\
 & & \text{“GeneralizedTime”, “ObjectDescriptor” }\}
\end{array}
$$

Donc l'équation (3) est vérifiée.

$$
\begin{array}{lcl}
\mathcal{P}(\text{SpecVal}) & = & \mathcal{P}(\text{SubtypeSpec}) \cup \{\text{“:”}\} \\
 & = & \{\text{ “(”, “:” }\}
\end{array}
$$

Donc l'équation (4) est vérifiée.

$$
\begin{array}{lcl}
\mathcal{P}(\text{AuxNamed}) & = & \{\text{ “,” }\} \\
\mathcal{P}(\text{AuxVal2}) & = & \{\text{ “<”, “:” }\} \\
\mathcal{P}(\text{BuiltInValue}) & = & \{\text{ TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,} \\
 & & \texttt{basednum}, \texttt{string}, \text{“\{” }\} \\
\mathcal{P}(\text{AuxVal0}) & = & \mathcal{P}(\text{BuiltInValue}) \cup \mathcal{P}(\text{AuxType}) \cup \{\text{NULL}\} \\
 & = & \{\text{ TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,} \\
 & & \texttt{basednum}, \texttt{string}, \text{“\{”, NULL, “[”, SET, SEQUENCE,} \\
 & & \text{BOOLEAN, INTEGER, BIT, OCTET, CHOICE, ANY,} \\
 & & \text{OBJECT, ENUMERATED, REAL, EXTERNAL,} \\
 & & \text{“NumericString”, “PrintableString”, “TeletexString”,} \\
 & & \text{“T61String”, “VideotexString”, “VisibleString”,} \\
 & & \text{“ISO646String”, “IA5String”, “GraphicString”,} \\
 & & \text{“GeneralString”, “UTCTime”, “GeneralizedTime”,} \\
 & & \text{“ObjectDescriptor” }\}
\end{array}
$$

Donc les équations (5) et (11) sont vérifiées.

$$\mathcal{P}(\text{ObjIdComponent}) = \{\texttt{number}, \texttt{upper}, \texttt{lower}\}$$

Donc l'équation (6) est vérifiée.

$$\begin{aligned}\mathcal{P}(\text{Value}) &= \mathcal{P}(\text{AuxVal0}) \cup \{\texttt{upper}, \texttt{lower}, \texttt{number}, \text{“-”}\} \\ &= \{\ \text{TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY, } \texttt{basednum}, \\ &\quad \texttt{string}, \text{“\{”, NULL, “[”, SET, SEQUENCE, BOOLEAN,} \\ &\quad \text{INTEGER, BIT, OCTET, CHOICE, ANY, OBJECT,} \\ &\quad \text{ENUMERATED, REAL, EXTERNAL,} \\ &\quad \text{“NumericString”, “PrintableString”, “TeletexString”,} \\ &\quad \text{“T61String”, “VideotexString”, “VisibleString”,} \\ &\quad \text{“ISO646String”, “IA5String”, “GraphicString”,} \\ &\quad \text{“GeneralString”, “UTCTime”, “GeneralizedTime”,} \\ &\quad \text{“ObjectDescriptor”, } \texttt{upper}, \texttt{lower}, \texttt{number}, \text{“-”}\ \}\end{aligned}$$

Donc les équations (7) et (8) sont vérifiées.

$$\begin{aligned}\mathcal{P}(\text{SVSAux}) &= \mathcal{P}(\text{BuiltInValue}) \cup \mathcal{P}(\text{Auxtype}) \\ &\quad \cup\ \{\ \text{NULL, } \texttt{upper}, \texttt{lower}, \texttt{number}, \text{“-”}\ \} \\ &= \{\ \text{TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,} \\ &\quad \texttt{basednum}, \texttt{string}, \text{“\{”, “[”, SET, SEQUENCE, BOOLEAN,} \\ &\quad \text{INTEGER, BIT, OCTET, CHOICE, ANY, OBJECT,} \\ &\quad \text{ENUMERATED, REAL, EXTERNAL, “NumericString”,} \\ &\quad \text{“PrintableString”, “TeletexString”, “T61String”,} \\ &\quad \text{“VideotexString”, “VisibleString”, “ISO646String”,} \\ &\quad \text{“IA5String”, “GraphicString”, “GeneralString”,} \\ &\quad \text{“UTCTime”, “GeneralizedTime”,“ObjectDescriptor”,} \\ &\quad \text{NULL, } \texttt{upper}, \texttt{lower}, \texttt{number}, \text{“-”}\ \}\end{aligned}$$

Donc l'équation (9) est vérifiée.

$$\mathcal{P}(\text{PresenceConstraint}) = \{\text{PRESENT, ABSENT, OPTIONAL}\}$$

Donc l'équation (10) est vérifiée.

$$\mathcal{P}(\text{SubValSetSuf}) = \{\text{“<”, “..”}\}$$

Donc l'équation (12) est vérifiée.

$$
\begin{aligned}
\mathcal{P}(\text{Type}) &= \{\texttt{lower}, \texttt{upper}, \text{NULL}\} \cup \mathcal{P}(\text{AuxType}) \\
&= \{\ \texttt{lower}, \texttt{upper}, \text{NULL}, \\
&\quad \text{“[”, SET, SEQUENCE, BOOLEAN, INTEGER, BIT, OCTET,} \\
&\quad \text{CHOICE, ANY, OBJECT, ENUMERATED, REAL, EXTERNAL,} \\
&\quad \text{“NumericString”, “PrintableString”, “TeletexString”,} \\
&\quad \text{“T61String”, “VideotexString”, “VisibleString”,} \\
&\quad \text{“ISO646String”, “IA5String”,“GraphicString”,} \\
&\quad \text{“GeneralString”, “UTCTime”, “GeneralizedTime”,} \\
&\quad \text{“ObjectDescriptor” }\}
\end{aligned}
$$

Donc l'équation (13) est vérifiée.

4.2.3 Équation P3

Nous donnons ici pour chaque production de chaque règle les contraintes imposées par l'équation P3 (Cf. 4.1.4). Pour ne pas charger inutilement le tableau, les équations redondantes au sein d'une même règle et les équations triviales n'apparaîtront pas. Nous simplifierons de même ce qui peut l'être localement.

Règle	Contraintes
ModuleDefinition	$\mathcal{P}(\text{TagDefault}) \cap \{\text{“::=”}\} = \emptyset$ $\mathcal{P}(\text{ModuleBody}) \cap \{\text{END}\} = \emptyset$
ModuleIdentifier	$\{\text{“\{”}\} \cap \mathcal{S}(\text{ModuleIdentifier}) = \emptyset$
ObjIdComponent	$\{\text{“(”}\} \cap \mathcal{S}(\text{ObjIdComponent}) = \emptyset$
TagDefault	
ModuleBody	$\mathcal{P}(\text{Exports}) \cap (\mathcal{P}(\text{Imports}) \cup \mathcal{P}(\text{Assignment})) = \emptyset$ $\mathcal{P}(\text{Imports}) \cap \mathcal{P}(\text{Assignment}) = \emptyset$
Exports	$(\mathcal{P}(\text{Symbol}) \cup \{\text{“,”}\}) \cap \{\text{“;”}\} = \emptyset$
Imports	$\mathcal{P}(\text{SymbolsFromModule}) \cap \{\text{“,”}\} = \emptyset$
SymbolsFromModule	
Symbol	
Assignment	
Type	$\{\text{“.”}\} \cap (\mathcal{P}(\text{SubtypeSpec}) \cup \mathcal{S}(\text{Type})) = \emptyset$ $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{Type}) = \emptyset$
AuxType	$\mathcal{P}(\text{Class}) \cap \mathcal{P}(\text{ClassNumber}) = \emptyset$ $\mathcal{P}(\text{TagDefault}) \cap \mathcal{P}(\text{Type}) = \emptyset$ $\mathcal{P}(\text{TypeSuf}) \cap \mathcal{S}(\text{AuxType}) = \emptyset$
SetSeq	
TypeSuf	$\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{TypeSuf}) = \emptyset$ $(\mathcal{P}(\text{ElementType}) \cup \{\text{“,”}\}) \cap \{\text{“}\}\text{”}\} = \emptyset$
BuiltInType	$\{\text{“\{”}\} \cap \mathcal{S}(\text{BuiltInType}) = \emptyset$ $\{\text{DEFINED}\} \cap \mathcal{S}(\text{BuiltInType}) = \emptyset$
NamedType	$\{\text{“<”}\} \cap \mathcal{P}(\text{Type}) = \emptyset$ $\{\text{“.”}\} \cap (\mathcal{P}(\text{SubtypeSpec}) \cup \mathcal{S}(\text{NamedType})) = \emptyset$ $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{NamedType}) = \emptyset$
NamedNumber	
AuxNamedNum	
NamedBit	
ElementType	$\mathcal{P}(\text{ElementTypeSuf}) \cap \mathcal{S}(\text{ElementType}) = \emptyset$
ElementTypeSuf	
Class	
ClassNumber	

Règle	Contraintes
Value	$\mathcal{P}(\text{AuxVal2}) \cap \mathcal{S}(\text{Value}) = \emptyset$
AuxVal0	$\mathcal{P}(\text{SpecVal}) \cap \mathcal{S}(\text{AuxVal0}) = \emptyset$
AuxVal1	
AuxVal2	
AuxVal11	
SpecVal	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“:”}\} = \emptyset$
BuiltInValue	
BetBraces	$\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{BetBraces}) = \emptyset$ $\mathcal{P}(\text{AuxBet1}) \cap \mathcal{S}(\text{BetBraces}) = \emptyset$ $\mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{BetBraces}) = \emptyset$
AuxBet1	$\mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet1}) = \emptyset$ $\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet1}) = \emptyset$ $\mathcal{P}(\text{AuxBet11}) \cap \mathcal{S}(\text{AuxBet1}) = \emptyset$ $\mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{AuxBet1}) = \emptyset$
AuxBet2	$\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet2}) = \emptyset$
AuxBet3	$\mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet3}) = \emptyset$
AuxBet11	$\mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet11}) = \emptyset$ $\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet11}) = \emptyset$
AuxBet21	$\mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet21}) = \emptyset$ $\mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{AuxBet21}) = \emptyset$
AuxNamed	$\{\text{“,”}\} \cap \mathcal{S}(\text{AuxNamed}) = \emptyset$
NamedValue	$\mathcal{P}(\text{NamedValSuf}) \cap \mathcal{S}(\text{NamedValue}) = \emptyset$
NamedValSuf	
SubtypeSpec	
SubtypeValueSet	
SubValSetSuf	$\{\text{“<”}\} \cap \mathcal{P}(\text{UpperEndValue}) = \emptyset$
UpperEndValue	
InnerTypeSuf	
MultipleTypeConstraints	$\{\text{“...”}, \text{“}\}\text{”}\} \cap \mathcal{P}(\text{NamedConstraint}) = \emptyset$
NamedConstraint	$\mathcal{P}(\text{SubtypeSpec}) \cap (\mathcal{P}(\text{PresenceConstraint})$ $\cup\ \mathcal{S}(\text{NamedConstraint})) = \emptyset$ $\mathcal{P}(\text{PresenceConstraint}) \cap \mathcal{S}(\text{NamedConstraint}) = \emptyset$
PresenceConstraint	
SVSAux	$\mathcal{P}(\text{SubValSetSuf}) \cap \mathcal{S}(\text{SVSAux}) = \emptyset$ $\mathcal{P}(\text{SVSAux3}) \cap \mathcal{S}(\text{SVSAux}) = \emptyset$ $\mathcal{P}(\text{SVSAux2}) \cap \mathcal{S}(\text{SVSAux}) = \emptyset$
SVSAux1	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“:”}\} = \emptyset$
SVSAux2	$\{\text{“<”}\} \cap \mathcal{P}(\text{UpperEndValue}) = \emptyset$
SVSAux3	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“:”}\} = \emptyset$
SVSAux11	$\mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“:”}\} = \emptyset$ $\mathcal{P}(\text{SubValSetSuf}) \cap \mathcal{S}(\text{SVSAux11}) = \emptyset$
SVSAux21	$\{\text{“<”}\} \cap \mathcal{P}(\text{UpperEndValue}) = \emptyset$

Finalement, après simplifications, le système d'équations suivant doit être satisfait :

$$
\begin{array}{ll}
(1) & \mathcal{P}(\text{TagDefault}) \cap \{\text{“::=”}\} = \emptyset \\
(2) & \mathcal{P}(\text{ModuleBody}) \cap \{\text{END}\} = \emptyset \\
(3) & \{\text{“\{”}\} \cap \mathcal{S}(\text{ModuleIdentifier}) = \emptyset \\
(4) & \{\text{“(”}\} \cap \mathcal{S}(\text{ObjIdComponent}) = \emptyset \\
(5) & \mathcal{P}(\text{Exports}) \cap \mathcal{P}(\text{Imports}) = \emptyset \\
(6) & \mathcal{P}(\text{Exports}) \cap \mathcal{P}(\text{Assignment}) = \emptyset \\
(7) & \mathcal{P}(\text{Imports}) \cap \mathcal{P}(\text{Assignment}) = \emptyset \\
(8) & \mathcal{P}(\text{TypeSuf}) \cap \mathcal{S}(\text{AuxType}) = \emptyset \\
(9) & \mathcal{P}(\text{SymbolsFromModule}) \cap \{\text{“;”}\} = \emptyset \\
(10) & \mathcal{P}(\text{SubtypeSpec}) \cap \{\text{“.”}, \text{“:”}\} = \emptyset \\
(11) & \mathcal{S}(\text{Type}) \cap \{\text{“.”}\} = \emptyset \\
(12) & \mathcal{P}(\text{Class}) \cap \mathcal{P}(\text{ClassNumber}) = \emptyset \\
(13) & \mathcal{P}(\text{TagDefault}) \cap \mathcal{P}(\text{Type}) = \emptyset \\
(14) & \mathcal{P}(\text{Type}) \cap \{\text{“<”}\} = \emptyset \\
(15) & \mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{TypeSuf}) = \emptyset \\
(16) & \mathcal{P}(\text{ElementType}) \cap \{\text{“}\}\text{”}\} = \emptyset \\
(17) & \{\text{“\{”}, \text{DEFINED}\} \cap \mathcal{S}(\text{BuiltInType}) = \emptyset \\
(18) & \{\text{“.”}\} \cap \mathcal{S}(\text{NamedType}) = \emptyset \\
(19) & \mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{NamedType}) = \emptyset \\
(20) & \mathcal{P}(\text{ElementTypeSuf}) \cap \mathcal{S}(\text{ElementType}) = \emptyset \\
(21) & \mathcal{P}(\text{AuxVal2}) \cap \mathcal{S}(\text{Value}) = \emptyset \\
(22) & \mathcal{P}(\text{SpecVal}) \cap \mathcal{S}(\text{AuxVal0}) = \emptyset \\
(23) & \mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{BetBraces}) = \emptyset \\
(24) & \mathcal{P}(\text{AuxBet1}) \cap \mathcal{S}(\text{BetBraces}) = \emptyset \\
(25) & \mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{BetBraces}) = \emptyset
\end{array}
$$

$$
\begin{array}{ll}
(26) & \mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet1}) = \emptyset \\
(27) & \mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet1}) = \emptyset \\
(28) & \mathcal{P}(\text{AuxBet11}) \cap \mathcal{S}(\text{AuxBet1}) = \emptyset \\
(29) & \mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{AuxBet1}) = \emptyset \\
(30) & \mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet2}) = \emptyset \\
(31) & \mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet3}) = \emptyset \\
(32) & \mathcal{P}(\text{ObjIdComponent}) \cap \mathcal{S}(\text{AuxBet11}) = \emptyset \\
(33) & \mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet11}) = \emptyset \\
(34) & \mathcal{P}(\text{AuxNamed}) \cap \mathcal{S}(\text{AuxBet21}) = \emptyset
\end{array}
$$

(35) $\mathcal{P}(\text{AuxBet3}) \cap \mathcal{S}(\text{AuxBet21}) = \emptyset$
(36) $\{\text{“,”}\} \cap \mathcal{S}(\text{AuxNamed}) = \emptyset$
(37) $\mathcal{P}(\text{NamedValSuf}) \cap \mathcal{S}(\text{NamedValue}) = \emptyset$
(38) $\{\text{“<”}\} \cap \mathcal{P}(\text{UpperEndValue}) = \emptyset$
(39) $\{\text{“...”}, \text{“}\}\text{”}\} \cap \mathcal{P}(\text{NamedConstraint}) = \emptyset$
(40) $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{P}(\text{PresenceConstraint}) = \emptyset$
(41) $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{NamedConstraint}) = \emptyset$
(42) $\mathcal{P}(\text{PresenceConstraint}) \cap \mathcal{S}(\text{NamedConstraint}) = \emptyset$
(43) $\mathcal{P}(\text{SubValSetSuf}) \cap \mathcal{S}(\text{SVSAux}) = \emptyset$
(44) $\mathcal{P}(\text{SVSAux3}) \cap \mathcal{S}(\text{SVSAux}) = \emptyset$
(45) $\mathcal{P}(\text{SVSAux2}) \cap \mathcal{S}(\text{SVSAux}) = \emptyset$
(46) $\mathcal{P}(\text{SubValSetSuf}) \cap \mathcal{S}(\text{SVSAux11}) = \emptyset$
(47) $\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{Type}) = \emptyset$

Les équations (10), (14) et (40) sont immédiatement vérifiées à l'aide des ensembles $\mathcal{P}$ précédemment calculés (Cf. 4.2.2). Nous calculons alors les ensembles $\mathcal{P}$ manquant :

$$\mathcal{P}(\text{TagDefault}) = \{\text{EXPLICIT, IMPLICIT}\}$$

Donc les équations (1) et (13) sont vérifiées.

$$\mathcal{P}(\text{Exports}) = \{\text{EXPORTS}\}$$
$$\mathcal{P}(\text{Imports}) = \{\text{IMPORTS}\}$$

Donc l'équation (5) est vérifiée.

$$\mathcal{P}(\text{Assignment}) = \{\texttt{upper}, \texttt{lower}\}$$

Donc les équations (6) et (7) sont vérifiées.

$$\begin{aligned} \mathcal{P}(\text{ModuleBody}) &= \mathcal{P}(\text{Exports}) \cup \mathcal{P}(\text{Imports}) \cup \mathcal{P}(\text{Assignment}) \\ &= \{\text{EXPORTS, IMPORTS, } \texttt{upper}, \texttt{lower}\} \end{aligned}$$

Donc l'équation (2) est vérifiée.

$$\begin{aligned} \mathcal{P}(\text{Symbol}) &= \{\ \texttt{upper}, \texttt{lower}\ \} \\ \mathcal{P}(\text{SymbolsFromModule}) &= \mathcal{P}(\text{Symbol}) \cup \{\text{“;”}\} \\ &= \{\ \texttt{upper}, \texttt{lower}, \text{“;”}\ \} \end{aligned}$$

Donc l'équation (9) est vérifiée.

$\mathcal{P}$(Class) = { UNIVERSAL, APPLICATION, PRIVATE }
$\mathcal{P}$(ClassNumber) = { `number`, `upper`, `lower` }

Donc l'équation (12) est vérifiée.

$$\mathcal{P}(\text{ElementType}) = \mathcal{P}(\text{NamedType}) \cup \{\text{COMPONENTS}\}$$

Donc l'équation (16) est vérifiée.

$$\mathcal{P}(\text{UpperEndValue}) = \mathcal{P}(\text{Value}) \cup \{MAX\}$$

Donc l'équation (38) est vérifiée.

$\mathcal{P}$(SubtypeSpec) = { "(" }
$\mathcal{P}$(NamedConstraint) = {`lower`} $\cup$ $\mathcal{P}$(SubtypeSpec) $\cup$ $\mathcal{P}$(PresenceConstraint)
= { `lower`, "(", PRESENT, ABSENT, OPTIONAL }

Donc l'équation (39) est vérifiée.

De plus :

$\mathcal{P}$(TypeSuf) = $\mathcal{P}$(SubtypeSpec) $\cup$ {"{", SIZE, OF}
= { "(", "{", SIZE, OF }

$\mathcal{P}$(ElementTypeSuf) = { OPTIONAL, DEFAULT }

$\mathcal{P}$(AuxVal2) = { "<", ":" }

$\mathcal{P}$(SpecVal) = $\mathcal{P}$(SubtypeSpec) $\cup$ {":"}
= { "(", ":" }

$\mathcal{P}$(AuxBet1) = {"(", "-", `upper`, `lower`, `number`} $\cup$ $\mathcal{P}$(AuxNamed)
$\cup$ $\mathcal{P}$(AuxVal2) $\cup$ $\mathcal{P}$(AuxVal0)
= { "(", "-", `upper`, `lower`, `number`, ",", "<", ":",
TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,
`basednum`, `string`, "{", NULL, "[", SET, SEQUENCE,
BOOLEAN, INTEGER, BIT, OCTET, CHOICE,
ANY, OBJECT, ENUMERATED, REAL, EXTERNAL,
"NumericString", "PrintableString", "TeletexString",
"T61String", "VideotexString", "VisibleString",
"ISO646String", "IA5String", "GraphicString",
"GeneralString", "UTCTime", "GeneralizedTime",
"ObjectDescriptor" }

$$
\begin{array}{lcl}
\mathcal{P}(\text{AuxBet3}) & = & \mathcal{P}(\text{ObjIdComponent}) \cup \mathcal{P}(\text{AuxNamed}) \\
 & = & \{ \texttt{number}, \texttt{upper}, \texttt{lower}, \text{“,”} \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}(\text{NamedValSuf}) & = & \mathcal{P}(\text{Value}) \cup \mathcal{P}(\text{AuxVal2}) \\
 & = & \{ \text{TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,} \\
 & & \texttt{basednum}, \texttt{string}, \text{“\{”, NULL, “[”, SET, SEQUENCE,} \\
 & & \text{BOOLEAN, INTEGER, BIT, OCTET, CHOICE,} \\
 & & \text{ANY, OBJECT, ENUMERATED, REAL, EXTERNAL,} \\
 & & \text{“NumericString”, “PrintableString”, “TeletexString”,} \\
 & & \text{“T61String”, “VideotexString”, “VisibleString”,} \\
 & & \text{“ISO646String”, “IA5String”, “GraphicString”,} \\
 & & \text{“GeneralString”, “UTCTime”, “GeneralizedTime”,} \\
 & & \text{“ObjectDescriptor”}, \texttt{upper}, \texttt{lower}, \texttt{number}, \text{“-”, “<”, “:” } \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}(\text{AuxBet11}) & = & \{\text{“(”}\} \cup \mathcal{P}(\text{ObjIdComponent}) \cup \mathcal{P}(\text{AuxVal2}) \cup \mathcal{P}(\text{AuxNamed}) \\
 & = & \{ \text{“(”}, \texttt{number}, \texttt{upper}, \texttt{lower}, \text{“<”, “:”, “,” } \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}(\text{SubValSetSuf}) & = & \{ \text{“<”, “..” } \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}(\text{SVSAux3}) & = & \mathcal{P}(\text{SubtypeSpec}) \cup \mathcal{P}(\text{SubValSetSuf}) \cup \{\text{“:”}\} \\
 & = & \{ \text{“(”, “<”, “..”, “:” } \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{P}(\text{SVSAux2}) & = & \{ \text{“:”, “..”, “<” } \}
\end{array}
$$

Donc $\mathcal{P}(\text{SubValSetSuf}) \subset \mathcal{P}(\text{SVSAux2}) \subset \mathcal{P}(\text{SVSAux3})$,
ce qui permet de supprimer les équations (43) et (45), qui sont impliquées par (44).

De plus, il est pertinent de noter ici :

$$
\begin{array}{lcl}
\mathcal{S}(\text{SVSAux11}) & = & \mathcal{S}(\text{SVSAux1}) \\
\mathcal{S}(\text{SVSAux1}) & = & \mathcal{S}(\text{SVSAux})
\end{array}
$$

Donc nous pouvons supprimer l'équation (46) car elle est impliquée par (44).

Nous calculons maintenant quelques ensembles $\mathcal{S}$ qui, à l'aide des ensembles $\mathcal{P}$ précédemment calculés, nous permettent de conclure en une étape.

$$
\begin{array}{lcl}
\mathcal{S}(\text{SymbolsFromModule}) & = & \mathcal{P}(\text{SymbolsFromModule}) \cup \{\text{“;”}\} \\
 & = & \{ \texttt{upper}, \texttt{lower}, \text{“;” } \} \\
\mathcal{S}(\text{ModuleIdentifier}) & = & \{\text{DEFINITIONS}\} \cup \mathcal{S}(\text{SymbolsFromModule}) \\
 & = & \{ \text{DEFINITIONS}, \texttt{upper}, \texttt{lower}, \text{“;” } \}
\end{array}
$$

Donc l'équation (3) est vérifiée.

$$\mathcal{S}(\text{ElementType}) = \{\text{“,”, “}\}\text{”}\}$$

Donc l'équation (20) est vérifiée.

$$\begin{array}{lcl} \mathcal{S}(\text{NamedType}) & = & \{\text{“,”, “}\}\text{”}\} \cup \mathcal{P}(\text{ElementTypeSuf}) \cup \mathcal{S}(\text{ElementType}) \\ & = & \{\text{ “,”, “}\}\text{”, OPTIONAL, DEFAULT }\} \end{array}$$

Donc les équations (18) et (19) sont vérifiées.

$$\mathcal{S}(\text{BetBraces}) = \{\text{“}\}\text{”}\}$$

Donc les équations (23), (24) et (25) sont vérifiées

$$\begin{array}{lcl} \mathcal{S}(\text{AuxBet1}) & = & \mathcal{S}(\text{BetBraces}) \\ & = & \{\text{ “}\}\text{” }\} \end{array}$$

Donc les équations (26), (27), (28), et (29) sont vérifiées.

$$\begin{array}{lcl} \mathcal{S}(\text{AuxBet11}) & = & \mathcal{S}(\text{AuxBet1}) \\ & = & \{\text{ “}\}\text{” }\} \end{array}$$

Donc les équations (32) et (33) sont vérifiées.

$$\begin{array}{lcl} \mathcal{S}(\text{AuxBet2}) & = & \mathcal{S}(\text{BetBraces}) \cup \mathcal{S}(\text{AuxBet1}) \\ & = & \{\text{ “}\}\text{” }\} \end{array}$$

Donc l'équation (30) est vérifiée.

$$\begin{array}{lcl} \mathcal{S}(\text{AuxBet21}) & = & \mathcal{S}(\text{AuxBet2}) \\ & = & \{\text{ “}\}\text{” }\} \end{array}$$

Donc les équations (34) et (35) sont vérifiées.

$$\begin{array}{lcl} \mathcal{S}(\text{AuxBet3}) & = & \mathcal{S}(\text{BetBraces}) \cup \mathcal{S}(\text{AuxBet1}) \cup \mathcal{S}(\text{AuxBet21}) \\ & = & \{\text{ “}\}\text{” }\} \end{array}$$

Donc l'équation (31) est vérifiée.

$$\mathcal{S}(\text{NamedConstraint}) = \{\text{“,”, “}\}\text{”}\}$$

Donc les équations (41) et (42) sont vérifiées.

$$\begin{array}{lcl} \mathcal{S}(\text{AuxNamed}) & = & \mathcal{S}(\text{BetBraces}) \cup \mathcal{S}(\text{AuxBet1}) \cup \mathcal{S}(\text{AuxBet2}) \\ & & \quad \cup \mathcal{S}(\text{AuxBet3}) \cup \mathcal{S}(\text{AuxBet11}) \cup \mathcal{S}(\text{AuxBet21}) \\ & = & \{\text{ “}\}\text{” }\} \end{array}$$

Donc l'équation (36) est vérifiée.

$$\begin{array}{rcl} \mathcal{S}(\text{NamedValue}) & = & \{\text{“,”}\} \cup \mathcal{S}(\text{AuxNamed}) \\ & = & \{\ \text{“,”, “}\}\text{”}\ \} \end{array}$$

Donc l'équation (37) est vérifiée.

$$\begin{array}{rcl} \mathcal{P}(\text{ObjIdComponent}) & = & \{\ \texttt{number}, \texttt{upper}, \texttt{lower}\ \} \\ \mathcal{S}(\text{ObjIdComponent}) & = & \mathcal{P}(\text{ObjIdComponent}) \cup \{\text{“}\}\text{”}\} \cup \mathcal{S}(\text{AuxBet1}) \cup \mathcal{S}(\text{AuxBet3}) \\ & & \cup\ \mathcal{S}(\text{AuxBet11}) \\ & = & \{\ \texttt{number}, \texttt{upper}, \texttt{lower}, \text{“}\}\text{”}\ \} \end{array}$$

Donc l'équation (4) est vérifiée.

Il reste maintenant à vérifier le système suivant (nous avons remplacé par leur valeur les ensembles $\mathcal{P}$, sauf $\mathcal{P}(\text{NamedValSuf})$) :

$$\begin{array}{ll} (8) & \{\text{“(”, “\{”, SIZE, OF}\} \cap \mathcal{S}(\text{AuxType}) = \emptyset \\ (11) & \mathcal{S}(\text{Type}) \cap \{\text{“.”}\} = \emptyset \\ (15) & \{\text{“(”}\} \cap \mathcal{S}(\text{TypeSuf}) = \emptyset \\ (17) & \{\text{“\{”, DEFINED}\} \cap \mathcal{S}(\text{BuiltInType}) = \emptyset \\ (21) & \{\text{“<”, “:”}\} \cap \mathcal{S}(\text{Value}) = \emptyset \\ (22) & \{\text{“(”, “:”}\} \cap \mathcal{S}(\text{AuxVal0}) = \emptyset \\ (44) & \{\text{“(”, “<”, “..”, “:”}\} \cap \mathcal{S}(\text{SVSAux}) = \emptyset \\ (47) & \{\text{“(”}\} \cap \mathcal{S}(\text{Type}) = \emptyset \end{array}$$

Remarquons que $\left\{ \begin{array}{lcl} \mathcal{S}(\text{AuxType}) & \subseteq & \mathcal{S}(\text{Type}) \\ \mathcal{S}(\text{Type}) & \subseteq & \mathcal{S}(\text{AuxType}) \end{array} \right.$

Donc : $\mathcal{S}(\text{AuxType}) = \mathcal{S}(\text{Type})$

De plus : $\left\{ \begin{array}{lcl} \mathcal{S}(\text{BuiltInType}) & = & \{\text{“(”}\} \cup \mathcal{S}(\text{AuxType}) \\ \mathcal{S}(\text{TypeSuf}) & = & \mathcal{S}(\text{AuxType}) \end{array} \right.$

Nous pouvons alors regrouper les équations (8), (11), (15), (17) et (47) en une seule, et le système est équivalent à :

(X)	$\{$“.”, “(”, “{”, DEFINED, SIZE, OF$\} \cap \mathcal{S}(\text{Type}) = \emptyset$
(21)	$\{$“<”, “:”$\} \cap \mathcal{S}(\text{Value}) = \emptyset$
(22)	$\{$“(”, “:”$\} \cap \mathcal{S}(\text{AuxVal0}) = \emptyset$
(44)	$\{$“(”, “<”, “..”, “:”$\} \cap \mathcal{S}(\text{SVSAux}) = \emptyset$

Nous avons :

$$\begin{array}{lcl}
\mathcal{S}(\text{AuxType}) & = & \mathcal{S}(\text{Type}) \cup \mathcal{S}(\text{NamedType}) \cup \{\text{“:”}\} \\
& = & \mathcal{S}(\text{Type}) \cup \{\text{“,”, “}\}\text{”, OPTIONAL, DEFAULT, “:”}\} \\
\mathcal{S}(\text{Type}) & = & \mathcal{S}(\text{Assignment}) \cup \{\text{“::=”}\} \cup \mathcal{S}(\text{AuxType}) \cup \mathcal{S}(\text{TypeSuf}) \\
& & \quad \cup\ \mathcal{S}(\text{NamedType}) \cup \mathcal{S}(\text{ElementType}) \cup \{\text{“:”}\} \\
& & \quad \cup\ \mathcal{S}(\text{SubtypeValueSet})
\end{array}$$

Et il vient, en remarquant que $\mathcal{S}(\text{ElementType}) \subset \mathcal{S}(\text{NamedType}) \subset \mathcal{S}(\text{AuxType})$:

$$\begin{array}{lcl}
\mathcal{S}(\text{Type}) & = & \{\text{“::=”, “:”}\} \cup \mathcal{S}(\text{Assignment}) \cup \mathcal{S}(\text{SubtypeValueSet}) \cup \mathcal{S}(\text{AuxType}) \\
& = & \{\text{“::=”, “:”}\} \cup \mathcal{S}(\text{Assignment}) \cup \mathcal{S}(\text{SubtypeValueSet}) \cup \mathcal{S}(\text{Type}) \\
& & \quad \cup\ \{\text{“,”, “}\}\text{”, OPTIONAL, DEFAULT, “:”}\} \\
& = & \{\text{“::=”, “:”, “,”, “}\}\text{”, OPTIONAL, DEFAULT}\} \cup \mathcal{S}(\text{Assignment}) \\
& & \quad \cup\ \mathcal{S}(\text{SubtypeValueSet})
\end{array}$$

D'autre part :

$$\begin{array}{lcl}
\mathcal{S}(\text{ModuleBody}) & = & \{\ \text{END}\ \} \\
\mathcal{S}(\text{Assignment}) & = & \mathcal{S}(\text{ModuleBody}) \cup \mathcal{P}(\text{Assignment}) \\
& = & \{\ \text{END, } \texttt{upper}\text{, } \texttt{lower}\ \} \\
\mathcal{S}(\text{SubtypeValueSet}) & = & \{\ \text{“|”, “)”}\ \}
\end{array}$$

Et finalement :

$$\begin{array}{lcl}
\mathcal{S}(\text{Type}) & = & \{\ \text{“::=”, “:”, “,”, “}\}\text{”, “|”, “)”, OPTIONAL, DEFAULT, END,} \\
& & \texttt{upper}\text{, } \texttt{lower}\ \}
\end{array}$$

Donc l'équation (X) est satisfaite.

Remarquons que $\begin{cases} \mathcal{S}(\text{Value}) \subseteq \mathcal{S}(\text{AuxVal0}) \\ \mathcal{S}(\text{AuxVal0}) \subseteq \mathcal{S}(\text{Value}) \end{cases}$

Donc : $\mathcal{S}(\text{AuxVal0}) = \mathcal{S}(\text{Value})$

Nous pouvons subséquemment fusionner les équations (21) et (22), et le système est équivalent à :

$$\begin{array}{ll} \text{(Y)} & \{\text{“<”, “:”, “(”}\} \cap \mathcal{S}(\text{Value}) = \emptyset \\ \text{(44)} & \{\text{“(”, “<”, “..”, “:”}\} \cap \mathcal{S}(\text{SVSAux}) = \emptyset \end{array}$$

De plus :

$$\begin{aligned} \mathcal{S}(\text{SVSAux}) &= \mathcal{S}(\text{SubtypeValueSet}) \cup \mathcal{S}(\text{SVSAux1}) \cup \mathcal{S}(\text{SVSAux2}) \\ &\quad \cup \mathcal{S}(\text{SVSAux3}) \cup \mathcal{S}(\text{SVSAux11}) \cup \mathcal{S}(\text{SVSAux21}) \end{aligned}$$

Or :

$$\begin{aligned} \mathcal{S}(\text{SVSAux11}) &= \mathcal{S}(\text{SVSAux1}) \\ \mathcal{S}(\text{SVSAux1}) &= \mathcal{S}(\text{SVSAux}) \\ \mathcal{S}(\text{SVSAux2}) &= \mathcal{S}(\text{SVSAux}) \\ \mathcal{S}(\text{SVSAux3}) &= \mathcal{S}(\text{SVSAux}) \\ \mathcal{S}(\text{SVSAux21}) &= \mathcal{S}(\text{SVSAux2}) \end{aligned}$$

Donc $\mathcal{S}(\text{SVSAux}) = \{\text{“|”, “)”}\}$ *et par conséquent l'équation (44) est vérifiée.*

Maintenant, il reste à calculer :

$$\begin{aligned} \mathcal{S}(\text{Value}) &= \mathcal{S}(\text{Assignment}) \cup \mathcal{S}(\text{ElementTypeSuf}) \cup \mathcal{S}(\text{AuxVal0}) \cup \mathcal{S}(\text{AuxVal2}) \\ &\quad \cup \mathcal{S}(\text{SpecVal}) \cup \mathcal{S}(\text{NamedValSuf}) \cup \mathcal{S}(\text{UpperEndValue}) \\ &= \{\text{END, upper, lower}\} \cup \mathcal{S}(\text{ElementTypeSuf}) \cup \mathcal{S}(\text{AuxVal2}) \cup \mathcal{S}(\text{SpecVal}) \\ &\quad \cup \mathcal{S}(\text{NamedValSuf}) \cup \mathcal{S}(\text{UpperEndValue}) \end{aligned}$$

Or :

$$\begin{aligned} \mathcal{S}(\text{ElementTypeSuf}) &= \mathcal{S}(\text{ElementType}) \\ \mathcal{S}(\text{NamedValSuf}) &= \mathcal{S}(\text{NamedValue}) \end{aligned}$$

D'où

$$\begin{aligned} \mathcal{S}(\text{Value}) &= \{\text{END, upper, lower, “,”, “}\text{”}\} \cup \mathcal{S}(\text{AuxVal2}) \cup \mathcal{S}(\text{SpecVal}) \\ &\quad \cup \mathcal{S}(\text{UpperEndValue}) \end{aligned}$$

De plus :

$$\begin{aligned} \mathcal{P}(\text{AuxNamed}) &= \{\text{ “,” }\} \\ \mathcal{S}(\text{AuxVal2}) &= \mathcal{S}(\text{Value}) \cup \mathcal{P}(\text{AuxNamed}) \cup \mathcal{S}(\text{AuxBet1}) \\ &\quad \cup \mathcal{S}(\text{AuxBet11}) \cup \mathcal{S}(\text{NamedValSuf}) \\ &= \mathcal{S}(\text{Value}) \cup \{\text{“,”, “}\text{”}\} \end{aligned}$$

Il vient :

$$\mathcal{S}(\text{Value}) = \{\text{END, upper, lower, ",", "}"\} \cup \mathcal{S}(\text{SpecVal}) \cup \mathcal{S}(\text{UpperEndValue})$$

D'autre part :

$$\begin{aligned}\mathcal{S}(\text{SpecVal}) &= \mathcal{S}(\text{AuxVal0}) \cup \mathcal{S}(\text{AuxVal1}) \cup \mathcal{S}(\text{AuxVal11}) \cup \mathcal{P}(\text{AuxNamed}) \\ &\quad \cup \mathcal{S}(\text{AuxBet2}) \cup \mathcal{S}(\text{AuxBet21}) \\ &= \mathcal{S}(\text{Value}) \cup \mathcal{S}(\text{AuxVal1}) \cup \mathcal{S}(\text{AuxVal11}) \cup \{\text{",", "}"\}\end{aligned}$$

Or

$$\begin{aligned}\mathcal{S}(\text{AuxVal11}) &= \mathcal{S}(\text{AuxVal1}) \\ \mathcal{S}(\text{AuxVal1}) &= \mathcal{S}(\text{Value}) \cup \mathcal{S}(\text{NamedValue}) \\ &= \mathcal{S}(\text{Value}) \cup \{\text{",", "}"\}\end{aligned}$$

D'où

$$\mathcal{S}(\text{SpecVal}) = \mathcal{S}(\text{Value}) \cup \{\text{",", "}"\}$$

Par conséquent :

$$\mathcal{S}(\text{Value}) = \{\text{END, upper, lower, ",", "}"\} \cup \mathcal{S}(\text{UpperEndValue})$$

Et enfin :

$$\begin{aligned}\mathcal{S}(\text{SubValSetSuf}) &= \mathcal{S}(\text{SubtypeValueSet}) \cup \mathcal{S}(\text{SVSAux}) \cup \mathcal{S}(\text{SVSAux3}) \\ &\quad \cup \mathcal{S}(\text{SVSAux11}) \\ &= \{\text{ "|", ")" }\} \\ \mathcal{S}(\text{UpperEndValue}) &= \mathcal{S}(\text{SubValSetSuf}) \cup \mathcal{S}(\text{SVSAux2}) \cup \mathcal{S}(\text{SVSAux21}) \\ &= \mathcal{S}(\text{SubValSetSuf}) \cup \{\text{"|", ")"}\} \\ &= \{\text{ "|", ")" }\}\end{aligned}$$

D'où

$$\mathcal{S}(\text{Value}) = \{\text{ END, upper, lower, ",", "}", "|", ")" }\}$$

Donc l'équation (Y) est vérifiée.

Conclusion Le système d'équations est entièrement vérifié, c'est-à-dire que la nouvelle grammaire ASN.1 est LL(1).

5 Réalisation d'un analyseur syntaxique en Caml Light

Nous allons décrire dans cette section une méthode générique pour réaliser des analyseurs syntaxiques de grammaires LL(1) en Caml Light, *mais en aucun cas elle ne se veut universelle*. Elle s'appuie sur la mise au format présenté à la section 1.2, et à quelques contraintes supplémentaires imposées par la sémantique du filtrage des motifs de flux. (cf. Introduction) Nous montrerons comment produire des messages d'erreurs (sans utilisation du contexte) uniquement pour chaque règle susceptible de faire échouer l'analyse, et ce, de façon systématique. Nous verrons de plus comment l'application partielle et la pleine fonctionnalité permettent de construire des analyseurs d'ordre supérieur. Comme exemple d'application, le code source commenté de l'analyseur ASN.1 est présenté à la section 8.

5.1 Contrainte des flux

Dans le cas général, il n'est pas possible de traduire *directement* une grammaire LL(1) en l'analyseur syntaxique correct correspondant.
Les règles qui posent problème sont celles de la forme $\mathrm{A} \rightarrow \mathrm{X\ B} \mid \mathrm{C}$ où $\mathrm{X} \stackrel{*}{\Longrightarrow} \varepsilon$.
Supposons en effet que X reconnaisse ε mais que B échoue ensuite ; dans ce cas l'exception `Parse_failure` déclenchée par B est transformée au niveau de A en `Parse_error` *parce que B n'est pas appelée en tête de motif de flux*, ce qui interrompt l'analyse alors que C aurait pu réussir. Il est clair qu'il n'y avait pas de danger à poursuivre l'analyse étant donné qu'aucun lexème n'avait été consommé dans le flux par X. Cette limitation se justifie par une sémantique plus simple des flux, mais oblige à analyser et à remanier (parfois en profondeur) les grammaires LL(1).
La première étape consiste donc à mettre la grammaire LL(1) au format présenté dans cette étude (cf. 1.2) et à effectuer les transformations supplémentaires suivantes, tant que possible :

$\mathrm{X} \rightarrow [\alpha]\ \beta$	*devient*	$\mathrm{X} \rightarrow \alpha\beta \mid \beta$
$\mathrm{X} \rightarrow \alpha^*\beta$	*devient*	$\mathrm{X} \rightarrow \alpha^+\beta \mid \beta$
$\mathrm{X} \rightarrow \{\ \mathrm{A\ a} \ldots\}^*\beta$	*devient*	$\mathrm{X} \rightarrow \{\ \mathrm{A\ a} \ldots\}^+\beta \mid \beta$
$\mathrm{X} \rightarrow \{\ [\mathrm{A}]\ \mathrm{a} \ldots\}\ \beta$	*devient*	$\mathrm{X} \rightarrow (\mathrm{a}\ [\mathrm{A}])^+\beta \mid \mathrm{A}\ (\mathrm{a}\ [\mathrm{A}])^*\beta \mid \beta$

De cette façon nous satisfaisons la contrainte imposée par les flux Caml Light.

Remarque Si le mot vide appartient au langage, nous voyons apparaître la seule production vide explicite de la grammaire (poser $\beta = \varepsilon$).

5.2 Plaidoyer pour les flux

La contrainte des flux précédemment exposée ne doit pas nous faire oublier les très nombreux avantages qu'ils nous apportent par ailleurs. Tout d'abord il est erroné de croire que

ces derniers ne nous permettent d'analyser que des grammaires LL(1)[9].

Considérons d'abord le cas du fameux « sinon en suspens ». Soit la grammaire ambiguë [2] :

S → **if** *BoolExpr* **then** S S' | *OtherInstr*
S' → **else** S | ε

Cette grammaire modélise la construction `si alors sinon`. Elle provoque chez des générateurs d'analyseurs syntaxique comme YACC un conflit décaler/réduire à la vue de la clause `sinon`. Habituellement dans ce cas, on privilégie l'action « décaler » pour associer le `sinon` au dernier `alors` non clos. Avec Caml Light, ceci est réalisé naturellement, bien qu'il n'y ait pas d'automate à pile qui sous-tende le processus d'analyse syntaxique. Il suffit pour cela d'écrire le filtre de S' *avec en premier motif* **else** *S'*. En effet, les motifs dans un filtre Caml Light sont évalués dans l'ordre d'écriture (de gauche à droite et du haut vers le bas) ; ainsi la fonction d'analyse de S' privilégiera (de par la sémantique d'évaluation du filtre) le choix **else** S' par rapport à ε.

Nous allons montrer pour terminer, en suivant l'exemple donné par [7, 2], que l'on peut aussi reconnaître des langages contextuels à l'aide des flux Caml Light et de la pleine fonctionnalité.
Soit le langage $\{wcw \mid w \in (a+b)^*\}$, où a, b et c sont des terminaux. On montre qu'il est contextuel. Pour analyser ce langage, l'idée consiste à reconnaître le préfixe w et ensuite à fabriquer *dynamiquement* une liste de fonctions d'analyse reconnaissant chaque lettre composant w. Puis, après avoir lu c, nous nous servons de cette liste pour reconnaître le suffixe w. Voyons le détail.
Nous avons donc d'abord besoin d'une fonction `wd` (pour « *word definition* ») qui reconnaisse w et fabrique la liste précitée. En Caml Light, les caractères (de type `char`) a et b sont notés `'a'` et `'b'`.

```
#let rec wd = function
    [< ''a'; wd w >] -> (function [< ''a' >] -> "a")::w
  | [< ''b'; wd w >] -> (function [< ''b' >] -> "b")::w
  | [< >] -> [ ]
;;
wd : char stream -> (char stream -> string) list = <fun>
```

La deuxième fonction d'analyse prend la liste d'analyseurs générés par `wd` et les applique au flux courant. Elle se nomme `wu` (pour « *word usage* »).

```
#let rec wu = function
    p::pl -> (function [< p x; (wu pl) w >] -> x^w)
```

[9] Modulo la contrainte donnée à la section précédente

```
  | [ ] → (function [⟨ ⟩] → "")
;;
wu : (α stream → string) list → α stream → string = ⟨fun⟩
```

Finalement, un analyseur pour le langage ci-dessus est :

```
#let wcw = function [⟨ wd pl; ''c'; (wu pl) w ⟩] → w;;
wcw : char stream → string = ⟨fun⟩
```

En application :

```
#wcw (stream_of_string "abaacabaa");;
- : string = "abaa"
```

5.3 La gestion des erreurs

Pour la réalisation d'un analyseur syntaxique, une préoccupation importante est la détection, le plus tôt possible, des erreurs (propriété dite « du plus long préfixe valide »). Il faudrait de plus que les messages d'erreurs soient les plus informatifs possible. Ce dernier point sous-entend que plus on a de contexte lors de la détection de l'erreur, plus le message pourra être précis et pertinent. Dans le cas d'une réalisation en Caml Light, cela signifie que nous devrions ajouter des paramètres aux fonctions d'analyse, dédiés spécifiquement à cette fin. Nous préférerons ici, par souci de simplicité, ne pas tenir compte du contexte. D'autre part, il faudra prendre garde à n'émettre qu'un seul message d'erreur ; c'est-à-dire que sur le chemin dans l'arbre de dérivation à partir du non-terminal en erreur jusqu'à la racine (axiome), aucun autre message ne devra être produit. Remarquons qu'une reprise sur erreur, même en mode panique[10] n'est pas simple *a priori*. Habituellement c'est le point-virgule (séparateur ou terminateur) qui sert à la reprise de l'analyse, mais l'absence de telle marque en ASN.1 ne permet rien de semblable. Nous pourrions imaginer de se re-synchroniser sur le symbole « ::= », mais dans le cas d'une déclaration de valeur l'identificateur de valeur peut se trouver arbitrairement loin derrière « ::= »... Il n'y a donc pas de solution simple, satisfaisante et générale dans le cas de l'analyse d'un module. Si une erreur se produit pendant l'analyse d'une valeur structurée, nous pouvons nous re-synchroniser sur l'accolade fermante qui suit ; sinon, s'il y a plusieurs modules dans le même fichier source, nous allons jusqu'au END du module courant où s'est produite l'erreur. Bien qu'elle ne présente aucune difficulté, cette stratégie n'a pas été réalisée.
Du point de vue de l'implémentation, nous disposerons d'un module `errors.ml` qui exportera

[10] On ignore les lexèmes jusqu'à tomber sur un qui appartient à un ensemble fixé à l'avance.

une fonction `syntax_error` de type `string` $\rightarrow$ α `stream` $\rightarrow$ β, où le premier argument est un message d'erreur et le second le flux de lexèmes courant contenant en tête le lexème non reconnu. Elle suspend l'exécution en déclenchant l'exception `Parsing_error`.

Nous proposons maintenant une méthode générale d'affichage des messages d'erreurs. Fondamentalement, il n'y a pas de différence de traitement entre les erreurs lexicales et syntaxiques : ce qui importe c'est qu'une erreur s'est produite et que l'on veut le signifier à l'usager le plus précisément possible . Modulo une convention peu contraignante à l'analyse lexicale, les fonctions ici proposées sont *indépendantes* des compilateurs dans lesquelles nous voudrions les insérer. Nous pourrions de même les utiliser pour des erreurs sémantiques, par exemple.

5.3.1 Choix pour l'analyseur lexical

Le principe de base est l'adjonction à chaque lexème reconnu de sa *localisation* dans le texte source, c'est-à-dire de sa position en nombre de caractères depuis le début, ainsi que de sa syntaxe concrète, c'est-à-dire de la chaîne de caractères identifiant le lexème dans le texte source. En fait, pour l'affichage des erreurs, la syntaxe concrète n'est pas nécessaire car seule importe la taille en nombre de caractères du fragment du texte source en cause (pour un lexème ce sera donc le nombre de caractères le composant). Les conventions à l'analyse lexicale sont les suivantes.

1. La localisation du premier caractère du texte source est 1.
2. Nous utiliserons un lexème fictif (dit aussi *virtuel*) qui fera office de sentinelle dans le flux produit par l'analyseur lexical, c'est-à-dire qu'il y aura toujours à la fin du flux produit ce lexème spécial dont l'unique argument sera une localisation. Celle-ci sera 1 si le flux de lexèmes *réels* est vide, et égale à la taille du texte source *plus un* sinon.
3. Nous conserverons toujours une copie originelle du flux de caractères où s'est produite l'erreur.

5.3.2 Le format d'affichage des messages d'erreur

Le format d'affichage sera identique à celui du compilateur Caml Light :

- Le nom du fichier source.
- La ligne où se situe l'erreur.
- Le premier caractère en erreur, compté à partir du début de la ligne.
- La ligne en cause, avec l'erreur soulignée.
- Le message d'erreur spécifique.

- Le premier caractère en erreur, compté à partir du début du fichier.

Par exemple, soit `err.asn1` le fichier source contenant la spécification ASN.1 suivante :

```
ERR {} DEFINITIONS ::=
BEGIN
END
```

Alors nous obtiendrons le résultat :

```
ASN.1 '90 parser
File "err.asn1", line 1, char 6
> ERR {} DEFINITIONS ::=
>      ^
> Object identifier component expected at char 6
```

5.3.3 Le module `errors.ml`

Tout d'abord voici trois fonctions auxiliaires mineures.

- La fonction `tabulation` compte le nombre de tabulations que contient la chaîne qui lui est passée en argument (elle est utilisée pour souligner correctement l'erreur lors de l'affichage de la ligne en cause).
- La fonction `out_string` envoie sur la sortie standard la chaîne qui lui est donnée en argument et vide ensuite ce périphérique, assurant, dans le cas par défaut où il s'agit du terminal, que la chaîne est effectivement affichée.
- La fonction `get_til_eol` prend comme unique argument un flux de caractères et renvoie la chaîne constituée en concaténant les caractères lus dans ce flux jusqu'à un *End Of Line* (exclu), ou sinon jusqu'à épuisement du flux.

```
let tabulations s =
  let tab = ref 0 in
  begin
    for n = 0 to string_length s - 1
    do
      if nth_char s n = '\t' then tab := !tab + 1
    done;
    !tab
  end
;;
```

```
let out_string s = print_string s; flush std_out
;;

let rec get_til_eol = function
  [⟨ '‘\n‘ ⟩] → ‘‘‘‘
| [⟨ 'c; get_til_eol t ⟩] → (make_string 1 c) ^ t
| [⟨ ⟩] → ‘‘‘‘
;;
```

Donnons maintenant une fonction auxiliaire `find_error` dont le rôle est, à partir de la localisation absolue (c'est-à-dire comptée par rapport au début du texte source) et du flux originel de caractères, de retourner un triplet constitué du numéro de la ligne erronée, de la localisation relative (c'est-à-dire comptée à partir du début de la ligne), et d'une chaîne représentant cette même ligne.

```
let find_error ofs strm =
  aux_err 1 1 ‘‘‘‘ ofs strm
  where rec aux_err l_num l_ofs line ofs strm =
    match strm with
      [⟨ '‘\n‘ ⟩] → if ofs = 1
                    then (l_num, l_ofs, line)
                    else aux_err (l_num+1) 1 ‘‘‘‘ (ofs−1) strm
    | [⟨ 'c ⟩] → if ofs = 1
                 then (l_num, l_ofs,
                       line ^ (make_string 1 c) ^ (get_til_eol strm))
                 else aux_err l_num (l_ofs+1) (line ^ (make_string 1 c))
                       (ofs−1) strm
    | [⟨ ⟩] → (l_num, l_ofs, line)
;;
```

Pour finir, voici la fonction principale `print_error` qui est la seule exportée par le module `errors.ml`, et qui, de ce fait, est la seule utilisable dans les compilateurs. Ses arguments dénotent :

- Un en-tête `header` qui qualifie la partie du compilateur où s'est produite l'erreur (par exemple : « ASN.1 '90 lexer »).
- Le flux originel de caractères (`strm`).
- Le nom du fichier source analysé (`filename`).
- Le message d'erreur (`msg`).
- La localisation absolue du premier caractère de la zone erronée (`ofs`).
- La longueur de la zone en erreur (`len`).

```
let print_error header strm filename msg ofs len =
  begin
    out_string ("\n" ^ header);
    (*The interaction between Caml Light and the operating system can generate a location 0
      (e.g. when the ASN.1 source file contains a unique token which is not ended by EOF).
      To prevent this, we use the following 'trick' constant :
     *)
    let trick = if ofs = 0 then 1 else ofs in
    let (l_num, l_ofs, line) = find_error trick strm in
    let s = create_string (l_ofs-1+len) in
    out_string ("\nFile \"" ^ filename ^ "\""
                ^ ", line " ^ (string_of_int l_num)
                ^ ", char " ^ (string_of_int l_ofs)
                ^ "\n");
    out_string ("> " ^ line ^ "\n");
    fill_string s 0 (l_ofs-1) ' ';
    fill_string s 0 (tabulations line) '\t';
    fill_string s (l_ofs-1) len '^';
    out_string ("> " ^ s ^ "\n");
    out_string ("> " ^ msg ^ " at char ");
    out_string ((string_of_int trick) ^ "\n")
  end
;;
```

5.4 Méthode d'analyse

Envisager une méthode d'analyse doit aussi dépendre du choix du langage d'implémentation de l'analyseur. Caml Light est un langage fonctionnel fortement et statiquement typé, ce qui implique notamment que les fonctions peuvent prendre en argument d'autres fonctions et renvoyer comme résultat une fonction. Dans la terminologie des grammaires attribuées, les valeurs passées aux fonctions d'analyse syntaxique sont appelées *attributs hérités*, et leur résultat *attributs synthétisés*. En termes opérationnels, l'arbre des appels de ces fonctions sera nommé *arbre de dérivation*.

Comme il est dit dans l'introduction, Caml Light permet une analyse *descendante*, c'est-à-dire que l'arbre de dérivation est construit des nœuds vers les feuilles. Nous appellerons *sémantique* le résultat renvoyé par une fonction d'analyse, et *arbre de syntaxe abstraite* la valeur retournée par la fonction d'analyse de plus haut niveau (c'est-à-dire l'attribut synthétisé de l'axiome de la grammaire). La sémantique d'une règle pourra être un sous-arbre de syntaxe abstraite ou une *fonction* dont l'application ultérieure produira un sous-arbre de syntaxe abstraite.

Dans le cadre de cette étude, nous nous sommes volontairement restreint à une *analyse purement synthétique*, c'est-à-dire que l'information, pour évaluer la sémantique, circule dans l'arbre de dérivation des feuilles vers les nœuds — autrement dit, il n'y a pas d'attributs hérités. Cela est réalisable grâce à des attributs synthétisés fonctionnels. En effet, supposons qu'à un nœud de l'arbre de dérivation nous ayons un attribut hérité, nous le supprimons alors et nous abstrayons[11] l'attribut synthétisé de ce même nœud par rapport à cet attribut hérité. Ainsi, le calcul final sera effectué au niveau du père du nœud, où se trouve toute l'information nécessaire.

Cette méthode présente l'avantage, dans le cadre d'une éventuelle production automatique de l'analyseur, de bien séparer la syntaxe de la sémantique. En effet, si la première passe est la production d'un analyseur qui répond oui ou non selon que le code source est syntaxiquement correct, la seconde passe consiste alors à *compléter* cet analyseur, plutôt qu'à en *réécrire* la partie de déclaration des arguments. Nous séparons de plus visuellement la nature des attributs : hérités pour l'usage à la reconnaissance de la syntaxe et la gestion des messages erreurs, synthétisés pour la sémantique. On pourrait justement objecter que si le nombre d'attributs hérités au départ est grand, alors nous finissons par perdre en lisibilité. L'exemple d'application de cette méthode à ASN.1 montre que c'est tout de même une approche valable. La seule concession à la fonctionnalité étant deux références globales, représentant le nom du module ASN.1 en cours d'analyse et le mode d'étiquetage par défaut des types[12], car ces informations peuvent servir à tout moment lors de l'analyse et il aurait été pénible d'abstraire *tous* les attributs synthétisés par rapport à ces valeurs.

La sémantique d'évaluation des filtres de flux Caml Light impose d'autre part que l'évaluation des attributs se fasse *de gauche à droite*. Ainsi, si nous avons la règle :
$Z \rightarrow X_0\ X_1\ \ldots\ X_n$, chaque X_i ayant pour attribut synthétisé s_i, alors s_i peut être fonction des $s_{j \leq i}$, mais pas des $s_{j>i}$.

5.5 Forme générale des fonctions d'analyse

5.5.1 Structuration du code

Nous devons d'abord définir un type Caml Light à deux constructeurs constants, et dont les valeurs passées aux fonctions d'analyse serviront à indiquer si, en cas d'échec, une fonction doit interrompre l'analyse — en émettant un message d'erreur — ou non.

```
type Parsing_mode = Abort | Fail;;
```

Pour permettre d'éventuelles applications partielles de ces fonctions d'analyse, nous placerons l'argument de type `Parsing_mode` en première position. Par conséquent la forme générique des fonctions d'analyse est :

[11] *Abstraire* une expression e par rapport à une variable x consiste à former la fonction `fun x → e`.
[12] *tagging*

```
let my_parser mode = function
  [⟨ ... ⟩] → ...
| ...
| [⟨ ... ⟩] → ...
| [⟨ strm ⟩] → match mode with
                 Fail → raise Parse_failure
               | Abort → syntax_error "My message" strm
;;
```

Si nous savons que la fonction ne fera jamais échouer l'analyse (on verra dans 5.7 comment en décider), il suffira de mettre pour l'instant un message vide.

Nous comprenons maintenant mieux un des intérêts de n'avoir aucune règle produisant explicitement ε : nous utilisons le motif de flux vide pour la gestion des échecs d'analyse.

5.5.2 Règle de nommage

Un petit problème a été jusqu'ici passé sous silence à propos de la convention de nommage des fonctions d'analyse. *A priori* on prend pour identificateurs de celles-ci les noms des règles de grammaires associées, mais il faut prendre soin de ne pas produire des identificateurs Caml Light non valides. Par exemple, la règle '*Value*' produirait un mot-clef Caml Light. Donc, dans chacun de ces cas, nous devons imaginer un traitement spécifique de ces noms. Ici, nous avons choisi de préfixer l'identificateur généré directement par le caractère `x`.

5.6 Codage des opérateurs rationnels

Les opérateurs rationnels correspondent à des fonctions d'analyse syntaxique d'ordre supérieur : elles prennent comme premier argument les fonctions d'analyses nécessaires, puis le mode d'échec, et enfin un flux de lexèmes. De cette façon, nous pouvons évaluer partiellement un opérateur en son premier argument, et nous obtenons une fonction d'analyse qui peut à son tour servir à évaluer un autre opérateur, etc. Autrement dit nous pouvons *combiner* arbitrairement les opérateurs.

5.6.1 $\mathbf{X} \rightarrow \alpha^*$

La définition de cet opérateur était : $\mathrm{X} \rightarrow \alpha\ \mathrm{X} \mid \varepsilon$. Sa sémantique sera la liste des sémantiques des α lus :

```
let rec star my_parser mode = function
  [⟨ (my_parser Fail) sem; (star mode my_parser) lst ⟩] → sem::lst
| [⟨ ⟩] → [ ]
;;
```

5.6.2 $\mathbf{X} \rightarrow \alpha^+$

La définition de cet opérateur était : $\mathrm{X} \rightarrow \alpha\alpha^*$. Sa sémantique sera la liste des sémantiques des α lus :

```
let plus my_parser mode = function
  [⟨ (my_parser mode) sem; (star mode my_parser) lst ⟩] → sem::lst
;;
```

5.6.3 $\mathbf{X} \rightarrow [\alpha]$

Rappelons la définition de cet opérateur : $\mathrm{X} \rightarrow \alpha \mid \varepsilon$. Une première approche cohérente avec les autres opérateurs est de retourner une liste (vide si ε a été lu).

```
let option my_parser mode = function
  [⟨ (my_parser Fail) sem ⟩] → [sem]
| [⟨ ⟩] → [ ]
;;
```

5.6.4 $\{$ **A a** ... $\}^*$

La définition de cet opérateur était : $\mathrm{X} \rightarrow \mid \mathrm{A}\ (\mathrm{a}\ \mathrm{A})^*$. Nous codons tout d'abord une fonction auxiliaire, qui sert aussi à l'opérateur $\{\ \mathrm{A}\ \mathrm{a} \ldots\}^*$, et qui correspond à $(\mathrm{a}\ \mathrm{A})^*$:

```
let rec aux1 elm term mode strm =
  let sym = function
    Symbol (_, syn) → syn = term
  | _ → false
  in match strm with
       [⟨ (stream_check sym) _; (elm Abort) e; (aux1 elm term mode) l ⟩] → e::l
     | [⟨ ⟩] → [ ]
;;
```

La fonction `elm` analyse le non-terminal « A », et `term` est celle qui lit le terminal « a ». `Symbol` est un constructeur de lexème dont le premier argument est la localisation de celui-ci dans le texte source (type `int`), et le second sa syntaxe concrète (type `string`).
Il vient alors :

```
let list_star elm term mode = function
  [⟨ (elm Fail) e; (aux1 elm term mode) lst ⟩] → e::lst
| [⟨ ⟩] → [ ]
;;
```

5.6.5 { **A a** ...}$^+$

La définition de cet opérateur était : X → A (a A)*. Nous obtenons alors directement :

```
let list_plus elm term mode = function
  [⟨ (elm mode) e; (aux1 elm term mode) lst ⟩] → e::lst
;;
```

5.6.6 {[**A**] **a** ...}

Rappelons la définition de cet opérateur : X → ε | A (a [A])* | (a [A])$^+$, ce qui peut se réécrire :
X → ε | A (a [A])* | a [A] (a [A])*. Nous procédons d'abord à la définition d'une fonction d'analyse auxiliaire, qui reconnaît (a [A])* :

```
let aux2 elm term mode strm =
  let sym = function
    Symbol (_, syn) → syn = term
  | _ → false
  in match strm with
       [⟨ (stream_check sym) _; (option elm mode) e; (aux2 elm term mode) lst ⟩] → e::lst
     | [⟨ ⟩] → [ ]
;;
```

Par conséquent, il vient :

```
let list_opt elm term mode = function
  [⟨ (elm Fail) e; (aux2 elm term mode) lst ⟩] → (Some e)::lst
| [⟨ 'Symbol (_, term); (option elm mode) e; (aux2 elm term mode) lst ⟩] → e::lst
| [⟨ ⟩] → [ ]
;;
```

5.7 Optimisations

Nous avons fait en sorte que les fonctions d'analyse possèdent jusqu'à présent le même format qui rend nécessaire un paramètre indiquant le comportement de la fonction en cas d'échec (cf. 5.5). Or il est clair que certaines fonctions, en cas d'échec, ne peuvent jamais interrompre l'analyse, et que d'autres le font toujours. Ce sont ces fonctions que nous allons optimiser, en supprimant leur argument de comportement inutile, et nous ferons de même pour les opérateurs rationnels.
L'avantage d'une telle optimisation est double : d'une part nous créons moins de fermetures, et d'autre part nous éliminons du code inutile. L'inconvénient est double lui aussi : d'un côté il faut une analyse supplémentaire de la grammaire, et d'un autre côté nous perdons la possibilité de combinaison arbitraire des opérateurs (pour raison de typage).

5.7.1 $\mathbf{X} \rightarrow \alpha^*$

Nous avions :

```
let rec star my_parser mode = function
  [⟨ (my_parser Fail) sem; (star my_parser mode) lst ⟩] → sem::lst
| [⟨ ⟩] → [ ]
;;
```

Nous constatons que l'argument `mode` est inutile. En effet, nous devons toujours accepter ε, et donc `my_parser` ne doit jamais interrompre l'exécution. Nous supposerons donc par la suite que l'on passe en argument une fonction d'analyse qui n'interrompt jamais le processus, soit qu'elle ne possède pas d'argument de comportement et qu'elle lève `Parse_failure` sur ε, soit qu'elle possède un argument de comportement et qu'elle a été évaluée en celui-ci avec `Fail`. Le code devient :

```
let rec star my_parser = function
  [⟨ my_parser sem; (star my_parser) lst ⟩] → sem::lst
| [⟨ ⟩] → [ ]
;;
```

5.7.2 $\mathbf{X} \rightarrow \alpha^+$

Nous avions :

```
let plus my_parser mode = function
  [⟨ (my_parser mode) sem; (star my_parser mode) lst ⟩] → sem::lst
;;
```

Ici, nous ne pouvons supprimer l'argument de comportement au niveau de l'opérateur : c'est `my_parser` qui en détermine l'usage. L'optimisation de `star` implique cependant la modification de ses appels :

```
let plus my_parser mode = function
  [⟨ (my_parser mode) sem; (star (my_parser Fail)) lst ⟩] → sem::lst
;;
```

5.7.3 $\mathbf{X} \rightarrow [\alpha]$

Nous avions :

```
let option my_parser mode = function
  [⟨ (my_parser Fail) sem ⟩] → [sem]
| [⟨ ⟩] → [ ]
;;
```

Nous comprenons ici que nous pouvons supprimer l'argument de comportement car il n'est pas utilisé dans le corps de la fonction (et ε doit toujours pouvoir être lu). Nous effectuons donc une modification semblable à celle de `star`, sachant qu'il faudra modifier tous les appels à `option` en conséquence. D'autre part, puisque nous avons pris le parti d'optimiser (et donc de perdre la forme commune des types des opérateurs), il est préférable de modifier le type de la valeur retournée par `option`. En effet, nous comprenons bien que la sémantique de cet opérateur est particulière : il faudrait exprimer dans un cas « la sémantique de α » et dans un autre cas « pas de sémantique ». Pour ce faire, nous allons définir un type polymorphe qui permet de construire ces valeurs optionnelles :

```
type α Option = Some of α
              | None
;;
```

Puis, il vient :

```
let option my_parser = function
  [⟨ my_parser sem ⟩] → Some sem
| [⟨ ⟩] → None
;;
```

5.7.4 { **A a** ...}*

Nous avions donné :

```
let rec aux1 elm term mode strm =
  let sym = function
    Symbol (_, syn) → syn = term
  | _ → false
  in match strm with
       [⟨ (stream_check sym) _; (elm Abort) e; (aux1 elm term mode) l ⟩] → e::l
     | [⟨ ⟩] → [ ]
;;
```

Il est clair que l'argument de comportement est inutile car il ne sert par ailleurs qu'à l'appel récursif (et nous devons toujours accepter ε). Donc :

```
let rec aux1 elm term strm =
  let sym = function
    Symbol (_, syn) → syn = term
  | _ → false
  in match strm with
       [⟨ (stream_check sym) _; (elm Abort) e; (aux1 elm term) lst ⟩] → e::lst
     | [⟨ ⟩] → [ ]
;;
```

De plus, nous avions :

```
let list_star elm term mode = function
  [⟨ (elm Fail) e; (aux1 elm term mode) lst ⟩] → e::lst
| [⟨ ⟩] → [ ]
;;
```

Donc :

```
let list_star elm term = function
  [⟨ (elm Fail) e; (aux1 elm term) lst ⟩] → e::lst
| [⟨ ⟩] → [ ]
;;
```

5.7.5 { **A a** ...}$^+$

Nous avions :

```
let list_plus elm term mode = function
  [⟨ (elm mode) e; (aux1 elm term) lst ⟩] → e::lst
;;
```

Ici nous ne pouvons pas supprimer l'argument de comportement.

5.7.6 {[**A**] **a** ...}

Nous avions :

```
let aux2 elm term mode strm =
  let sym = function
    Symbol (_, syn) → syn = term
  | _ → false
  in match strm with
       [⟨ (stream_check sym) _; (option elm mode) e; (aux2 elm term mode) lst ⟩] → e::lst
     | [⟨ ⟩] → [ ]
;;
```

Nous devons supprimer l'argument `mode` de l'appel à `option`. L'argument de comportement peut donc être supprimé au niveau de l'opérateur car il ne sert qu'à l'appel récursif (et ε est accepté). Par conséquent, il vient :

```
let rec aux2 elm term strm =
  let sym = function
    Symbol (_, syn) → syn = term
  | _ → false
  in match strm with
       [⟨ (stream_check sym) _; (option elm) e; (aux2 elm term) lst ⟩] → e::lst
     | [⟨ ⟩] → [ ]
;;
```

D'autre part, nous donnions :

```
let list_opt elm term mode = function
  [⟨ (elm Fail) e; (aux2 elm term mode) lst ⟩] → (Some e)::lst
| [⟨ 'Symbol (_, term); (option elm mode) e; (aux2 elm term mode) lst ⟩] → e::lst
| [⟨ ⟩] → [ ] ;;
```

Nous devons éliminer l'argument `mode` de l'appel à `option`. Nous pouvons donc supprimer l'argument de comportement car il ne sert alors qu'à l'appel récursif (et ε est reconnu) :

```
let list_opt elm term = function
  [⟨ elm e; (aux2 elm term) lst ⟩] → (Some e)::lst
| [⟨ 'Symbol (_, term); (option elm) e; (aux2 elm term) lst ⟩] → e::lst
| [⟨ ⟩] → [ ]
;;
```

5.7.7 Analyse des non-terminaux

Supposons que nous travaillons ici avec la grammaire non implémentatoire d'ASN.1 (cf. 3.5), bien que la méthode d'optimisation reste valable avec la grammaire d'implémentation (cf. 7.1). Nous classons en trois catégories les fonctions d'analyse, selon leur comportement en cas d'échec de filtrage de tous les motifs en tête de flux.

Passantes Ce sont les fonctions d'analyse qui n'interrompent jamais l'exécution.

Bloquantes Ce sont les fonctions d'analyse qui interrompent toujours l'exécution.

Mixtes Ce sont les fonctions d'analyse qui peuvent interrompre ou non l'exécution, selon le contexte d'appel.

Si l'axiome n'est jamais appelé dans la grammaire, alors sa fonction d'analyse est considérée comme étant passante ou bloquante, selon que le mot vide appartient ou non au langage.

En premier lieu, nous parcourons la grammaire en ignorant les expressions impliquant les opérateurs rationnels vus en (1.3). Si une fonction est toujours appelée en tête de motif de flux, alors elle est passante ; si elle est toujours appelée après la tête d'un motif de flux, alors elle est bloquante ; et si elle est appelée en tête *et* après la tête d'un motif de flux, elle est mixte.
Les fonctions des non-terminaux qui n'apparaissent que *dans des expressions rationnelles sont considérées comme étant passantes pour la suite.*

Puis nous examinons les expressions rationnelles :

1. α^*
   Il faut distinguer le premier élément de α. Si c'est un non-terminal dont la fonction d'analyse associée était bloquante, alors elle devient mixte. Pour chaque non-terminal suivant, si leur fonction d'analyse était passante, elle devient mixte.

2. α^+
   Il faut mettre de côté le premier élément de α. Pour chaque non-terminal suivant, si leur fonction d'analyse était passante, elle devient mixte. Il faut distinguer maintenant selon la position de l'expression rationnelle. Si α^+ est en tête de motif de flux et que la fonction d'analyse du premier élément de α était bloquante, alors elle devient mixte. Si α^+ n'est pas en tête de motif de flux et que la fonction d'analyse du premier élément de α était passante, alors elle devient mixte.

3. $[\alpha]$
   Même procédé que pour α^*.

4. $\{$ **A a** $\ldots\}^*$
   Si la fonction d'analyse de *A* était passante ou bloquante, alors elle devient mixte.

5. $\{$ **A a** $\ldots\}^+$
   Comme A^+.

6. $\{$ [**A**] **a** $\ldots\}$
   Même chose que pour [A].

Pour résumer, et en supposant que dans l'énumération précédente $\alpha = \mathrm{A}$:

	Tête	A *avant*	A *après*
A*		Bloquante	Mixte
A+	Oui / Non	Bloquante / Passante	Mixte
[A]		Bloquante	Mixte
{A a ...}*			Mixte
{A a ...}+	Oui / Non	Bloquante / Passante	Mixte
{[A] a ...}		Bloquante	Mixte

Toutes les fonctions mixtes nécessitent un argument de comportement ; cependant, certaines fonctions passantes ou bloquantes, bien que n'utilisant pas dans leur corps un tel argument, peuvent le nécessiter pour des raisons de typage. Pour cela, il suffit qu'une de ces fonctions soit passée en argument aux opérateurs α^+ ou { A a ... }$^+$, car ils nécessitent un argument de comportement pour pouvoir s'appliquer de façon uniforme. Donc la dernière étape de la méthode consiste à relever quelles sont ces fonctions, et à leur imposer un argument de comportement. C'est ce phénomène qui restreint en partie la portée de l'optimisation, mais l'application à ASN.1 montre que seule une fonction possède un tel argument inutilisé, sur un total d'une soixantaine.

Les fonctions d'analyse bloquantes sont donc de la forme :

```
let my_parser mode^opt = function
  [⟨ ... ⟩] → ...
| ...
| [⟨ ... ⟩] → ...
| [⟨ strm ⟩] → syntax_error "My message" strm
;;
```

Notons que l'argument facultatif de comportement *mode* est mis en italique avec un exposant *opt*.

Les fonctions d'analyse passantes ont la forme :

```
let my_parser mode^opt = function
  [⟨ ... ⟩] → ...
| ...
| [⟨ ... ⟩] → ...
;;
```

Idem pour la notation de l'argument facultatif. Remarquons qu'il n'y a pas dans ce cas de motif vide : `Parse_failure` est déclenchée automatiquement en cas d'échec de toutes les têtes de motif.

Les fonctions d'analyse mixtes conservent bien entendu la forme donnée en (5.5).

5.7.8 Analyse des lexèmes

Nous analysons la grammaire et nous formons l'ensemble des terminaux qui ne sont en début d'aucune règle. Pour chacun de ceux-ci nous définissons une fonction d'analyse *bloquante*. Ainsi, lors de l'écriture de l'analyseur syntaxique, nous prendrons soin de lire explicitement les lexèmes en tête de flux, et les autres terminaux à l'aide de leur fonction dédiée. Ainsi, nous supprimons la possibilité d'une levée de `Parse_error` lors d'un échec sur un terminal dans le corps d'un motif de flux.

6 Analyse lexicale d'ASN.1

6.1 Une grammaire pour le lexique d'ASN.1

À partir de la norme nous pouvons extraire une grammaire du lexique (non LL(1)). Pour plus de détails cf. le code de l'analyseur lexical en annexe.

Lower	→	"a" | "b" | ... | "z"
Upper	→	"A" | "B" | ... | "Z"
Letter	→	Lower | Upper
Digit	→	"0" | "1" | ... | "9"
Alpha	→	Letter | Digit
ExtAlpha	→	Alpha | extrasym
HexaBin	→	"H" | "B"

Lexer	→	Tokens*
Tokens	→	Blank* Start
Blank	→	"␣" | "\t" | "\n"
Start	→	stdsym
	|	Digit$^+$
	|	"-" [AuxMinus]
	|	"." [AuxDot]
	|	":" [AuxColon]
	|	"\"" AuxString
	|	"'" Alpha* "'" HexaBin
	|	Lower AuxRef
	|	Upper AuxRef

AuxMinus	→	"-" [Comment]
AuxDot	→	"." ["."]
AuxColon	→	":" [Four]
AuxString	→	ExtAlpha* "\"" ["\"" AuxString]
AuxRef	→	Alpha* "-" (Alpha$^+$ "-")* AuxMinus

Comment	→	"\n" | '-' [AuxCom] | ExtAlpha [Comment]
AuxCom	→	"-" | Comment
Four	→	"=" | AuxColon

6.2 Ambiguïtés lexicales

D'après le document ISO, plusieurs terminaux *sémantiquement* différents sont *lexicalement* indistinguables. Seul le contexte où ils apparaissent permet de les distinguer. Ainsi, un identificateur de type `typereference` est identique à un identificateur de module `modulereference`. De même, un identificateur de valeur `valuereference` est identique à un identificateur de champ `identifier` dans un type SEQUENCE. Dans la grammaire de la syntaxe d'ASN.1, ils seront dénotés respectivement par les identificateurs `upper` et `lower`. Lorsqu'il n'y a pas d'ambiguïté, nous ferons figurer en indice la véritable nature du terminal :

`typereference`, `modulereference`	$\rightsquigarrow$	$\texttt{upper}_{typ}$, $\texttt{upper}_{mod}$, `upper`
`valuereference`, `identifier`	$\rightsquigarrow$	$\texttt{lower}_{val}$, $\texttt{lower}_{id}$, `lower`

Les terminaux `bstring` et `hstring` ont la même sémantique (qui est celle de dénoter un nombre en base binaire ou hexadécimale), et seront donc fusionnés sous le nom `basednum`.

De plus, `cstring` qui dénote une chaîne de caractères à été rebaptisé `string`.

7 Analyse syntaxique d'ASN.1

Nous allons mettre maintenant en œuvre tout ce qui a été dit à la section précédente pour la réalisation d'un analyseur syntaxique pour ASN.1. Pour la partie concernant spécifiquement la sémantique (ou plus exactement l'arbre de syntaxe abstraite et sa construction) cf. (8).

7.1 Une grammaire d'implémentation

Nous effectuons donc les transformations précédentes ainsi que la suivante qui consiste, pour uniformiser le codage des opérateurs rationnels spéciaux (cf. 1.3), en :

$X \to \alpha\ [\beta]\ \gamma \mid \ldots$	*devient*	$X \to \alpha\ [B]\ \gamma \mid \ldots$ $B \to \beta$
$X \to \alpha\ \beta^{+}\ \gamma \mid \ldots$	*devient*	$X \to \alpha\ B^{+}\ \gamma \mid \ldots$ $B \to \beta$
$X \to \alpha\ \beta^{*}\ \gamma \mid \ldots$	*devient*	$X \to \alpha\ B^{*}\ \gamma \mid \ldots$ $B \to \beta$

sauf si β est en fait déjà un non-terminal.

Il est clair que la grammaire résultante restera LL(1). Voici le résultat sur la grammaire ASN.1 :

MODULES

ModuleDefinition	→	ModuleIdentifier DEFINITIONS [Tagging] "::=" BEGIN [ModuleBody] END

ModuleIdentifier	→	`upper`$_{mod}$ [ObjIdCompLst]
ObjIdCompLst	→	"{" ObjIdComponent$^+$ "}"
ObjIdComponent	→	`number`
	|	`upper`$_{mod}$ "." `lower`$_{val}$
	|	`lower` [ClassAttr]
ClassAttr	→	"(" ClassNumber ")"

Tagging	→	TagDefault TAGS
TagDefault	→	EXPLICIT
	|	IMPLICIT

ModuleBody	→	Exports [Imports] Assignment$^+$
	|	Imports Assignment$^+$
	|	Assignment$^+$
Exports	→	EXPORTS {Symbol "," ...}* ";"
Imports	→	IMPORTS SymbolsFromModule* ";"
SymbolsFromModule	→	{Symbol "," ...}$^+$ FROM ModuleIdentifier
Symbol	→	`upper`$_{typ}$
	|	`lower`$_{val}$

Assignment	→	`upper`$_{typ}$ "::=" Type
	|	`lower`$_{val}$ Type "::=" Value

TYPES

Type	→	`lower`$_{id}$ "<" Type
	|	`upper` [AccessType] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType
AccessType	→	"." `upper`$_{typ}$
AuxType	→	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	BuiltInType SubtypeSpec*
	|	SetSeq [TypeSuf]
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	SubtypeSpec$^+$
	|	"{" {ElementType "," ...}* "}" SubtypeSpec*
	|	SIZE SubtypeSpec OF Type
	|	OF Type

BuiltInType	→	BOOLEAN
	|	INTEGER [NamedNumLst]
	|	BIT STRING [NamedBitLst]
	|	OCTET STRING
	|	CHOICE "{" {NamedType "," ...}$^+$ "}"
	|	ANY [AnySuf]
	|	OBJECT IDENTIFIER
	|	ENUMERATED NamedNumLst
	|	REAL
	|	"NumericString"
	|	"PrintableString"
	|	"TeletexString"
	|	"T61String"
	|	"VideotexString"
	|	"VisibleString"
	|	"ISO646String"
	|	"IA5String"
	|	"GraphicString"
	|	"GeneralString"
	|	EXTERNAL
	|	"UTCTime"
	|	"GeneralizedTime"
	|	"ObjectDescriptor"
NamedNumLst	→	"{" {NamedNumber "," ...}$^+$ "}"
NamedBitLst	→	"{" {NamedBit "," ...}$^+$ "}"
AnySuf	→	DEFINED BY `lower`$_{id}$

NamedType	→	`lower`$_{id}$ ["<"] Type
	|	`upper` [AccessType] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType

NamedNumber	→	`lower`$_{id}$ "(" AuxNamedNum ")"
AuxNamedNum	→	`number`
	|	"-" `number`
	|	`lower`$_{val}$
	|	`upper`$_{mod}$ "." `lower`$_{val}$

NamedBit	→	`lower`$_{id}$ "(" ClassNumber ")"

ElementType	→	NamedType [ElementTypeSuf]
	|	COMPONENTS OF Type
ElementTypeSuf	→	OPTIONAL
	|	DEFAULT Value

Class	→	UNIVERSAL
	|	APPLICATION
	|	PRIVATE
ClassNumber	→	`number`
	|	`lower`$_{val}$
	|	`upper`$_{mod}$ "." `lower`$_{val}$

VALEURS

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	`number`
	|	"-" `number`
AuxVal0	→	BuiltInValue
	|	AuxType ":" Value
	|	NULL [SpecVal]
AuxVal1	→	SpecVal
	|	"." AuxVal11
AuxVal2	→	":" Value
	|	"<" Type ":" Value
AuxVal11	→	`upper`$_{typ}$ SpecVal
	|	`lower`$_{val}$
SpecVal	→	SubtypeSpec^{+} ":" Value
	|	":" Value

BuiltInValue	→	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" [BetBraces] "}"

BetBraces	→	AuxVal0 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	`lower` [AuxBet1]
	|	`upper` AuxBet2
	|	`number` [AuxBet3]
AuxBet1	→	"(" ClassNumber ")" ObjIdComponent*
	|	AuxNamed
	|	AuxVal2 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	AuxVal0 [AuxNamed]
	|	`lower` [AuxBet11]
	|	`number` [AuxBet3]
	|	`upper` AuxBet2
AuxBet2	→	SpecVal [AuxNamed]
	|	"." AuxBet21
AuxBet3	→	ObjIdComponent$^+$
	|	AuxNamed
AuxBet11	→	"(" ClassNumber ")" ObjIdComponent*
	|	ObjIdComponent$^+$
	|	AuxVal2 [AuxNamed]
	|	AuxNamed
AuxBet21	→	`upper`$_{typ}$ SpecVal [AuxNamed]
	|	`lower`$_{val}$ [AuxBet3]
AuxNamed	→	"," {NamedValue "," ...}$^+$
NamedValue	→	`lower` [NamedValSuf]
	|	`upper` AuxVal1
	|	`number`
	|	"-" `number`
	|	AuxVal0
NamedValSuf	→	Value
	|	AuxVal2

SOUS-TYPES

SubtypeSpec	→	"(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet	→	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
	|	SVSAux

SubValSetSuf	→	".." ["<"] UpperEndValue
	|	"<" ".." ["<"] UpperEndValue
UpperEndValue	→	Value
	|	MAX

InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	"{" ["..." ","] {[NamedConstraint] "," ...} "}"
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	|	SubtypeSpec [PresenceConstraint]
	|	PresenceConstraint
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL

SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType ":" SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	`number` [SubValSetSuf]
	|	"-" `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec$^+$ ":" SVSAux
	|	":" SVSAux
	|	"." SVSAux11
SVSAux2	→	":" SVSAux
	|	".." ["<"] UpperEndValue
	|	"<" SVSAux21
SVSAux3	→	SubtypeSpec$^+$ ":" SVSAux
	|	":" SVSAux
	|	SubValSetSuf
SVSAux11	→	`upper`$_{typ}$ SubtypeSpec* ":" SVSAux
	|	`lower`$_{val}$ [SubValSetSuf]
SVSAux21	→	Type ":" SVSAux
	|	".." ["<"] UpperEndValue

7.2 Optimisation de l'analyseur

Nous appliquons ici la méthode d'optimisation précédente pour les fonctions d'analyse à la grammaire d'implémentation d'ASN.1. Notons que l'on a rajouté une règle supplémentaires (Specification → ModuleDefinition$^+$) qui fait office de « super-axiome », pour autoriser l'analyse de plusieurs modules ASN.1 dans le même code source. Nous indiquons pour chaque fonction d'analyse si elle nécessite l'argument de comportement *mode* (même si elle ne l'utilise pas). Nous obtenons :

Fonction d'analyse	Statut	Mode	Message d'erreur
Specification	Bloquante	Non	Module definition expected
ModuleDefinition	Passante	Oui	
ModuleIdentifier	Mixte	Oui	Module reference expected
ObjIdCompLst	Passante	Non	
ObjIdComponent	Mixte	Oui	Object identifier component expected
ClassAttr	Passante	Non	
Tagging	Passante	Non	
TagDefault	Passante	Non	
ModuleBody	Passante	Non	
Exports	Passante	Non	
Imports	Passante	Non	
SymbolsFromModule	Passante	Non	
Symbol	Mixte	Oui	Type reference or value reference expected
Assignment	Mixte	Oui	Type definition or value definition expected
Type	Mixte	Oui	Type expected
AccessType	Passante	Non	
AuxType	Passante	Non	
SetSeq	Passante	Non	
TypeSuf	Passante	Non	
BuiltInType	Passante	Non	
NamedNumLst	Mixte	Oui	Left braces beginning a named number list expected
NamedBitLst	Passante	Non	
AnySuf	Passante	Non	
NamedType	Mixte	Oui	Named type expected
NamedNumber	Mixte	Oui	Named number expected
AuxNamedNum	Bloquante	Non	Number or external value reference expected
NamedBit	Mixte	Oui	Named bit expected
ElementType	Mixte	Oui	Named type or inclusion clause expected
ElementTypeSuf	Passante	Non	
Class	Passante	Non	
ClassNumber	Bloquante	Non	Unsigned number or external value reference expected
Value	Mixte	Oui	Value expected
AuxVal0	Passante	Non	
AuxVal1	Bloquante	Non	Subtype specifcation or symbol ':' or symbol '.' expected
AuxVal2	Passante	Non	
AuxVal11	Bloquante	Non	Type reference or value reference expected
SpecVal	Mixte	Oui	Subtype specification or symbol ':' expected
BuiltInValue	Passante	Non	

Fonction d'analyse	Statut	Mode	Message d'erreur
BetBraces	Passante	Non	
AuxBet1	Passante	Non	
AuxBet2	Bloquante	Non	Subtype specifcation or symbol ':' or symbol '.' expected
AuxBet3	Passante	Non	
AuxBet11	Passante	Non	
AuxBet21	Bloquante	Non	Type reference or value reference expected
AuxNamed	Passante	Non	
NamedValue	Mixte	Oui	Named value expected
NamedValSuf	Passante	Non	
SubtypeSpec	Mixte	Oui	Left bracket beginning a subtype specification expected
SubtypeValueSet	Mixte	Oui	Subtype value set expected
SubValSetSuf	Mixte	Oui	Symbol '..' or symbol '<' expected
UpperEndValue	Bloquante	Non	Value or MAX clause expected
InnerTypeSuf	Bloquante	Non	Keyword COMPONENT or keyword COMPONENTS expected
MultipleTypeConstraints	Bloquante	Non	Multiple type constraints expected
NamedConstraint	Passante	Non	
PresenceConstraint	Passante	Non	
SVSAux	Mixte	Oui	Value expected
SVSAux1	Bloquante	Non	Subtype specification or symbol ':' or symbol '.' expected
SVSAux2	Passante	Non	
SVSAux3	Passante	Non	
SVSAux11	Bloquante	Non	Type reference or value reference expected
SVSAux21	Bloquante	Non	Type or symbol '..' expected

Nous donnons maintenant ici une partie de l'interface de l'analyseur lexical, pour comprendre le codage des fonctions de reconnaissance des terminaux.

```
type Location == int
and Syntax == string
;;

type Token = Keyword of Location * Syntax
           | Lower of Location * Syntax
           | Upper of Location * Syntax
           | Number of Location * Syntax
           | BasedNum of Location * Syntax
```

```
          | XString of Location * Syntax
          | Symbol of Location * Syntax
          | Sentry of Location
;;
```

Le premier argument des constructeurs correspond à la localisation du premier caractère du lexème dans le texte source ASN.1, et le second à la *syntaxe concrète* de celui-ci, c'est-à-dire la suite des caractères le caractérisant dans le texte source. Les quatre premiers sont évidents. `BaseNum` correspond au `basednum` de la grammaire, et `XString` au `string` (cf. 6.2). `Symbol` regroupe tous les symboles ASN.1, comme ':', '..', '(', '{', etc. `Sentry` est un lexème fictif à usage interne.

Voici donc maintenant le code des fonctions d'analyse des terminaux ASN.1 :

```
let term_kwd syn strm =
  let kwd = function
    Keyword (_, x) → x = syn
  | _ → false
  in match strm with
       [⟨ (stream_check kwd) _ ⟩] → ()
     | [⟨ ⟩] → syntax_error ("Keyword " ^ syn ^ " expected") strm
;;

let term_sym syn strm =
  let sym = function
    Symbol (_, x) → x = syn
  | _ → false
  in match strm with
       [⟨ (stream_check sym) _ ⟩] → ()
     | [⟨ ⟩] → syntax_error ("Symbol " ^ syn ^ " expected") strm
;;

let term_val = function
  [⟨ 'Lower (_, id) ⟩] → id
| [⟨ strm ⟩] → syntax_error "Value reference expected" strm
;;

let term_id = function
  [⟨ 'Lower (_, id) ⟩] → id
| [⟨ strm ⟩] → syntax_error "Value identifier expected" strm
;;

let term_type = function
  [⟨ 'Upper (_, id) ⟩] → id
```

```
| [⟨ strm ⟩] → syntax_error "Type reference expected" strm
;;

let term_macro = function
  [⟨ 'Upper (_, id) ⟩] → id
| [⟨ strm ⟩] → syntax_error "Macro reference expected" strm
;;

let term_num = function
  [⟨ 'Number (_, n) ⟩] → n
| [⟨ strm ⟩] → syntax_error "Number expected" strm
;;
```

Remarquons l'absence de `string` ou `basednum`, due à leur présence exclusivement en tête de production.

7.3 Une spécification YACC

Il est aisé, à partir de la grammaire d'implémentation, de produire automatiquement une spécification YACC correcte. Cela signifie que, en omettant la possibilité d'une erreur dans YACC, la grammaire proposée ici est aussi LALR(1)[13]. Insistons ici sur un point très important : une spécification YACC n'a d'intérêt que si l'on est capable ensuite de construire un arbre de syntaxe *en C*. Or, la nouvelle grammaire ASN.1 proposée ici est LL(1) et très éloignée de la forme originelle sur laquelle il est assez aisé de calquer la sémantique. C'est pourquoi il a été nécessaire d'employer toutes les ressources d'un langage fonctionnel comme Caml Light pour construire l'arbre de syntaxe abstraite, et notamment les fonctions d'ordre supérieur[14]. De même, c'est cette fonctionnalité qui permet le traitement « à la volée » d'un certain nombre de macros ASN.1, donc de conserver un analyseur syntaxique en une passe. C'est donc cette caractéristique qui fait défaut à C. Cela dit, il est peut-être possible de modifier la spécification YACC, sans créer de conflits (décaler/réduire ou réduire/réduire), pour en obtenir une qui soit adaptée à l'usage de C. Mais il faudra oublier le traitement des macros en une passe... Une autre approche pour le lecteur désirant développer une application en C interfacée avec ce frontal en Caml Light est d'utiliser des compilateurs comme *Bigloo*, qui est un compilateur optimisant de Caml[15] vers C, librement distribué en *ftp* anonyme par l'INRIA. Cf. [6]

[13] Une grammaire LL(1) n'est pas forcément LALR(1).
[14] Les fonctions peuvent prendre en argument une fonction et retourner comme résultat une fonction.
[15] Et Scheme !

8 Sémantique d'ASN.1

Quand on parle de sémantique, il faut bien garder à l'esprit ce que désigne ce mot dans son contexte (Il est très surchargé !). Ici lorsque nous parlons de sémantique d'ASN.1, il faut entendre tout ce qui a trait à la définition d'un arbre de syntaxe abstraite et à son calcul. Si nous avions présenté un outil de vérification du typage des spécifications, nous aurions utilisé le terme « sémantique » pour recouvrir aussi cet aspect. Nous tâcherons donc ici d'évaluer le maximum de la « sémantique » d'ASN.1 à l'analyse syntaxique (c'est-à-dire en levant le maximum d'ambiguïtés), sans pour autant parcourir l'arbre produit pour vérifier, par exemple, si tel type référencé existe, si tel sous-type est non vide, si telle valeur a un type correct, ou si tel module existe — ce qui relève typiquement d'un analyseur sémantique.
Le lecteur doit aussi comprendre qu'il ne s'agit pas ici de paraphraser le document ISO ou d'en donner une version plus claire, mais de *commenter* du code Caml Light, ce qui explique parfois l'apparente incomplétude du discours.

8.1 Un arbre de syntaxe abstraite

La définition d'un arbre de syntaxe abstraite pose plusieurs problèmes.

Premièrement, il est évident qu'il faut *interpréter* le plus « correctement » possible une sémantique en langue naturelle, avec tous les risques biens connus d'ambiguïté et d'incohérence que cela comporte.

Deuxièmement, pour l'implémentation en Caml Light il est nécessaire de poser une convention de nommage très uniforme. Par exemple, les nombres entiers apparaissent dans de nombreux contextes sémantiquement très différents, et il faudra produire un identificateur de constructeur différent pour chaque contexte. Par exemple, `NumCat` pour désigner le numéro de l'étiquette d'un type, et `NumBit` pour qualifier la position d'un bit nommé dans une valeur *BitString*. La convention consiste à abréger la signification locale (« *C'est un nombre, donc* `Num` »), puis à concaténer l'abréviation de son contexte (« *... et qualifiant la position d'un bit nommé, donc* `NumBit` »).

Troisièmement, les définitions de types Caml Light sont parfois liées au calcul même de l'arbre (cf. 8.2), ce qui peut rendre nécessaire la définition de nœuds temporaires, c'est-à-dire qui ne servent qu'à la construction d'autres nœuds, et n'apparaissent jamais dans l'arbre final.

Finalement, la difficulté majeure est que de nombreuses définitions se justifient par les limites de la levée des ambiguïtés sur les types ASN.1 dès l'analyse syntaxique. Ainsi des sous-arbres dénoteront une valeur ASN.1 qui a plusieurs types ASN.1 possibles à ce stade de l'analyse. Du point de vue de l'implémentation en Caml Light, cela sera signifié par des identificateurs de constructeurs (les nœuds) qui seront la concaténation d'abréviations des types ASN.1 possibles pour le sous-arbre dont ils sont la racine. Par exemple, `BitOctStrV`

dénotera une valeur de type *BitString* ou *OctetString*, l'ambiguïté ne pouvant être levée à partir de leur syntaxe concrète seulement.

Nous retrouverons les définitions suivantes de types Caml Light, correspondant à l'arbre de syntaxe abstraite ASN.1, dans le module `ast.mli`.

8.1.1 Définitions élémentaires

Tout d'abord, voyons quelques définitions de base qui correspondent essentiellement aux lexèmes. Nous noterons que nous distinguons maintenant les `valuereference` des `identifier`, ainsi que les `typereference` des `modulereference`. Nous remarquerons d'autre part le type `Option`, vu en (5.6.3). Attention : dans la suite nous utiliserons le mot « identificateur » dans un sens plus large que celui de la norme ISO (`identifier`). Pour éviter les ambiguïtés, nous le flanquerons d'un complément (exemple : « identificateur de type ») ; et sinon il faut le prendre au sens général de « suite de caractères alphanumériques dénotant un type ASN.1, une valeur ASN.1, un module ASN.1 ou un identificateur ASN.1 ».

```
type α Option = Some of α
              | None
;;

type Nat == int;;

type VRef = VRef of string                                   (* valuereference *)
and Ident = Ident of string                                  (* identifier *)
and TRef = TRef of string                                    (* typereference *)
and MRef = MRef of string                                    (* modulereference *)
and SignNum = SignNum of Sign * Nat
and Sign = Plus
         | Minus
;;
```

8.1.2 Les valeurs auxiliaires

Nous présentons maintenant les définitions de nœuds (valeurs Caml Light) temporaires qui servent à la construction d'autres nœuds et qui n'apparaissent jamais dans l'arbre de syntaxe abstraite.

```
type Generic = NumTmp of Nat                                          (* number *)
             | LowNumTmp of string * Nat                              (* (lower, number) *)
             | UpLowTmp of string * string          (* (upper, lower) or (lower, upper) *)
             | LowEVRTmp of string * (MRef * VRef)        (* (lower, (MRef, VRef)) *)
             | UpTmp of string                                        (* upper *)
;;
```

```
type TagMode = Explicit
             | Implicit
;;
```

`Generic` permet le transfert de lexèmes ASN.1 dont la sémantique n'est pas encore trouvée, et qui le sera (peut-être) dans une fonction d'analyse dépendante (cf. 8.2). `TagMode` sert à indiquer comment calculer l'étiquette d'un type.

```
type ClassNum = NumCl of Nat
              | DefValCl of MRef * VRef
;;
```

En observant les transformations de la grammaire ASN.1, nous constatons que certaines règles ont été identifiées (car elles engendraient le même sous-langage), et par conséquent il faut retrouver les différentes sémantiques originelles, dont nous ne nous étions pas préoccupé à ce stade, pour construire l'arbre de syntaxe abstraite. C'est le cas par exemple de la règle '*ClassNumber*' (cf. 3.1.3). La sémantique de cette règle est une valeur Caml Light de type `ClassNum`. Ce peut être un entier (`NumCl`) ou une valeur entière exportée par un autre module (`DefValCl`). Mais c'est le contexte d'appel de '*ClassNumber*' qui décidera de la sémantique exacte (cf. 8.2).

8.1.3 Modules

Voici d'abord les nœuds correspondant à la spécification des modules ASN.1 au sens large, c'est-à-dire sans le détail des types, des sous-types et des valeurs (pour la syntaxe cf. 3.1).

```
type Spec = Spec of ModId * Scope * Def list
and ...
```

La spécification d'*un* module ASN.1 est le triplet constitué d'un identificateur de module (`ModId`) — qui le référencera de façon unique dans l'arborescence ISO —, d'une clause (`Scope`) définissant la portée des identificateurs, et d'une liste de définitions (`Def`) de types et valeurs.

```
and ...
and ModId = ModId of MRef * ObjIdComp list
and Scope = Scope of Import * Export
and Def = TypeDef of TRef * Type
        | ValDef of VRef * Type * Value
and ...
```

Un identificateur de module est la paire constituée d'un nom de module (`MRef`) et d'une liste de nœuds (`ObjIdComp`) dans l'arborescence ISO. La clause de portée (`Scope`) regroupe les identificateurs importés (`Import`) et exportés (`Export`). Les définitions (`Def`) sont de deux sortes : de types (`TypeDef`) ou de valeurs (`ValDef`). Une définition de type est une paire formée par un identificateur de type (`TRef`) et par une définition de type (`Type`) proprement dite. Une définition de valeur est un triplet formé par un identificateur de valeur (`VRef`), un type (`Type`) et par une définition de valeur proprement dite (`Value`).

```
and ...
and ObjIdComp = NumObj of Nat
              | EVRObj of MRef * VRef
              | IdObj of Ident * Form
              | IdVRefObj of string
and Form = NumForm of Nat
         | DefValForm of MRef * VRef
and ...
```

Un nœud dans l'arbre ISO est qualifié

- soit par un nombre (`NumObj`).
- soit par une référence de valeur externe (`EVRObj`), c'est-à-dire une valeur exportée par un autre module.
- soit par `IdObj` : un nom de nœud explicite (`Ident`) précisé par un numéro de sous-arbre directement (`NumForm`) ou indirectement (dans un autre module : `DefValForm`).
- soit par un identificateur de valeur ou une référence de valeur (`IdVRefObj`).

```
and ...
and Import = Import of SymMod list
and Export = Export of Sym list
and Sym = SymVal of VRef
        | SymType of TRef
and SymMod = SymMod of Sym list * ModId
and Sym = SymVal of VRef
        | SymType of TRef
and ...
```

Une clause d'importation (`Import`) est constituée par une liste d'identificateurs exportés par d'autres modules (`SymMod`). Chaque élément de cette liste est une paire constituée de la liste proprement dite d'identificateurs (`Sym`) et de l'identificateur du module qui les exporte. Une clause d'exportation (`Export`) est constituée par une liste d'identificateurs (qualifiant des entités définies au sein du module courant : `Sym`).
Un identificateur (`Sym`) dénote soit une valeur (`SymVal`), soit un type (`SymType`).

8.1.4 Types

```
and ...
and Type = Type of Tag list * Desc * Constraint list
and ...
```

Un type ASN.1 est qualifié complètement par la donnée d'une liste d'étiquettes (`Tag list`), d'une description effective de la structure du type (`Desc`) et par une liste de contraintes de sous-typage (`Constraint`).

```
and ...
and Tag = Tag of Class * Cat * TagMode
        | Undefined
and Class = Universal
          | Application
          | Private
          | Context
and Cat = NumCat of Nat
        | DefValCat of MRef * VRef
and ...
```

Une étiquette (`Tag`) de type ASN.1 peut être *indéterminée* (`Undefined`), pour marquer les types quelconques (*AnyType*), ou bien est un triplet constitué d'une classe normalisée (`Class`), d'une référence dans le catalogue ISO (`Cat`) et du mode d'étiquetage (`TagMode`). Cette référence peut être directe (`NumCat`), sous la forme d'un numéro, ou bien indirecte (`DefValCat`), sous la forme d'une valeur exportée par un autre module.

```
and ...
and Desc = DefType of MRef * TRef
         | BooleanT
         | IntegerT of NamedNum list
         | BitStrT of NamedBit list
         | OctStrT
         | NullT
         | SeqT of ElementType list
         | SeqOfT of Type
         | SetT of ElementType list
         | SetOfT of Type
         | ChoiceT of NamedType list
         | SelectT of Ident * Type
         | AnyT of Ident Option
         | ObjIdT
         | EnumT of NamedNum list
```

```
          | RealT
          | UsefulT of UsefulT
          | CharStrT of CharStr
and ...
```

La description structurelle d'un type ASN.1 repose sur celles des types suivants : type exporté par un autre module (`DefType`), booléen (`BooleanT`), entier (`IntegerT`), chaîne de bits (`BitStrT`), chaîne d'octets (`OctStrT`), non spécifié[16] (`NullT`), ensemble ordonné (dit aussi «séquence») hétérogène de types (`SeqT`), ensemble ordonné homogène de types (`SeqOfT`), ensemble non ordonné hétérogène de types (`SetT`), ensemble non ordonné homogène de types (`SetOfT`), choix (`ChoiceT`), selectionné (`SelectT`), quelconque (`AnyT`), qualifieur de module dans l'arborescence ISO (`ObjIdT`), énuméré (`EnumT`), réel (`RealT`), utilitaires (`UsefulT`), et chaîne de caractères (`CharStrT`).

```
and ...
and NamedNum = NamedNum of Ident * AuxNamedNum
and AuxNamedNum = NumNum of Sign * Nat
                | DefValNum of MRef * VRef
and ...
```

Une liste d'entiers relatifs nommés (`NamedNum`) qualifie les types entiers et énumérés. Le «nommage» se fait par un identificateur de valeur (`Ident`), l'entier proprement dit (`AuxNamedNum`) pouvant quant à lui être littéral (`NumNum`), ou exporté par un autre module (`DefValNum`).

```
and ...
and NamedBit = NamedBit of Ident * AuxNamedBit
and AuxNamedBit = NumBit of Nat
                | DefValBit of MRef * VRef
and ...
```

Une liste de bits nommés (`NamedBit`) qualifie les chaînes de bits. Le «nommage» se fait par un identificateur de valeur (`Ident`), le bit proprement dit (`AuxNamedBit`) pouvant quant à lui être littéral (`NumBit`), ou exporté par un autre module (`DefValBit`).

```
and ...
and ElementType = Mandatory of NamedType
                | Optional of NamedType
                | Default of NamedType * Value
                | Included of Type
and NamedType = NamedType of Ident Option * Type
and ...
```

[16] On parle parfois (incorrectement) de type «vide».

Le type des ensembles ordonnés de types (ou « séquences ») est composé d'une liste de types élémentaires (`ElementType`). Un composant (ou « champ ») peut être de la forme :

- `Mandatory` Cela signifie qu'il faudra explicitement fournir la valeur correspondant à ce champ lors de la définition de la valeur de la séquence.
- `Optional` Dans ce cas, nous pourrons omettre la valeur du champ lors de la définition de la valeur de la séquence (la possible ambiguïté ne peut être détectée à l'analyse syntaxique).
- `Default` C'est comme pour `Optional`, sauf qu'en cas d'omission de la valeur du champ, c'est une valeur par défaut qui sera employée.
- `Included` Le composant alors peut être *logiquement* remplacé par les composants du type indiqué, qui doit donc être une séquence (ce qui ne peut être vérifié sans analyseur sémantique). Il s'agit d'une règle d'inclusion « à plat », c'est-à-dire que les champs inclus sont au même niveau d'imbrication que les autres. La syntaxe ASN.1 pour ce cas est COMPONENTS OF.

Un type nommé (`NamedType`) sert à qualifier les composants ci-dessus (sauf s'ils sont inclus). La norme 1990 d'ASN.1 permet, quand il n'y a pas d'ambiguïté sémantique, d'omettre l'identificateur du type nommé — enfantant par là même une drôle de bête qu'on pourrait appeler « type nommé anonyme ». C'est pourquoi `Ident` est optionnel dans la déclaration Caml Light.

```
and ...
and UsefulT = UTCTime
            | GenTime
            | ObjDesc
            | External
and CharStrT = Numeric
             | Printable
             | Teletex
             | Videotex
             | Visible
             | IA5
             | Graphic
             | General
             | T61
             | ISO646
and ...
```

Sans commentaire.

8.1.5 Sous-types

Fondamentalement un sous-type est un type. Ce qui distingue les sous-types c'est une liste *non vide* de contraintes (`Constraint`). Le lecteur attentif aura d'ailleurs remarqué que dans la sous-section précédente (8.1.4) le type `Constraint` n'a pas été défini... Cela nous permet de mieux structurer le propos et de présenter ici spécifiquement les sous-types.

```
and ...
and Constraint = Constraint of SubValSet list
and ...
```

Une contrainte de sous-typage est une liste d'ensembles de valeurs (`SubValSet`) du type « parent ».

```
and ...
and SubValSet = Single of Value
              | Contained of Type
              | Range of Bound * Bound
              | Alphabet of SubValSet list
              | Size of SubValSet list
              | Inner of Inner
and Bound == Limit * EndVal
and Limit = Strict
          | Large
and EndVal = Min
           | Max
           | EndVal of Value
and ...
```

Un ensemble de valeurs de sous-type peut être :

- `Single` Le sous-type possède alors une seule valeur `Value` du type parent.
- `Contained` Le sous-type inclut les valeurs du type `Type` (qui doit être aussi un sous-type du même type parent — ce qui ne peut être vérifié à l'analyse syntaxique).
- `Range` Le sous-type contient les valeurs entières ou réelles, modulo étiquetage, comprises entre les bornes données (`Bound`). Une borne peut être stricte (`Strict`) ou large (`Large`). Il existe deux pseudo-valeurs spécifiant la valeur minimale de l'intervalle (`Min`) et maximale (`Max`), sinon une valeur explicite est introduite par `EndVal`.
- `Alphabet` Le sous-type ne contient qu'une partie des caractères du type parent (qui doit donc être du type « chaîne de caractères », modulo étiquetage).

- `Size` Le sous-type impose alors une restriction sur la taille des valeurs du type parent (qui doit donc posséder une métrique).
- `Inner` Le sous-type contient les valeurs du type parent structuré satisfaisant des contraintes sur leur présence (éventuellement implicites).

```
and ...
and Inner = SingleConst of SubValSet list
          | MultConst of MultConst
and MultConst = Full of NamedConst Option list
              | Partial of NamedConst Option list
and NamedConst = NamedConst of Ident Option * Constraint Option * Member Option
and Member = Present
           | Absent
           | Option
and ...
```

Une contrainte peut être simple (`SingleConst`) ou multiple (`MultConst`). Dans le premier cas, la contrainte s'applique au sous-type entier (le type parent doit être un ensemble non ordonné homogène) ; dans le second, elle s'applique aux champs et peut être complète (`Full`) ou partielle (`Partial`). Dans les deux cas, nous devons fournir une liste de contraintes nommées (`NamedConst`) optionnelles. Notons que la liste vide et les éléments « logiquement absents » (c'est-à-dire `None`), correspondent à une spécification abrégée qui peut être sémantiquement ambiguë. Une contrainte nommée est le triplet aux champs optionnels composé d'un identificateur de valeur, d'une contrainte et d'une clause de présence (`Member`). En suivant la grammaire (cf.3.5) et l'évaluation de la sémantique (cf. 8.2), nous verrons qu'il n'est pas possible d'avoir tous les champs de ce triplet « logiquement absents » en même temps — c'est juste une commodité de codage que de le permettre théoriquement.

8.1.6 Valeurs

Nous présentons ici les types Caml Light définissant la structure des valeurs ASN.1. Nous verrons qu'il faudra se frotter à de nombreuses ambiguïtés et, dans la mesure où elles peuvent être levées à l'analyse syntaxique (cf. 8.2), donner un arbre de syntaxe abstraite le moins ambigu possible. Un point de vocabulaire : nous appellerons *structurée* une valeur ASN.1 entre accolades.

```
and ...
and Infinity = PlusInf
             | MinusInf
and ...
```

Nous donnons tout de suite, par commodité de présentation, la définition correspondant aux deux lexèmes ASN.1 `PLUS-INFINITY` et `MINUS-INFINITY`. Nous découvrirons un peu plus avant la raison de son existence.

```
and ...
and Value = DefVal of MRef * VRef
          | BooleanV of bool
          | IntegerV of SignNum
          | NullV
          | ChoiceV of Ident * Value
          | AnyV of Type * Value
          | CharStrV of string
          | EmptyV
          | ...
```

Une valeur peut être référencée dans un autre module qui l'exporte (`DefVal`), être un booléen (`BooleanV`), un entier signé (`IntegerV`), non spécifiée (`NullV`), choisie (`ChoiceV`), instanciée (`AnyV`), chaîne de caractères (`CharStrV`), etc. Il existe la valeur « valeur structurée ASN.1 vide » (`EmptyV`), mise pour les ensembles vides de tout type (y compris séquences) et les chaînes de bits vides (cf. *'BuiltInValue'* dans 3.3.2).

```
          | ...
          | IntEnumVRefV of string                    (* lower *)
          | ...
```

En suivant la transformation de grammaire 3.3.3, nous constatons que *IntegerValue*, *EnumeratedValue* et *DefinedValue* ont (syntaxiquement) en commun la production `lower`. Cela nous amène à créer un nœud ambigu regroupant ces différentes possibilités : `IntEnumVRefV`. Notons que le nœud `IntegerV` présenté juste avant exprime la sémantique de la seule construction syntaxique non ambiguë d'un entier ASN.1 : l'entier signé.

```
          | ...
          | RealV of Infinity
          | ...
```

Les transformations présentées en 3.3.1, 3.3.3 et 3.3.4 montrent que les seules productions syntaxiquement non ambiguës de *RealValue* sont celles correspondant aux lexèmes `PLUS-INFINITY` et `MINUS-INFINITY`. C'est ce qui justifie un nœud `RealV`. Si nous voulions être tout à fait pointilleux, nous pourrions objecter que la production "`0`" de *RealValue* est ambiguë, car, transformée en `number` (cf. 3.3.2), elle est ensuite fusionnée avec celle identique de *IntegerValue* dans *Value* (cf. 3.3.3), et sa sémantique devient `IntegerV`. Cet argument est correct : la valeur 0 est la seule valeur entière qui puisse dénoter un nombre réel en ASN.1, mais du fait de son unicité et de la simplicité du test de nullité par un analyseur sémantique, nous acceptons de la considérer ici comme étant toujours entière.

```
  | ...
  | BitOctStrV of string
  | ...
```

L'étape montrée en 3.3.2 montre que *BitStringValue* inclut la règle *OctetStringValue* (`basednum`). De plus, l'étape vue en 3.3.3 montre que la production "{" {$\texttt{lower}_{id}$ "," ...}* "}" est ambiguë et doit être fusionnée dans *BetBraces*. Par conséquent nous définissons un nœud `BitOctStrV` qui regroupe les deux sémantiques possibles d'un lexème `basednum`. La sémantique de l'autre construction ambiguë de *BitStringValue* sera en partie démêlée dans la présentation suivante des nœuds restants du type Caml Light `Value`.

```
  | ...
  | OfV of Value list
  | RealOfV of SignNum * Nat * SignNum
  | BitOfV of Ident list
  | ObjBitOfV of string                                  (* lower *)
  | ObjOfV of Melting
  | GenOfV of NamedVal list
  | ObjGenOfV of Melting
  | ObjIdV of ObjIdComp list

and NamedVal = NamedVal of Ident Option * Value
and Melting = LowNum of string * Nat                        (* lower num *)
            | LowEVR of string * (MRef * VRef)        (* lower upper "." lower *)
            | LowLow of string * string                     (* lower lower *)
            | UpLow of MRef * VRef                          (* upper "." lower *)
            | NumMelt of Nat                                      (* num *)
;;
```

La figure p. 171 regroupe toutes les constructions syntaxiques pouvant apparaître dans une valeur ASN.1 structurée (c'est-à-dire produites par la règle 'BetBraces'). Chaque ensemble est caractérisé par une pseudo-grammaire qui engendre ses éléments. Par exemple, pour l'ensemble étiqueté « ... "," lower Value "," ... », il faut voir que son sur-ensemble est {NamedValue "," ...}$^+$; c'est donc qu'il s'agit de l'ensemble des suites non vides de valeurs nommées *dont au moins une est explicitement nommée.*
Chaque intersection de ces ensembles met par conséquence en évidence une ambiguïté syntaxique. Nous associons à chaque partie un constructeur Caml Light spécifique, identifiant ainsi du mieux que nous le pouvons dès l'analyse syntaxique la sémantique d'une construction. Cette correspondance est graphiquement mise en évidence par des étiquettes donnant l'identificateur du constructeur (selon la convention habituelle) et pointant vers l'intérieur de chaque partie.
Certains éléments des ensembles sont entourés. Dans ce cas, cela signifie que la partie les contenant est finie, et que tous les autres éléments sont aussi entourés.

8.2 Calcul d'un arbre de syntaxe abstraite

Nous présentons ici le calcul de l'arbre de syntaxe abstraite, en Caml Light.

Au cours des transformations de la grammaire ASN.1, nous avons parfois été amené à identifier des règles car elles engendraient le même sous-langage. Par exemple, dans 3.1.3 nous montrons que nous pouvons supprimer la règle 'NumberForm' car elle peut être remplacée partout par 'ClassNumber' (cf. 8.1.2 et 8.2.2).

8.2.1 Définitions auxiliaires

Voici quelques définitions qui servent aux fonctions d'analyse de flux de différentes manières. Remarquons que la plupart des fonctions ne doivent être appelées qu'après avoir vérifié certaines (pré-)conditions sur leurs arguments. Ces vérifications seront parfois implicites, les conditions étant alors remplies par le flot de contrôle, c'est-à-dire par la simple connaissance du motif où est effectué l'appel (voir plus avant).

```
let curMod = ref (MRef "") ;;
let gTags = ref Implicit ;;
```

Ce sont les références globales dont il était question dans (5.4). La première (`curMode`) donne l'identificateur du module ASN.1 en cours d'analyse[17], et la seconde (`gTags`) fournit le mode par défaut d'étiquetage des types dans le module ASN.1 en cours d'analyse.

```
let list_of = function
  Some l → l
| None → [ ]
;;
let fun_of = function
  Some f → f
| None → fun x → [x]
;;
```

Ces deux fonctions ont pour seule utilité de rendre le code plus uniforme. En effet, lorsque nous avons affaire à des arguments de type optionnel (cf. 5.6.3), nous devons pour en exploiter l'information les « ouvrir » dans une construction `match` ; alors que quelquefois le comportement pourrait être uniforme (en imposant une valeur par défaut au cas `None`). C'est dans ce but que ces deux fonctions ont été écrites. La première (`list_of`) est utile quand le contenu de la valeur optionnelle est une liste et que nous pouvons assimiler le cas « liste absente » à celui de la liste vide. La seconde (`fun_of`) sert quand le contenu de la

[17] Un même fichier source peut contenir plusieurs spécifications de modules ASN.1.

valeur optionnelle est une fonction de type $\alpha \rightarrow \alpha\ list$ et que le cas «fonction absente» peut être traité comme le cas `fun x → [x]`.

Nous présentons maintenant les fonctions du module Caml Light `auxiliaries`, qui exporte trois fonctions qui servent à lever des ambiguïtés syntaxiques liées aux valeurs structurées ASN.1 (cf. 8.1.6).

```
let rec is_ValueList = function
  NamedVal (None, _)::l → is_ValueList l
| [ ] → true
| _ → false
;;
```

Cette fonction `is_ValueList` est un prédicat qui prend en argument une liste de valeurs nommées ASN.1, et répond `true` si celle-ci est en fait une liste de valeur nommées *anonymes*, soit : {Value "," ...}*.

```
let rec aux_OfV = function
  NamedVal (None, v)::l → v::(aux_OfV l)
| [ ] → [ ]
| _ → failwith "aux_OfV"
;;
```

Cette fonction `aux_OfV` prend en argument une liste de valeurs nommées *anonymes* et renvoie la liste de valeurs équivalente.

```
let make_OfV l = OfV (aux_OfV l);;
```

Cette fonction `make_OfV` prend en argument une liste de valeurs nommées *anonymes* et renvoie une valeur Caml Light de type `Value`, construite par `OfV` (cf. figure p. 171.).

```
let genOf_Of l = if is_ValueList l
                 then make_OfV l
                 else GenOfV l
;;
```

Cette fonction `genOf_Of` est exportée par le module `auxiliaries.ml`. Elle prend en argument une liste de valeurs nommées ASN.1 *dont nous savons qu'elle peut appartenir soit à la catégorie sémantique* `GenOfV`, *soit* `OfV` (cf. figure p. 171.).

```
let rec is_lowerList = function
  NamedVal (None, IntEnumVRefV _)::l → is_lowerList l
| [ ] → true
| _ → false
;;
```

Cette fonction `is_lowerList` est un prédicat qui prend en argument une liste de valeurs nommées ASN.1, et répond `true` si celle-ci est en fait une liste d'identificateurs de valeur, soit : {`lower` "," ...}*.

```
let rec aux_BitOfV = function
  NamedVal (None, IntEnumVRefV s)::l → (Ident s)::(aux_BitOfV l)
| [ ] → [ ]
| _ → failwith "aux_BitOfV"
;;
```

Cette fonction `aux_BitOfV` prend en argument une liste de valeurs nommées ASN.1 *dont nous savons qu'elle est en fait composée d'identificateurs de valeur*, et renvoie la liste d'identificateurs équivalente.

```
let make_BitOfV l = BitOfV (aux_BitOfV l);;
```

Cette fonction `make_BitOfV` prend en argument une liste de valeurs nommées ASN.1 *dont nous savons qu'elle est en fait composée d'identificateurs de valeur*, et renvoie une valeur Caml Light de type `Value`, construite par `BitOfV` (cf. figure p. 171.).

```
let bitOf_Of l = if is_lowerList l
                 then make_BitOfV l
                 else make_OfV l
;;
```

Cette fonction `bitOf_Of` est exportée par le module `auxiliaries.ml`. Elle prend en argument une liste de valeurs nommées ASN.1 *dont nous savons qu'elle peut appartenir soit à la catégorie sémantique* `BitOfV`, *soit* `OfV` (cf. figure p. 171.).

```
let discr l = match l with
  (NamedVal (None, IntegerV (SignNum _ as man)))
  ::(NamedVal (None, IntegerV (SignNum (Plus, bas))))
  ::(NamedVal (None, IntegerV (SignNum _ as exp)))::[ ]
  → RealOfV (man, bas, exp)
| NamedVal (None, IntEnumVRefV _)::_ → bitOf_Of l
| _ → genOf_Of l
;;
```

Cette fonction `discr` est exportée par le module `auxiliaries.ml`. Elle prend en argument une liste de valeurs nommées ASN.1 *dont nous savons qu'elle appartient au domaine syntaxique* { NamedValue "," ... }$^{+}$ \ ObjIdComponent^{+}[18]. Elle fournit en retour la valeur Caml Light de type `Value` dont le constructeur dénote la catégorie sémantique de l'argument : `BitOfV`, `OfV`, `RealOfV`, ou `GenOfV`. (cf. figure p. 171.)

[18] $A \backslash B$ symbolise le complémentaire de B dans A.

8.2.2 Modules

Nous présentons ici le code correspondant à la spécification de modules ASN.1, abstraction faite des définitions.

```
let rec specification = function
  [⟨ (plus moduleDefinition Fail) modDefns ⟩]
  → modDefns
| [⟨ strm ⟩]
  → syntax_error "Module definition expected" strm
and ...
```

Cette fonction d'analyse n'est pas spécifiée par la grammaire. Elle est une commodité ajoutée pour permettre le traitement de fichiers source ASN.1 incluant plusieurs spécifications de modules. Elle renvoie donc la liste des arbres de syntaxe abstraite correspondant à chacun des modules présents.

```
and ...
and moduleDefinition mode = function
  [⟨ (moduleIdentifier Fail) modId;
     (term_kwd "DEFINITIONS") _;
     (option tagging) tagOpt; (term_sym "::=") _;
     (term_kwd "BEGIN") _;
     (option moduleBody) bodyOpt;
     (term_kwd "END") _
  ⟩]
  → (match tagOpt with
       Some tag → gTags := tag
     | None → ());
     (match bodyOpt with
       Some (scope, defns) → Spec (modId, scope, defns)
     | None → Spec (modId, Scope (Import [ ], Export [ ]), [ ]))
and ...
```

Cette fonction lit l'en-tête du module : positionne le mode d'étiquetage implicite des types, structure la liste des identificateurs importés et exportés, ainsi que les définitions.

```
and ...
and moduleIdentifier mode = function
  [⟨ 'Upper (_, id); (option objIdCompLst) objs ⟩]
  → curMod := MRef id; ModId (MRef id, list_of objs)
| [⟨ strm ⟩]
  → match mode with
```

```
      Fail → raise Parse_failure
    | Abort → syntax_error "Module reference expected" strm
and ...
```

Cette fonction analyse un identificateur de module. Notons l'emploi de la fonction auxiliaire `list_of` qui évite ici un filtrage (`match`) casse-pieds.

```
and ...
and objIdCompLst = function
  [⟨ 'Symbol (_, "{"); (plus objIdComponent Abort) objs; (term_sym "}") _ ⟩]
  → objs
and ...
```

Cette fonction d'analyse ne fait que renvoyer une valeur calculée par une unique sous-fonction d'analyse[19] : une liste de nœuds dans l'arborescence ISO, caractérisant de façon normalisée le module défini.

```
and ...
and objIdComponent mode = function
  [⟨ 'Number (_, n) ⟩]
  → NumObj (int_of_string n)
| [⟨ 'Upper (_, m); (term_sym ".") _; term_val v ⟩]
  → EVRObj (MRef m, VRef v)
| [⟨ 'Lower (_, id); (option classAttr) clOpt ⟩]
  → (match clOpt with
       Some cl → (match cl with
                    NumCl x → IdObj (Ident id, NumForm x)
                  | DefValCl x → IdObj (Ident id, DefValForm x))
     | None → IdVRefObj id)
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Object identifier component expected" strm

and classAttr = function
  [⟨ 'Symbol (_, "("); classNumber cl; (term_sym ")") _ ⟩]
  → cl
and ...
```

La fonction `objIdComponent` analyse un nœud ISO. Notons que `classAttr` est une règle de recopie (de `classNumber`) et que son résultat est déstructuré pour construire une valeur Caml Light dont le type dénote la sémantique correcte dans ce contexte.

[19] On parle alors de «règle de recopie».

```
and ...
and tagging = function
  [⟨ tagDefault td; (term_kwd "TAGS") _ ⟩] → td


and tagDefault = function
  [⟨ 'Keyword (_, "EXPLICIT") ⟩] → Explicit
| [⟨ 'Keyword (_, "IMPLICIT") ⟩] → Implicit
and ...
```

Évident.

```
and ...
and moduleBody = function
  [⟨ exports ex; (option imports) imOpt; (plus assignment Abort) decls ⟩]
  → (Scope (Import (list_of imOpt), Export ex), decls)
| [⟨ imports im; (plus assignment Abort) decls ⟩]
  → (Scope (Import im, Export [ ]), decls)
| [⟨ (plus assignment Fail) decls ⟩]
  → (Scope (Import [ ], Export [ ]), decls)


and exports = function
  [⟨ 'Keyword (_, "EXPORTS"); (list_star symbol ",") syms; (term_sym ";") _ ⟩]
  → syms


and imports = function
  [⟨ 'Keyword (_, "IMPORTS"); (star symbolsFromModule) syms; (term_sym ";") _ ⟩]
  → syms
and ...
```

La fonction `moduleBody` regroupe et structure les informations de portée des identificateurs (importations et exportations) et les déclarations. Les fonctions **`exports`** et **`imports`** renvoient respectivement la liste des exportations et des importations. (Ce sont des règles de recopie.)

```
and ...
and symbolsFromModule = function
  [⟨ (list_plus symbol "," Fail) syms;
     (term_kwd "FROM") _; (moduleIdentifier Abort) modId ⟩]
  → SymMod (syms, modId)
```

```
and symbol mode = function
  [⟨ 'Upper (_, id) ⟩] → SymType (TRef id)
| [⟨ 'Lower (_, id) ⟩] → SymVal (VRef id)
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Type reference or value reference expected" strm
and ...
```

La fonction `symbolsFromModule` associe une liste d'identificateurs au module qui les exporte. La fonction `symbol` lit les identificateurs de type ou de valeur.

```
and ...
and assignment mode = function
  [⟨ 'Upper (_, id); (term_sym "::=") _; (xType Abort) t ⟩]
  → TypeDef (TRef id, t)
| [⟨ 'Lower (_, id); (xType Abort) t; (term_sym "::=") _; (xValue Abort) v ⟩]
  → ValDef (VRef id, t, v)
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Type definition or value definition expected" strm
and ...
```

La fonction `assignment` analyse une définition de type ou de valeur.

8.2.3 Types

Nous présentons maintenant l'analyse des types ASN.1.

```
and ...
and xType mode = function
  [⟨ 'Lower (_, id); (term_sym "<") _; (xType Abort) t ⟩]
  → let (Type (tags, desc, cons)) = t
    in Type ([ ], SelectT (Ident id, Type (tags, desc, [ ])), cons)
| ...
```

Nous analysons les types sélectionnés (*SelectionType*). Le seul détail important ici est que les contraintes de sous-typage (`cons`) portent sur le type sélectionné et non pas sur le type choix (*ChoiceType*) associé (`t`). cf. [3] §36.4 .

```
| ...
| [⟨ 'Upper (_, id); (option accessType) extOpt; (star (subtypeSpec Fail)) cons ⟩]
  → (match extOpt with
       Some ext → Type ([ ], DefType (MRef id, ext), cons)
     | None → Type ([ ], DefType (!curMod, TRef id), cons))
| ...
```

Nous lisons ici une référence de type (interne ou externe au module courant).

```
| ...
| [⟨ 'Keyword (_, "NULL"); (star (subtypeSpec Fail)) cons ⟩]
  → Type ([Tag (Universal, NumCat 5, Explicit)], NullT, cons)
| ...
```

C'est le cas spécial du type non spécifié (*NullType*), dont il est théoriquement possible de dériver des sous-types (!).

```
| ...
| [⟨ auxType t ⟩]
  → t
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Type expected" strm

and accessType = function
  [⟨ 'Symbol (_, "."); term_type id ⟩]
  → TRef id

and auxType = function
  [⟨ 'Symbol (_, "["); (option class) clOpt; classNumber clNb;
     (term_sym "]") _; (option tagDefault) tagOpt; (xType Abort) t ⟩]
  → let cl = (match clOpt with
                Some scope → scope
              | None → Context) in
    let tag = (match tagOpt with
                 Some def → def
               | None → !gTags) in
    let nb = (match clNb with
                NumCl x → NumCat x
              | DefValCl x → DefValCat x) in
    let (Type (tags, desc, cons)) = t
    in Type ((Tag (cl, nb, tag))::tags, desc, cons)
| ...
```

Nous construisons les types étiquetés (*TaggedType*). La classe par défaut est `Context` (cf. [3] §26). Le mode d'étiquetage[20] par défaut est celui du module (`!gTags`). L'étiquette complète reconnue est ensuite insérée en tête de la liste des étiquettes données dans une même déclaration. Notons d'autre part la déstructuration puis la restructuration de la valeur retournée par `classNumber`, pour cause de partage de sémantique (cf. 8.1.2).

```
| ...
| [⟨ builtInType bt; (star (subtypeSpec Fail)) cons ⟩]
  → let (Type (tags, desc, _)) = bt
    in Type (tags, desc, cons)
| ...
```

Remarquons juste que `builtInType` renvoie toujours un type sans contraintes de sous-typage (c'est-à-dire une liste vide), auquel nous lui imposons donc `cons`.

```
| ...
| [⟨ setSeq n; (option typeSuf) sp ⟩]
  → (match sp with
       Some f → f n
     | None → let anyT = Type ([Undefined], AnyT None, [ ]) in
              let desc = (if n = 16 then SeqOfT anyT else SetOfT anyT)
              in Type ([Tag (Universal, NumCat n, Explicit)], desc, [ ]))
and ...
```

Un type spécifié uniquement par le mot-clef **SET** (respectivement **SEQUENCE**) est un ensemble non ordonné homogène (*SetOfType*) de types quelconques (*AnyType*) (respectivement un ensemble ordonné homogène (*SequenceOfType*) de types quelconque).
Si `typeSuf` retourne une valeur fonctionnelle, alors nous lui appliquons le numéro d'étiquette normalisé remonté par `setSeq`.

```
and ...
and setSeq = function
  [⟨ 'Keyword (_, "SET") ⟩] → 17
| [⟨ 'Keyword (_, "SEQUENCE") ⟩] → 16
and ...
```

La fonction renvoie le numéro normalisé de l'étiquette des types ensemble et séquence. Nous profitons du fait que le numéro des ensembles non ordonnés (homogènes ou non) sont identiques (17). Idem pour les séquences (16).

[20] *tagging*

```
and ...
and typeSuf = function
  [⟨ (plus subtypeSpec Fail) cons ⟩]
  → (fun n → let anyT = Type ([Undefined], AnyT None, [ ]) in
             let desc = (if n = 16 then SeqOfT anyT else SetOfT anyT)
             in Type ([Tag (Universal, NumCat n, Explicit)], desc, cons))
| ...
```

Ici nous sommes sûr de retourner un sous-type d'un type ensemble ou séquence de types quelconques. C'est l'appelant (*AuxType*) qui sait s'il s'agit d'un ensemble ou d'une séquence (d'où la nécessité de renvoyer une fonction).

```
| ...
| [⟨ 'Symbol (_, "{"); (list_star elementType ",") elms; (term_sym "}") _;
    (star (subtypeSpec Fail)) cons ⟩]
  → (fun n → let desc = (if n = 16 then SeqT elms else SetT elms)
             in Type ([Tag (Universal, NumCat n, Explicit)], desc, cons))
| ...
```

Par rapport au motif précédent, nous donnons ici explicitement une liste de types nommés — au lieu du type par défaut qu'est le type quelconque —, rendant donc le type construit (séquence ou ensemble) hétérogène.

```
| ...
| [⟨ 'Keyword (_, "OF"); (xType Abort) t ⟩]
  → (fun n → let desc = (if n = 16 then SeqOfT t else SetOfT t)
             in Type ([Tag (Universal, NumCat n, Explicit)], desc, [ ]))
| ...
```

Nous fabriquons une séquence ou un ensemble homogène de type `t`. Remarquons que les contraintes (s'il y en a) portent sur `t` et non sur le type séquence ou ensemble (qui a donc une liste vide associée).

```
| ...
| [⟨ 'Keyword (_, "SIZE"); (subtypeSpec Abort) cons; (term_kwd "OF") _;
    (xType Abort) t ⟩]
  → (fun n → let desc = (if n = 16 then SeqOfT t else SetOfT t)
             in Type ([Tag (Universal, NumCat n, Explicit)], desc, [cons]))
and ...
```

C'est la même chose que pour le motif précédent, sauf que nous introduisons des contraintes de sous-typage `cons` (qui portent sur le type séquence ou ensemble).

```
and ...
and builtInType = function
  [⟨ 'Keyword (_, "BOOLEAN") ⟩]
  → Type ([Tag (Universal, NumCat 1, Explicit)], BooleanT, [])
| [⟨ 'Keyword (_, "INTEGER"); (option (namedNumLst Fail)) nums ⟩]
  → Type ([Tag (Universal, NumCat 2, Explicit)], IntegerT (list_of nums), [])
| [⟨ 'Keyword (_, "BIT"); (term_kwd "STRING") _;(option namedBitLst) bits ⟩]
  → Type ([Tag (Universal, NumCat 3, Explicit)], BitStrT (list_of bits), [])
| [⟨ 'Keyword (_, "OCTET"); (term_kwd "STRING") _ ⟩]
  → Type ([Tag (Universal, NumCat 4, Explicit)], OctStrT, [])
| [⟨ 'Keyword (_, "CHOICE"); (term_sym "{") _; (list_plus namedType "," Abort) types;
     (term_sym "}") _ ⟩]
  → Type ([], ChoiceT types, [])
| [⟨ 'Keyword (_, "ANY"); (option anySuf) dep ⟩]
  → Type ([Undefined], AnyT dep, [])
| [⟨ 'Keyword (_, "OBJECT"); (term_kwd "IDENTIFIER") _ ⟩]
  → Type ([Tag (Universal, NumCat 6, Explicit)], ObjIdT, [])
| [⟨ 'Keyword (_, "ENUMERATED"); (namedNumLst Abort) nums ⟩]
  → Type ([Tag (Universal, NumCat 10, Explicit)], EnumT nums, [])
| [⟨ 'Keyword (_, "REAL") ⟩]
  → Type ([Tag (Universal, NumCat 9, Explicit)], RealT, [])
| [⟨ 'Keyword (_, "NumericString") ⟩]
  → Type ([Tag (Universal, NumCat 18, Explicit)], CharStrT Numeric, [])
| [⟨ 'Keyword (_, "PrintableString") ⟩]
  → Type ([Tag (Universal, NumCat 19, Explicit)], CharStrT Printable, [])
| [⟨ 'Keyword (_, "TeletexString") ⟩]
  → Type ([Tag (Universal, NumCat 20, Explicit)], CharStrT Teletex, [])
| [⟨ 'Keyword (_, "T61String") ⟩]
  → Type ([Tag (Universal, NumCat 20, Explicit)], CharStrT T61, [])
| [⟨ 'Keyword (_, "VideotexString") ⟩]
  → Type ([Tag (Universal, NumCat 21, Explicit)], CharStrT Videotex, [])
| [⟨ 'Keyword (_, "VisibleString") ⟩]
  → Type ([Tag (Universal, NumCat 26, Explicit)], CharStrT Visible, [])
| [⟨ 'Keyword (_, "ISO646String") ⟩]
  → Type ([Tag (Universal, NumCat 26, Explicit)], CharStrT ISO646, [])
| [⟨ 'Keyword (_, "IA5String") ⟩]
  → Type ([Tag (Universal, NumCat 22, Explicit)], CharStrT IA5, [])
| [⟨ 'Keyword (_, "GraphicString") ⟩]
  → Type ([Tag (Universal, NumCat 25, Explicit)], CharStrT Graphic, [])
| [⟨ 'Keyword (_, "GeneralString") ⟩]
  → Type ([Tag (Universal, NumCat 27, Explicit)], CharStrT General, [])
| [⟨ 'Keyword (_, "EXTERNAL") ⟩]
```

```
  → Type ([Tag (Universal, NumCat 8, Explicit)], UsefulT External, [])
| [⟨ 'Keyword (_, "UTCTime") ⟩]
  → Type ([Tag (Universal, NumCat 23, Explicit)], UsefulT UTCTime, [])
| [⟨ 'Keyword (_, "GeneralizedTime") ⟩]
  → Type ([Tag (Universal, NumCat 24, Explicit)], UsefulT GenTime, [])
| [⟨ 'Keyword (_, "ObjectDescriptor") ⟩]
  → Type ([Tag (Universal, NumCat 7, Explicit)], UsefulT ObjDesc, [])
and ...
```

Ce sont les types prédéfinis ASN.1.

```
and ...
and namedNumLst mode = function
  [⟨ 'Symbol (_, "{"); (list_plus namedNumber "," Abort) nums; (term_sym "}") _ ⟩]
  → nums
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Left braces beginning a named number list expected" strm
and ...
```

Nous renvoyons une liste d'entiers nommés (pour des types prédéfinis **INTEGER** ou **ENUMERATED**). C'est une règle de recopie.

```
and ...
and namedBitLst = function
  [⟨ 'Symbol (_, "{"); (list_plus namedBit "," Abort) bits; (term_sym "}") _ ⟩]
  → bits
and ...
```

Nous lisons une liste de bits nommés (pour le type prédéfini **BIT STRING**). C'est une règle de recopie.

```
and ...
and anySuf = function
  [⟨ 'Keyword (_, "DEFINED"); (term_kwd "BY") _; term_val id ⟩]
  → Ident id
and ...
```

Nous renvoyons l'identificateur (qui doit correspondre à celui d'un type nommé. cf. [3] §27.3) pour former un type quelconque (*AnyType*). C'est une simple règle de recopie.

```
and ...
and namedType mode = function
  [⟨ 'Lower (_, id); (option (function [⟨ 'Symbol (_, "<") ⟩] → ())) opt; (xType Abort) t ⟩]
  → (match opt with
       None → NamedType (Some (Ident id), t)
     | Some _ → let (Type (tags, desc, cons)) = t
                in NamedType (None,
                              Type ([ ], SelectT (Ident id, Type (tags, desc, [ ])), cons)))
| ...
```

Dans le cas où nous ne lisons pas le symbole "<", nous avons affaire à un type nommé tout simple. Sinon, c'est un type nommé *anonyme* : un type sélectionné (*SelectionType*) plus précisément.

```
| ...
| [⟨ 'Upper (_, id); (option accessType) extOpt; (star (subtypeSpec Fail)) cons ⟩]
  → (match extOpt with
       Some ext → NamedType (None, Type ([ ], DefType (MRef id, ext), cons))
     | None → NamedType (None, Type ([ ], DefType ( !curMod, TRef id), cons)))
| ...
```

Ici sont analysées des références (éventuellement externes) de types nommés anonymes (éventuellement avec contraintes).

```
| ...
| [⟨ 'Keyword (_, "NULL"); (star (subtypeSpec Fail)) cons ⟩]
  → NamedType (None, Type ([ Tag (Universal, NumCat 5, Explicit) ], NullT, cons))
| ...
```

Nous lisons ici le type non spécifié (*NullType*), éventuellement contraint, et nous l'assimilons à un type nommé anonyme.

```
| ...
| [⟨ auxType t ⟩]
  → NamedType (None, t)
| ...
```

La fonction d'analyse `auxType` renvoie un type dont nous construisons un type nommé anonyme.

```
| ...
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Named type expected" strm
```

```
and namedNumber mode = function
  [⟨ 'Lower (_, id); (term_sym "(") _; auxNamedNum auxn; (term_sym ")") _ ⟩]
  → NamedNum (Ident id, auxn)
| [⟨ strm ⟩]
  → syntax_error "Named number expected" strm
and ...
```

Nous récupérons la valeur numérique de `auxNamedNum` et nous construisons avec l'identificateur de valeur `id` un entier nommé.

```
and ...
and auxNamedNum = function
  [⟨ 'Number (_, n) ⟩]
  → NumNum (Plus, int_of_string n)
| [⟨ 'Symbol (_, "-"); term_num n ⟩]
  → NumNum (Minus, int_of_string n)
| [⟨ 'Lower (_, id) ⟩]
  → DefValNum (!curMod, VRef id)
| [⟨ 'Upper (_, modId); (term_sym ".") _; term_val valId ⟩]
  → DefValNum (MRef modId, VRef valId)
| [⟨ strm ⟩]
  → syntax_error "Number or external value reference expected" strm
and ...
```

Nous analysons les valeurs numériques composant un entier nommé.

```
and ...
and namedBit mode = function
  [⟨ 'Lower (_, id); (term_sym "(") _; classNumber cl; (term_sym ")") _ ⟩]
  → (match cl with
       NumCl n → NamedBit (Ident id, NumBit n)
     | DefValCl dv → NamedBit (Ident id, DefValBit dv))
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Named bit expected" strm
and ...
```

Nous analysons les bits nommés. Remarquons une fois de plus la surcharge sémantique de la règle `classNumber`. (cf. 8.1.2)

```
and ...
and elementType mode = function
  [⟨ (namedType Fail) nt; (option elementTypeSuf) elmOpt ⟩]
  → (match elmOpt with
        Some fMode → fMode nt
      | None → Mandatory nt)
| [⟨ 'Keyword (_, "COMPONENTS"); (term_kwd "OF") _; (xType Abort) t ⟩]
  → Included t
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Named type or inclusion clause expected" strm
and ...
```

Notons simplement dans le premier motif que si le mode de présence du champ (*ElementType*) est absent (cas `None`), alors c'est le mode par défaut qui s'applique, à savoir « obligatoire » (*Mandatory*).

```
and ...
and elementTypeSuf = function
  [⟨ 'Keyword (_, "OPTIONAL") ⟩]
  → (fun nType → Optional nType)
| [⟨ 'Keyword (_, "DEFAULT"); (xValue Abort) val ⟩]
  → (fun nType → Default (nType, val))
and ...
```

Nous retournons une fonction qui prendra en argument un type nommé et qui construira un champ (*ElementType*) avec le mode de présence. L'évaluation de cette fonction retournée se fait dans `elementType`.

```
and ...
and class = function
  [⟨ 'Keyword (_, "UNIVERSAL") ⟩] → Universal
| [⟨ 'Keyword (_, "APPLICATION") ⟩] → Application
| [⟨ 'Keyword (_, "PRIVATE") ⟩] → Private
and ...
```

Lecture du mode d'étiquetage des types ASN.1.

```
and ...
and classNumber = function
  [⟨ 'Number (_, n) ⟩]
```

```
  → NumCl (int_of_string n)
| [⟨ 'Lower (_, id) ⟩]
  → DefValCl ( !curMod, VRef id)
| [⟨ 'Upper (_, id); (term_sym ".") _; term_val v ⟩]
  → DefValCl (MRef id, VRef v)
| [⟨ strm ⟩]
  → syntax_error "Unsigned number or external value reference expected" strm
and ...
```

Nous renvoyons une valeur entière, soit sous forme littérale (`NumCl`), soit sous forme de référence interne ou externe (`DefValCl`). Les valeurs renvoyées seront toujours déstructurées au niveau de l'appelant. cf. (8.1.2).

8.2.4 Valeurs

Voici l'implémentation des fonctions d'analyses relatives aux valeurs ASN.1.

```
and ...
and xValue mode = function
  [⟨ auxVal0 v ⟩]
  → v
| [⟨ 'Upper (_, id); auxVal1 val1 ⟩]
  → val1 id
| [⟨ 'Lower (_, id); (option auxVal2) val2Opt ⟩]
  → (match val2Opt with
       Some val2 → val2 id
     | None → IntEnumVRefV id)
| ...
```

Si la valeur est réduite à un identificateur, alors elle est ambiguë (`IntEnumVRefV`) car elle peut dénoter soit un entier, soit un énuméré, soit une référence de valeur.

```
| ...
| [⟨ 'Number (_, n) ⟩]
  → IntegerV (SignNum (Plus, int_of_string n))
| [⟨ 'Symbol (_, "-"); term_num n ⟩]
  → IntegerV (SignNum (Minus, int_of_string n))
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Value expected" strm
```

```
and auxVal0 = function
  [⟨ builtInValue bv ⟩]
  → bv
| [⟨ auxType t; (term_sym ":") _; (xValue Abort) v ⟩]
  → AnyV (t, v)
| [⟨ 'Keyword (_, "NULL"); (option (specVal Fail)) specOpt ⟩]
  → (match specOpt with
       Some spec → spec NullT
     | None → NullV)
and ...
```

Cette fonction doit renvoyer une valeur Caml Light de type `Value`. Notons, avec un peu d'avance, que `specVal` renvoie une fonction qui prend en argument une description de type ASN.1 et qui renvoie une *AnyValue*. D'autre part, nous constatons que le mot-clef `NULL` peut avoir deux sémantiques différentes (type ou valeur ASN.1) selon la présence ou l'absence de contraintes de sous-typage.

```
and ...
and auxVal1 = function
  [⟨ (specVal Fail) spec ⟩]
  → (fun id → spec (DefType (!curMod, TRef id)))
| [⟨ 'Symbol (_, "."); auxVal11 val11 ⟩]
  → val11
| [⟨ strm ⟩]
  → syntax_error "Subtype specification or symbol ':' or symbol '.'  expected" strm
and ...
```

La fonction $\texttt{auxVal}_1$ est appelée dans `xValue` et `namedValue`. Dans les deux cas, il faut appliquer un identificateur à $\texttt{auxVal}_1$. Étant donné que cet identificateur pourra être tantôt considéré comme un identificateur de module (cf. $\texttt{auxVal}_{11}$), tantôt comme un identificateur de type ($\texttt{auxVal}_1$), il convient de profiter du type Caml Light *string* pour passer l'argument — cela évite de créer des nœuds temporaires supplémentaires (cf. 8.1.2).

```
and ...
and auxVal2 = function
  [⟨ 'Symbol (_, ":"); (xValue Abort) v ⟩]
  → (fun id → ChoiceV (Ident id, v))
| [⟨ 'Symbol (_, "<"); (xType Abort) t; (term_sym ":") _;(xValue Abort) v ⟩]
  → (fun id → AnyV (Type ([ ], SelectT (Ident id, t), [ ]), v))
and ...
```

Cette fonction d'analyse est appelée dans les fonctions `xValue`, $\texttt{auxBet}_1$, $\texttt{auxBet}_{11}$ et `namedValSuf`. Dans tous ces contextes elle est amenée à définir une valeur choisie (*ChoiceValue*) ou instanciée (*AnyValue*). Pour cela, elle aura toujours besoin d'un identificateur en

paramètre (Remarquons que, bien que cet identificateur pourrait être typé `Ident` d'après l'usage qu'on en fait dans $\texttt{auxVal}_2$, nous devons le typer `string` car d'autres contextes d'appel peuvent être ambigus. cf. $\texttt{auxBet}_1$).

```
and ...
and auxVal11 = function
  [⟨ 'Upper (_, id); (specVal Abort) spec ⟩]
  → (fun m → spec (DefType (MRef m, TRef id)))
| [⟨ 'Lower (_, id) ⟩]
  → (fun m → DefVal (MRef m, VRef id))
| [⟨ strm ⟩]
  → syntax_error "Type reference or value reference expected" strm
and ...
```

Cette fonction $\texttt{auxVal}_{11}$ termine l'analyse commencée par $\texttt{auxVal}_1$. Elle doit donc s'évaluer en une fonction qui prend en argument un identificateur (qui ne peut pourtant pas être typé `MRef` à cause du type de la première production de $\texttt{auxVal}_1$). Cette fonction retournée s'évaluera à son tour en une valeur instanciée (*AnyValue*) ou une référence de valeur (`DefVal`).

```
and ...
and specVal mode = function
  [⟨ (plus subtypeSpec Fail) cons; (term_sym ":") _; (xValue Abort) v ⟩]
  → (fun t → AnyV (Type ([ ], t, cons), v))
| [⟨ 'Symbol (_, ":"); (xValue Abort) v ⟩]
  → (fun t → AnyV (Type ([ ], t, [ ]), v))
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Subtype specification or symbol ':' expected" strm
and ...
```

Cette fonction `specVal` est appelée par $\texttt{auxVal}_0$, $\texttt{auxVal}_2$, $\texttt{auxVal}_{21}$, $\texttt{auxBet}_2$ et $\texttt{auxBet}_{21}$. Dans tous ces contextes, le plus simple était de renvoyer une fonction prenant une description de type (et non un type complet) pour fabriquer une valeur instanciée (*AnyValue*).

```
and ...
and builtInValue = function
  [⟨ 'Keyword (_, "TRUE") ⟩]
  → BooleanV true
| [⟨ 'Keyword (_, "FALSE") ⟩]
  → BooleanV false
| [⟨ 'Keyword (_, "PLUS-INFINITY") ⟩]
```

```
  → RealV PlusInf
| [⟨ 'Keyword (_, "MINUS-INFINITY") ⟩]
  → RealV MinusInf
| [⟨ 'BasedNum (_, n) ⟩]
  → BitOctStrV n
| [⟨ 'XString (_, s) ⟩]
  → let str = sub_string s 1 (string_length s − 2) in CharStrV str
| [⟨ 'Symbol (_, "{"); (option betBraces) betOpt; (term_sym "}") _ ⟩]
  → (match betOpt with
       Some bet → bet
     | None → EmptyV)
and ...
```

Dans `builtInValue`, il n'y a rien d'extraordinaire, si ce n'est l'unique usage du nœud `EmptyV`, pour spécifier une valeur structurée ASN.1 vide (dont le type ne peut être déterminé que par un analyseur sémantique). Notez d'autre part que l'on ôte les parenthèses à la syntaxe concrète du lexème `XString`.

```
and ...
and betBraces = function
  [⟨ auxVal0 v; (option auxNamed) nvLstOpt ⟩]
  → (match nvLstOpt with
       Some nvLst → genOf_Of (nvLst (NamedVal (None, v)))
     | None → OfV [v])
| ...
```

Rappelons tout d'abord que `auxNamed` renvoie une fonction qui prend une valeur nommée (*NamedValue*), lit une liste de valeurs nommées, ajoute en tête l'argument et renvoie la nouvelle liste ainsi constituée.
Si la valeur optionnelle `nvLstOpt` se trouve être « vide », alors c'est que nous avons lu une valeur nommée anonyme via $\texttt{auxVal}_0$, c'est-à-dire ici que l'on a forcément affaire à une liste (à un élément) de catégorie `OfV` (cf. figure p. 171.). Sinon, il est possible que la liste de valeurs nommées qui suit contienne au moins une valeur nommée *explicite*, et donc il faut faire appel à la fonction `genOf_Of` (cf. 8.2.1) pour lever l'ambiguïté.

```
| ...
| [⟨ 'Symbol (_, "-"); term_num n; (option auxNamed) nvLstOpt ⟩]
  → let sInt = IntegerV (SignNum (Minus, int_of_string n))
    in (match nvLstOpt with
          None → OfV [sInt]
        | Some nvLst → discr (nvLst (NamedVal (None, sInt))))
| ...
```

Commençons par lire un entier négatif (`sInt`). S'il est suivi d'une liste (non vide) de valeurs nommées, alors nous utilisons la fonction `discr` (cf. 8.2.1) pour trouver la plus petite catégorie sémantique de la liste. Sinon, nous sommes sûr d'avoir une liste (à un élément) de catégorie `OfV`. Cf. figure p. 171.

```
| ...
| [⟨ 'Lower (_, id); (option auxBet1) bet1Opt ⟩]
  → (match bet1Opt with
       Some bet1 → bet1 id
     | None → ObjBitOfV id)
| ...
```

Ici, la première valeur débute par un identificateur de valeur ou de référence de valeur (`lower`). Si elle est seule, alors nous sommes certain de construire une valeur de catégorie sémantique `ObjBitOfV`. (cf. figure p. 171.) Sinon, nous laissons tout le travail à $\texttt{auxBet}_1$.

```
| ...
| [⟨ 'Upper (_, up); auxBet2 bet2 ⟩]
  → bet2 (UpTmp up)
| ...
```

La première valeur commence par un identificateur de type ou de module. Puisque ce doit être une valeur ASN.1, il doit être nécessairement suivi par quelque chose ($\texttt{auxBet}_2$). Notons que $\texttt{auxBet}_2$, apparaissant dans des contextes hétérogènes, a besoin explicitement d'un nœud temporaire (cf. 8.1.2).

```
| ...
| [⟨ 'Number (_, n); (option auxBet3) bet3Opt ⟩]
  → let num = int_of_string n
    in (match bet3Opt with
          Some bet3 → bet3 (NumTmp num)
        | None → ObjOfV (NumMelt num))
and ...
```

Si nous lisons un entier positif seul, c'est que nous sommes en train de former une valeur de catégorie sémantique `ObjOfV`. (cf. figure p. 171.) Sinon, nous comptons sur $\texttt{auxBet}_3$ pour traiter les possibles ambiguïtés. Remarquons, pour les mêmes raisons données ci-dessus pour $\texttt{auxBet}_2$, il faut passer à $\texttt{auxBet}_3$ un nœud temporaire.

```
and ...
and auxBet1 = function
  [⟨ 'Symbol (_, "("); classNumber cl; (term_sym ")") _;
```

```
    (star (objIdComponent Fail)) objs ⟩]
  → (fun id → let h = (match cl with
                         NumCl x → IdObj (Ident id, NumForm x)
                       | DefValICl x → IdObj (Ident id, DefValForm x))
              in ObjIdV (h::objs))
| ...
```

La fonction d'analyse `auxBet`$_1$ n'est appelée que depuis `xValue`, ce qui simplifie sa présentation car il faut juste garder en mémoire qu'elle a été appelée alors qu'un identificateur de type ou de module venait d'être lu (`upper`).
Dans cette première production de `auxBet`$_1$, nous sommes certains de construire une valeur du domaine sémantique `ObjIdV`. (cf. figure p. 171.) Remarquons une fois de plus la déstructuration du nœud temporaire `ClassNum`. (cf. 8.1.2)

```
| ...
| [⟨ auxNamed nvLst ⟩]
  → (fun id → bitOf_Of (nvLst (NamedVal (None, IntEnumVRefV id))))
| ...
```

Ici nous avons analysé une liste de plus d'un élément, dont le premier est un identificateur ; donc il faut décider si elle appartient au domaine `BitOfV` ou `OfV`. C'est le rôle de la fonction `bitOf_Of`. cf. (8.2.1).

```
| ...
| [⟨ auxVal2 val2; (option auxNamed) nvLstOpt ⟩]
  → (match nvLstOpt with
       Some nvLst → (fun id → genOf_Of (nvLst (NamedVal (None, val2 id))))
     | None → (fun id → OfV [val2 id]))
| ...
```

Après la lecture de `auxVal`$_2$ nous savons que nous avons analysé une valeur instanciée (*AnyValue*), donc une valeur nommée anonyme. Si elle n'est pas suivie d'autres valeurs nommées (cas `None`), alors nous sommes sûr d'avoir reconnu une valeur du domaine sémantique `OfV`. (cf. figure p. 171.) Sinon, il faut déterminer si la liste (dénote une valeur qui) appartient à la catégorie `GenOfV` ou `OfV`. C'est l'utilité de la fonction `genOf_Of`. (cf. 8.2.1)

```
| ...
| [⟨ 'Symbol (_, "-"); term_num n; (option auxNamed) nvLstOpt ⟩]
  → (fun id → GenOfV ((fun_of nvLstOpt)
                        (NamedVal (Some (Ident id),
                                   IntegerV (SignNum (Minus, int_of_string n))))))
| ...
```

Il est clair ici que la première valeur nommée de la liste est `lower "-" number`, donc nous construisons nécessairement une valeur du domaine `GenOfV`. (cf. figure p. 171.) Notons l'usage de la fonction `fun_of` qui évite d'écrire un filtre sur une valeur optionnelle qui rendrait moins lisible le code alors que le traitement serait uniforme dans tous les cas. (cf. 8.2.1)

```
| ...
| [⟨ auxVal0 val0; (option auxNamed) nvLstOpt ⟩]
  → (fun id → GenOfV ((fun_of nvLstOpt) (NamedVal (Some (Ident id), val0))))
| ...
```

La fonction d'analyse $\texttt{auxVal}_0$ renvoie une valeur Caml Light de type `Value`, donc la première valeur nommée de la liste est explicite (c'est-à-dire qu'elle est « vraiment » nommée). Nous sommes par conséquent certain de construire une valeur de la catégorie sémantique `GenOfV`. Même remarque que ci-dessus concernant `fun_of`.

```
| ...
| [⟨ 'Lower (_, i1); (option auxBet11) bet11Opt ⟩]
  → (match bet11Opt with
       Some bet11 → bet11 i1
     | None → (fun i0 → ObjGenOfV (LowLow (i0, i1))))
| ...
```

Après la lecture de $\texttt{i}_1$, nous avons reconnu `lower lower`. Si c'est l'unique production de la valeur structurée, alors nous avons une valeur du domaine `OjGenOfV` (cf. figure p. 171.), sinon nous reportons la décision au niveau de $\texttt{auxBet}_{11}$. (Ce qui implique au passage que cette dernière devra rendre une expression abstraite deux fois, sur deux identificateurs, dans le bon ordre.)

```
| ...
| [⟨ 'Number (_, n); (option auxBet3) bet3Opt ⟩]
  → let num = int_of_string n
    in (match bet3Opt with
          Some bet3 → (fun low → bet3 (LowNumTmp (low, num)))
        | None → (fun low → ObjGenOfV (LowNum (low, num))))
| ...
```

Après la lecture de `n` nous savons que nous avons reconnu le préfixe `lower number`. Si c'est l'unique élément de la valeur structurée, alors nous savons que celle-ci appartient à la catégorie `ObjGenOfV`. (cf. figure p. 171.) Sinon nous reportons la levée de l'ambiguïté au niveau de $\texttt{auxBet}_3$. Nous avons déjà vu $\texttt{auxBet}_3$ appelée dans `BetBraces`, c'est-à-dire à un autre niveau dans la hiérarchie des appels, donc dans un autre contexte ; c'est pourquoi nous lui passons un nœud temporaire. (cf. 8.1.2)

```
| ...
| [⟨ 'Upper (_, up); auxBet2 bet2 ⟩]
  → (fun low → bet2 (UpLowTmp (low, up)))
and ...
```

Après la lecture de `up` nous savons que nous avons reconnu `lower upper`. Nous construisons alors un nœud temporaire (cf. 8.1.2) avec ces deux identificateurs et nous le transmettons à $\texttt{auxBet}_2$. Nous nous souviendrons que l'argument de `UpLowTmp` peut être interprété comme (`lower`, `upper`) ou (`upper`, `lower`). Ainsi, dans $\texttt{auxBet}_2$ nous filtrerons en pensant au premier cas, alors que dans $\texttt{auxBet}_3$ ce sera la seconde possibilité. (Cela permet l'économie d'un nœud temporaire.)

```
and ...
and auxBet2 = function
  [⟨ (specVal Fail) spec; (option auxNamed) nvLstOpt ⟩]
  → (function
       UpTmp up → let any = spec (DefType ( !curMod, TRef up))
                  in (match nvLstOpt with
                        Some nvLst → genOf_Of (nvLst (NamedVal (None, any)))
                      | None → OfV [any])
     | ...
```

La fonction d'analyse $\texttt{auxBet}_2$ est appelée dans différents contextes : depuis `betBraces` et $\texttt{auxBet}_1$. La fonction qu'elle renvoie sera appelée avec un nœud temporaire `UpTmp` si et seulement si $\texttt{auxBet}_2$ fut appelée à partir de `betBraces`. À ce stade de l'analyse nous avons reconnu le préfixe `upper` de la première valeur de la valeur structurée ASN.1. Puisque `specVal` réussit ensuite, nous savons que cette première valeur est une valeur instanciée (*AnyValue*) : `any`. Maintenant, si elle n'est suivie d'aucune autre valeur nommée, c'est que nous avons affaire à une valeur structurée du domaine sémantique `OfV`. (cf. figure p. 171.) Sinon, nous l'ajoutons à la tête des valeurs nommées (elle est anonyme), et nous levons l'ambiguïté par `genOf_Of`. (cf. 8.2.1)

```
     | ...
     | UpLowTmp (low, up)
       → let nVal = NamedVal (Some (Ident low), spec (DefType ( !curMod, TRef up)))
         in (match nvLstOpt with
               Some nvLst → genOf_Of (nvLst nVal)
             | None → GenOfV [nVal])
     | ...
```

La fonction renvoyée par $\texttt{auxBet}_2$ est appelée avec un nœud temporaire `UpLowTmp` si et seulement si $\texttt{auxBet}_2$ est appelée à partir de $\texttt{auxBet}_1$. Donc nous savons qu'à ce niveau,

nous avons déjà reconnu le préfixe `lower upper` de la première valeur de la valeur structurée ASN.1. Puisque `specVal` réussit ensuite, nous savons que cette première valeur est une valeur instanciée (*AnyValue*) nommée *explicite* (c'est-à-dire avec un nom l'identifiant explicitement). Si elle n'est suivie par aucune autre valeur nommée, alors c'est qu'elle appartient à la catégorie sémantique `GenOfV`. (cf. figure p. 171.) Sinon nous l'adjoignons à la liste des autres valeurs nommées et `genOf_Of` (cf. 8.2.1) se charge de déterminer si la valeur structurée ASN.1 appartient alors au domaine `GenOfV` ou `OfV`.

```
    | ...
    | _ → failwith "Fatal error in 'auxBet2'.  Please report.")
```

Ce motif «joker» filtrera tous les nœuds qui ne seront pas `UpTmp` ou `UpLowTmp` : *cela ne devrait jamais se produire*. En effet, il n'existe aucun appel à la fonction retournée par $\texttt{auxBet}_2$ avec d'autres nœuds de type (Caml Light) `Generic` (cf. 8.1.2) ; mais ce dernier motif est cependant nécessaire pour la bonne compilation du code (complétude du filtre).

```
| ...
| [⟨ 'Symbol (_, "."); auxBet21 bet21 ⟩]
  → bet21
| ...
```

La fonction d'analyse $\texttt{auxBet}_2$ est toujours appelée après qu'un identificateur de type ou de module ait été lu (`upper`), quelque soit le lieu d'appel (`betBraces` ou $\texttt{auxBet}_1$). C'est dans ce même contexte qu'est appelée $\texttt{auxBet}_{21}$, et donc il ne faudra pas oublier d'abstraire par rapport à un nœud temporaire l'expression qu'elle renvoie. (cf. motif précédent.)

```
| ...
| [⟨ strm ⟩]
  → syntax_error "Subtype specification or symbol ':' or symbol '.' expected" strm
and auxBet3 = function
  [⟨ (plus objIdComponent Fail) objs ⟩]
  → (function
       NumTmp num → ObjIdV ((NumObj num)::objs)
     | ...
```

La fonction d'analyse $\texttt{auxBet}_3$ est appelée à partir de `betBraces`, $\texttt{auxBet}_1$ et $\texttt{auxBet}_{21}$, c'est-à-dire dans des contextes différents, ce qui justifie qu'on passe en argument un nœud temporaire à la fonction qu'elle renvoie. Cette dernière sera appelée avec un nœud temporaire `NumTmp` si et seulement si $\texttt{auxBet}_3$ fut appelée à partir de `betBraces`. Par conséquent, à ce stade, nous sommes en train de construire une valeur structurée ASN.1 de domaine sémantique `ObjIdV` (car nous avons lu une suite *non vide*[21] de nœuds ISO. cf. 8.1.3), dont le premier élément est un entier (`num`).

[21] Si elle était vide, la valeur se trouverait dans la catégorie `ObjOfV`. cf. figure p. 171.

```
| ...
| LowNumTmp (low, num) → ObjIdV ((IdVRefObj low)::(NumObj num)::objs)
| ...
```

La fonction renvoyée par $\texttt{auxBet}_3$ sera appelée avec un nœud temporaire `LowNumTmp` si et seulement si $\texttt{auxBet}_3$ fut appelée à partir de $\texttt{auxBet}_1$. Par conséquent, nous déduisons, comme précédemment, que la valeur structurée ASN.1 doit être de la catégorie sémantique `ObjIdV` (cf. figure p. 171.) car la liste de nœuds ISO qui suit *est non vide*[22], et dont les deux premiers éléments (nœuds ISO) sont `lower` et `number`. Remarquons simplement que le premier nœud ISO est ambigu.

```
| ...
| UpLowTmp (up, low) → ObjIdV ((EVRObj (MRef up, VRef low))::objs)
| ...
```

La fonction renvoyée par $\texttt{auxBet}_3$ sera appelée avec un nœud temporaire `UpLowTmp` si et seulement si $\texttt{auxBet}_3$ fut appelée à partir de $\texttt{auxBet}_{21}$. Par conséquent, nous sommes en train de construire ici une valeur structurée ASN.1 du domaine sémantique `ObjIdV` (cf. figure p. 171.) dont le premier élément (nœud ISO) est une référence de valeur externe.

```
| ...
| LowEVRTmp (low, evr) → ObjIdV ((IdVRefObj low)::(EVRObj evr)::objs)
| ...
```

La fonction renvoyée par $\texttt{auxBet}_3$ sera appelée avec un nœud temporaire `LowEVRTmp` si et seulement si $\texttt{auxBet}_3$ fut appelée à partir de $\texttt{auxBet}_{21}$. Par conséquent, nous sommes en train de construire ici une valeur structurée ASN.1 du domaine sémantique `ObjIdV` (cf. figure p. 171.) dont les premiers éléments (nœuds ISO) sont `lower` (ambigu) et une référence de valeur externe.

```
| ...
| _ → failwith "Fatal error in 'auxBet3'.  Please report." )
```

Ce motif « joker » filtrera tous les nœuds qui ne sont pas `NumTmp`, `LowNumTmp`, `UpLowTmp` ou `LowEVRTmp`. *Cela ne devrait jamais se produire.* En effet, il n'existe aucun appel à la fonction renvoyée par $\texttt{auxBet}_3$ qui passe un nœud `UpTmp`, par exemple ; nous avons besoin de ce motif pour assurer la complétude du filtre et le déclenchement d'exception sera utile si le code est ultérieurement modifié de façon incorrecte.

[22] Sinon nous tomberions dans la catégorie `ObjGenOfV`.

```
| [⟨ auxNamed nvLst ⟩]
  → (function
       NumTmp num → let uInt = IntegerV (SignNum (Plus, num))
                    in discr (nvLst (NamedVal (None, uInt)))
     | ...
```

La fonction renvoyée par `auxBet`$_3$ sera appelée avec un nœud temporaire `NumTmp` si et seulement si `auxBet`$_3$ fut appelée à partir de `betBraces`. De plus, nous lisons une suite de valeurs nommées (`auxNamed`), donc nous savons que la valeur structurée ASN.1 que nous analysons n'est pas dans le domaine sémantique `ObjIdV` (cf. figure p. 171.) ; l'appel à la fonction `discr` permet de lui attribuer le plus petit domaine possible. (cf. 8.2.1) Notons que la première valeur nommée est un entier anonyme.

```
     | ...
     | LowNumTmp (low, num) → let uInt = IntegerV (SignNum (Plus, num))
                              in GenOfV (nvLst (NamedVal (Some (Ident low), uInt)))
     | ...
```

La fonction renvoyée par `auxBet`$_3$ sera appelée avec un nœud temporaire `LowNumTmp` si et seulement si `auxBet`$_3$ fut appelée à partir de `auxBet`$_1$. D'autre part, nous lisons une suite *non vide*[23] de valeurs nommées (`auxNamed`), donc nous déduisons que la valeur structurée ASN.1 que nous analysons est forcément dans le domaine sémantique `GenOfV`. (cf. figure p. 171.) La première valeur nommée est donc un entier nommé explicitement.

```
     | UpLowTmp (up, low) → genOf_Of (nvLst (NamedVal (None,
                                                      DefVal (MRef up, VRef low))))
     | ...
```

La fonction renvoyée par `auxBet`$_3$ sera appelée avec un nœud temporaire `UpLowTmp` si et seulement si `auxBet`$_3$ fut appelée à partir de `auxBet`$_{21}$. De plus, nous lisons une suite de valeurs nommées (`auxNamed`), donc nous savons que la valeur structurée ASN.1 que nous analysons est soit dans le domaine sémantique `GenOfV`, soit `OfV`. (cf. figure p. 171.) L'appel à la fonction *ad hoc* `genOf_Of` permet de lever l'ambiguïté.

```
     | ...
     | LowEVRTmp (low, evr) → GenOfV (nvLst (NamedVal (Some (Ident low), DefVal evr)))
     | ...
```

La fonction renvoyée par `auxBet`$_3$ sera appelée avec un nœud temporaire `LowEVRTmp` si et seulement si `auxBet`$_3$ fut appelée à partir de `auxBet`$_{21}$. De plus, nous lisons une suite de valeurs nommées *non vide*[24], donc nous savons que la valeur structurée ASN.1 que nous analysons est dans le domaine sémantique `GenOfV`.

[23] Si elle avait été vide, nous aurions placé la valeur dans la catégorie `ObjGenOfV`. cf. figure p. 171.
[24] Si elle avait été vide, la valeur aurait été dans `ObjGenOfV`. cf. figure p. 171.

```
  | ...
  | _ → failwith "Fatal error in 'auxBet3'.  Please report.")
```

Comme précédemment, ce motif «joker» filtrera tous les nœuds différents de `NumTmp`, `LowNumTmp`, `UpLowTmp` et `LowEVRTmp`. *Cela ne devrait jamais se produire.* En effet, il n'existe aucun appel à la fonction d'analyse $\texttt{auxBet}_3$ avec le nœud temporaire `UpTmp` ; ce motif sert donc à assurer la complétude du filtre et à se prémunir d'une éventuelle modification incorrecte du code.

```
and ...
and auxBet11 = function
  [⟨ 'Symbol (_, "("); classNumber cl; (term_sym ")") _; (star (objIdComponent Fail)) objs ⟩]
  → (fun i1 i0 → let h = (match cl with
                            NumCl x → IdObj (Ident i1, NumForm x)
                          | DefValCl x → IdObj (Ident i1, DefValForm x))
                 in ObjIdV ((IdVRefObj i0)::h::objs))
| ...
```

La fonction d'analyse $\texttt{auxBet}_{11}$ est appelée uniquement à partir de $\texttt{auxBet}_1$, et cela lorsque nous avons reconnu le préfixe `lower lower` dans la valeur structurée ASN.1 (`betBraces`). C'est ce qui explique que $\texttt{auxBet}_{11}$ retourne une expression abstraite sur deux identificateurs. Le premier ($\texttt{i}_1$) correspond au *second* `lower` (c'est-à-dire au niveau de la règle *'AuxBet1'*), et le second ($\texttt{i}_0$) correspond au *premier* `lower` (c'est-à-dire au niveau de la règle *'BetBraces'*).
Dans ce motif en particulier, nous sommes alors certain de construire une valeur appartenant au domaine sémantique `ObjIdV`. Remarquons une fois de plus la déstructuration du nœud temporaire `ClassNum`. (cf. 8.1.2)

```
| ...
| [⟨ (plus objIdComponent Fail) objs ⟩]
  → (fun i1 i0 → ObjIdV ((IdVRefObj i0)::(IdVRefObj i1)::objs))
| ...
```

Dans ce cas, nous lisons une suite *non vide* de nœuds ISO, donc la valeur structurée ASN.1 associée relève de la catégorie sémantique `ObjIdV`. (Si la suite avait été vide, nous serions alors dans le domaine sémantique `ObjGenOfV`.) Les deux identificateurs lus avant l'appel à $\texttt{auxBet}_{11}$ sont alors deux nœuds ISO (ambigus d'ailleurs).

```
| ...
| [⟨ auxVal2 val2; (option auxNamed) nvLstOpt ⟩]
  → (fun i1 i0 → GenOfV ((fun_of nvLstOpt) (NamedVal (Some (Ident i0), val2 i1))))
| ...
```

Ici, la réussite de la fonction d'analyse `auxVal`$_2$ nous assure que nous sommes en train de lire une liste de valeurs nommées, dont la première est une valeur instanciée (*AnyValue*) explicitement nommée (par `i`$_0$). Il ne fait aucun doute alors que la valeur structurée ASN.1 appartient au domaine sémantique `GenOfV`. (cf. figure p. 171.)

```
| ...
| [⟨ auxNamed nvLst ⟩]
  → (fun i1 i0 → GenOfV (nvLst (NamedVal (Some (Ident i0), IntEnumVRefV i1))))
```

Puisque `auxNamed` réussit, c'est que nous étions en train de lire une liste de valeurs nommées, dont la première est donc `lower`$_{id}$ `lower`. C'est-à-dire que la valeur structurée ASN.1 correspondante est dans le domaine sémantique `GenOfV`. (cf. figure p. 171.)

```
and ...
and auxBet21 = function
  [⟨ 'Upper (_, id); (specVal Abort) spec; (option auxNamed) nvLstOpt ⟩]
  → (function
       UpTmp up → let any = spec (DefType (MRef up, TRef id))
                  in (match nvLstOpt with
                        Some nvLst → genOf_Of (nvLst (NamedVal (None, any)))
                      | None → OfV [any])
     | ...
```

La fonction d'analyse `auxBet`$_{21}$ est uniquement appelée à partir de `auxBet`$_2$, qui renvoie le resultat de `auxBet`$_{21}$ comme son résultat. Or `auxBet`$_2$ peut être appelée dans différents contextes, donc il faut gérer dans `auxBet`$_{21}$ ces différents cas, à l'aide de nœuds temporaires. La fonction renvoyée par `auxBet`$_{21}$ sera appelée avec un nœud temporaire `UpTmp` si et seulement si `auxBet`$_2$ fut appelée à partir de `betBraces`. Dans ce cas, au moment de débuter l'analyse dans `auxBet`$_{21}$, nous avons déjà reconnu le préfixe `upper "."` de la première valeur de la valeur structurée ASN.1. Dans ce premier motif, puisque `specVal` réussit, nous reconnaissons que la première valeur est une valeur instanciée (*AnyValue*). Maintenant, si elle n'est suivie d'aucune autre valeur nommée, c'est que nous avons affaire à une valeur structurée du domaine sémantique `OfV`. (cf. figure p. 171.) Sinon, nous l'ajoutons à la tête des valeurs nommées (elle est anonyme), et nous levons l'ambiguïté par `genOf_Of`. (cf. 8.2.1)

```
     | ...
     | UpLowTmp (low, up)
       → let nVal = NamedVal (Some (Ident low), spec (DefType (MRef up, TRef id)))
         in (match nvLstOpt with
               Some nvLst → genOf_Of (nvLst nVal)
             | None → GenOfV [nVal])
     | ...
```

La fonction retournée par `auxBet`$_{21}$ est appelée avec un nœud temporaire `UpLowTmp` si et seulement si `auxBet`$_2$ est appelée à partir de `auxBet`$_1$. Donc nous savons qu'à ce niveau, nous avons déjà reconnu le préfixe `lower upper` de la première valeur de la valeur structurée ASN.1. Puisque `specVal` réussit ensuite, nous savons que cette première valeur est une valeur instanciée (*AnyValue*) nommée *explicite* (c'est-à-dire avec un nom l'identifiant explicitement). Si elle n'est suivie par aucune autre valeur nommée, alors c'est qu'elle appartient à la catégorie sémantique `GenOfV`. Sinon nous l'adjoignons à la liste des autres valeurs nommées et `genOf_Of` (cf. 8.2.1) se charge de déterminer si la valeur structurée ASN.1 appartient alors au domaine `GenOfV` ou `OfV`.

```
    | ...
    | _ → failwith "Fatal error in 'auxBet21'.  Please report.")
```

Ce motif «joker» filtrera tous les nœuds qui ne seront pas `UpTmp` ou `UpLowTmp` : *cela ne devrait jamais se produire*. En effet, il n'existe aucun appel à la fonction retournée par `auxBet`$_{21}$ avec d'autres nœuds de type (Caml Light) `Generic` (cf. 8.1.2) ; mais ce dernier motif est cependant nécessaire pour la bonne compilation du code (complétude du filtre).

```
| [⟨ 'Lower (_, id); (option auxBet3) bet3Opt ⟩]
  → (function
       UpTmp up → (match bet3Opt with
                     Some bet3 → bet3 (UpLowTmp (up, id))
                   | None → ObjOfV (UpLow (MRef up, VRef id)))
    | ...
```

La fonction renvoyée par `auxBet`$_{21}$ sera appelée avec un nœud temporaire `UpTmp` si et seulement si `auxBet`$_2$ fut appelée à partir de `betBraces`. Dans ce cas, au moment de débuter l'analyse dans `auxBet`$_{21}$, nous avons déjà reconnu le préfixe `upper "."` de la première valeur de la valeur structurée ASN.1. Dans ce second motif, nous savons que nous avons lu la première valeur `upper "." lower`. Si elle n'est suivie d'aucune autre, alors c'est que la valeur structurée appartient au domaine sémantique `ObjOfV`. (cf. figure p. 171.) Sinon, nous construisons un nœud temporaire `UpLowTmp` que nous passons à la fonction d'analyse `auxBet`$_3$. (cf. *supra*)

```
    | ...
    | UpLowTmp (low, up)
      → (match bet3Opt with
           Some bet3 → bet3 (LowEVRTmp (low, (MRef up, VRef id)))
         | None → ObjGenOfV (LowEVR (low, (MRef up, VRef id))))
    | ...
```

La fonction retournée par `auxBet`$_{21}$ est appelée avec un nœud temporaire `UpLowTmp` si et seulement si `auxBet`$_2$ est appelée à partir de `auxBet`$_1$. Donc nous savons qu'à ce niveau, nous

avons déjà reconnu le préfixe `lower upper "."` de la première valeur de la valeur structurée ASN.1. Dans ce motif, nous lisons `lower` ensuite, donc cette première valeur a pour syntaxe concrète : `lower upper "." lower`

```
    | _ → failwith "Fatal error in 'auxBet21'.  Please report.")
```

Ce motif « joker » filtrera tous les nœuds qui ne seront pas `UpTmp` ou `UpLowTmp` : *cela ne devrait jamais se produire*. En effet, il n'existe aucun appel à la fonction retournée par $\texttt{auxBet}_{21}$ avec d'autres nœuds de type (Caml Light) `Generic` (cf. 8.1.2) ; mais ce dernier motif est cependant nécessaire pour la bonne compilation du code (complétude du filtre).

```
| [⟨ strm ⟩]
  → syntax_error "Type reference or value reference expected" strm


and auxNamed = function
  [⟨ 'Symbol (_, ","); (list_plus namedValue "," Abort) nvLst ⟩]
  → (fun nv → nv::nvLst)
and ...
```

La fonction d'analyse `auxNamed` reconnaît une liste de valeurs nommées séparées par des virgules, et introduite par une virgule. Elle retourne une fonction qui attend une valeur nommée (celle qui se trouvait *avant* la première virgule), l'insère en tête et retourne la liste complète.

```
and ...
and namedValue mode = function
  [⟨ 'Lower (_, id); (option namedValSuf) nvsOpt ⟩]
  → (match nvsOpt with
       Some nvs → nvs id
     | None → NamedVal (None, IntEnumVRefV id))
| ...
```

La fonction d'analyse `namedValue` reconnaît une valeur nommée. Dans ce premier motif, nous commençons par `lower` ; si la valeur optionnelle qui suit équivaut à « absence de valeur », alors nous formons une valeur nommée anonyme ambiguë (`IntEnumVRefV`), sinon nous laissons la fonction retournée par `namedValSuf` s'occuper de la formation de la valeur.

```
| ...
| [⟨ 'Upper (_, id); auxVal1 val1 ⟩]
  → NamedVal (None, val1 id)
| [⟨ 'Number (_, n) ⟩]
```

```
  → NamedVal (None, IntegerV (SignNum (Plus, int_of_string n)))
| [⟨ 'Symbol (_, "-"); term_num n ⟩]
  → NamedVal (None, IntegerV (SignNum (Minus, int_of_string n)))
| [⟨ auxVal0 val ⟩]
  → NamedVal (None, val)
| ...
```

Ces derniers motifs correspondent en partie à ceux de `xValue`. Notons que leur évaluation produira des valeurs nommées anonymes.

```
| ...
| [⟨ strm ⟩]
  → syntax_error "Named value expected" strm


and namedValSuf = function
  [⟨ (xValue Fail) val ⟩]
  → (fun id → NamedVal (Some (Ident id), val))
| [⟨ auxVal2 val2 ⟩]
  → (fun id → NamedVal (None, val2 id))
and ...
```

La fonction d'analyse `namedValSuf` est appelée uniquement à partir de `namedValue`, après que celle-ci ait reconnu `lower`. Dans le premier motif, nous construisons une valeur nommée explicite, alors que dans le second, ce sera une valeur anonyme — une valeur choisie (*ChoiceValue*) ou instanciée (*AnyValue*).

8.2.5 Sous-types

Dans cette section nous présentons le code analysant les contraintes de sous-typage, et construisant les sous-arbres de syntaxe abstraite associés. Cette partie de l'analyseur est plus ardue à comprendre car l'évaluation des arbres se fait à l'aide de fonctions d'ordre supérieur, c'est-à-dire que les fonctions d'analyse renvoient (souvent) des fonctions qui prennent elle-mêmes en argument une fonction et qui renvoient une fonction (par évaluation partielle)[25]...

```
and ...
and subtypeSpec mode = function
  [⟨ 'Symbol (_, "("); (list_plus subtypeValueSet "|" Abort) cons; (term_sym ")") _ ⟩]
  → Constraint cons
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Left bracket beginning a subtype specification expected" strm
```

[25] *Enjoy having* `fun` *!*

```
and subtypeValueSet mode = function
  [⟨ 'Keyword (_, "INCLUDES"); (xType Abort) t ⟩]
  → Contained t
| [⟨ 'Keyword (_, "MIN"); (subValSetSuf Abort) sub ⟩]
  → sub Min
| [⟨ 'Keyword (_, "FROM"); (subtypeSpec Abort) cons ⟩]
  → (match cons with Constraint c → Alphabet c)
| [⟨ 'Keyword (_, "SIZE"); (subtypeSpec Abort) cons ⟩]
  → (match cons with Constraint c → Size c)
| [⟨ 'Keyword (_, "WITH"); innerTypeSuf inner ⟩]
  → Inner inner
| [⟨ (SVSAux Fail) svs ⟩]
  → svs (fun x → x)
| [⟨ strm ⟩]
  → syntax_error "Subtype value set expected" strm
and ...
```

La fonction d'analyse `subtypeValueSet` reconnaît les différentes clauses possibles de sous-typage. (cf. 8.1.5) Notons, en avant-goût, que l'on évalue la fonction retournée par `SVSAux` avec l'identité polymorphe... Comme nous le verrons plus loin, c'est cet appel qui déclenche *toutes* les évaluations retardées, mises en place par les sous-fonctions d'analyse à partir de `SVSAux`.

```
and ...
and subValSetSuf mode = function
  [⟨ 'Symbol (_, ".."); (option (function [⟨ 'Symbol (_, "<") ⟩] → ())) ltOpt;
     upperEndValue up ⟩]
  → (match ltOpt with
       Some _ → (fun low → Range ((Large, low), (Strict, up)))
     | None → (fun low → Range ((Large, low), (Large, up))))
| [⟨ 'Symbol (_, "<"); (term_sym "..") _;
     (option (function [⟨ 'Symbol (_, "<") ⟩] → ())) ltOpt; upperEndValue up ⟩]
  → (match ltOpt with
       Some _ → (fun low → Range ((Strict, low), (Strict, up)))
     | None → (fun low → Range ((Strict, low), (Large, up))))
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Symbol '..' or symbol '<' expected" strm
and ...
```

La fonction d'analyse `subValSetSuf` lit une contrainte de type par intervalle, exceptée la borne inférieure. Elle renvoie un intervalle abstrait sur la valeur de sa borne inférieure.

```
and ...
and upperEndValue = function
  [⟨ (xValue Fail) val ⟩] → EndVal val
| [⟨ 'Keyword (_, "MAX") ⟩] → Max
| [⟨ strm ⟩]
  → syntax_error "Value or MAX clause expected" strm
and ...
```

Cette fonction d'analyse, `upperEndValue`, est très simple : elle lit la valeur de la borne supérieure d'un intervalle (Valeur spéciale `MAX` incluse.).

```
and ...
and innerTypeSuf = function
  [⟨ 'Keyword (_, "COMPONENT"); (subtypeSpec Abort) cons ⟩]
  → (match cons with Constraint c → SingleConst c)
| [⟨ 'Keyword (_, "COMPONENTS"); multipleTypeConstraints mtc ⟩]
  → MultConst mtc
| [⟨ strm ⟩]
  → syntax_error "Keyword COMPONENT or keyword COMPONENTS expected" strm
and ...
```

La fonction d'analyse `innerTypeSuf` analyse les spécifications générales de sous-typage (*Inner subtyping.* cf. 8.1.5), et construit les sous-arbres correspondants. Le premier motif concerne les contraintes simples, c'est-à-dire qui portent sur les composants du type parent (pour un exemple, cf. [3] §E.5.4). Le second motif correspond aux contraintes multiples, c'est-à-dire pouvant porter *en plus* sur la présence ou non des composants du type parent (pour des exemples, cf. [3] §E.5.2 et §E.5.3).

```
and ...
and multipleTypeConstraints = function
  [⟨ 'Symbol (_, "{"); (option (function [⟨ 'Symbol (_, "..."); (term_sym ",") _ ⟩] → ())) opt;
     (list_opt namedConstraint ",") consts; (term_sym "}") _ ⟩]
  → (match opt with
        Some _ → Partial consts
      | None → Full consts)
| [⟨ strm ⟩]
  → syntax_error "Multiple type constraints expected" strm
and ...
```

La fonction d'analyse `multipleTypeConstraints` lit les contraintes sur la présence des composant du type parent (cf. `innerTypeSuf` ci-dessus). La spécification peut être partielle (`Partial`) ou bien complète (`Full`). Ce qui distingue les deux c'est que dans le premier cas aucune contrainte (même par par défaut) n'est imposée sur les composants implicitement désignés par l'ellipse.

```
and ...
and namedConstraint = function
  [⟨ 'Lower (_, id); (option (subtypeSpec Fail)) subOpt; (option presenceConstraint) presOpt ⟩]
  → NamedConst (Some (Ident id), subOpt, presOpt)
| [⟨ (subtypeSpec Fail) cons; (option presenceConstraint) presOpt ⟩]
  → NamedConst (None, Some cons, presOpt)
| [⟨ presenceConstraint pres ⟩]
  → NamedConst (None, None, Some pres)
and presenceConstraint = function
  [⟨ 'Keyword (_, "PRESENT") ⟩] → Present
| [⟨ 'Keyword (_, "ABSENT") ⟩] → Absent
| [⟨ 'Keyword (_, "OPTIONAL") ⟩] → Option
and ...
```

La fonction d'analyse `namedConstraint` reconnaît les contraintes de sous-typage multiples (cf. ci-dessus.). Remarquons au passage que, malgré leur nom, ces contraintes peuvent tout de même être anonymes...

```
and ...
and SVSAux mode = function
  [⟨ builtInValue bv; (option (subValSetSuf Fail)) subOpt ⟩]
  → (match subOpt with
       Some sub → (fun f → sub (EndVal (f bv)))
     | None → (fun f → Single (f bv)))
| ...
```

La fonction d'analyse `SVSAux` est appelée depuis `subtypeValueSet`, `SVSAux`, $\texttt{SVSAux}_1$, $\texttt{SVSAux}_2$, $\texttt{SVSAux}_3$, $\texttt{SVSAux}_{11}$ et $\texttt{SVSAux}_{21}$. Pour gérer tous ces contextes d'appels différents, nous utilisons des fonctions d'ordre supérieur : `SVSAux` renvoie une fonction qui prend en argument une fonction de type *Value* → *Value*.
Dans ce premier motif, si la valeur optionnelle `subOpt` est « vide », alors c'est que la contrainte de sous-typage associée se réduit à une simple valeur (*SingleValue*) ; sinon c'est que nous avons affaire à une contrainte de type intervalle, dénotée ici par la fonction `subValSetSuf`. Nous mettons donc en place les expressions associées, que nous abstrayons sur le traitement (`f`) à effectuer sur la valeur (`bv`) ; nous retardons ainsi l'évaluation jusqu'au moment où nous pourrons décider de ce que nous construisons (et donc d'où provenait l'appel).

```
| ...
| [⟨ auxType t; (term_sym ":") _; (SVSAux Abort) svs ⟩]
  → (fun f → svs (fun v → f (AnyV (t, v))))
| ...
```

Dans ce second motif nous obtenons une information partielle sur le sous-arbre de syntaxe abstraite que l'on est en train de construire. En effet, `auxType` suivie de " :" nous révèle que nous analysons une valeur instanciée (*AnyValue*). La valeur retournée par `SVSAux` (*svs*) ne nous dit rien sur la nature du sous-arbre construit (Est-ce une contrainte de type par intervalle, ou juste une valeur ?), mais nous pouvons avec ce simple renseignement mettre en place d'ores et déjà le mécanisme d'évaluation retardée. Notons que la position de `f` dans le corps de la fonction retournée par `SVSAux` dans ce motif dépend de l'associativité de l'opérateur " :" (ici à droite).
Pour se convaincre que ce schéma est correct, nous pouvons essayer un petit exemple. Considérons la dérivation suivante :

$$\text{SubtypeValueSet} \stackrel{+}{\Longrightarrow} \text{AuxType}_{t_0} \text{ ":" } \text{AuxType}_{t_1} \text{ ":" } \text{SVSAux}_{svs}$$

En indice apparaissent les sous-arbres retournés par chacune des fonctions d'analyse correspondantes. Pour simplifier, nous utiliserons la notation standard du λ-calcul [4]. (Par rapport à Caml Light, il suffit de remplacer « $\rightarrow$ » par « `.` », et « `fun` » par « λ ».) L'arbre associé à *SubtypeValueSet* est donc calculé par :

$$(\lambda f.(\lambda f.svs(\lambda v.f(\text{AnyV}(t_1, v)))) \ (\lambda v.f(\text{AnyV}(t_0, v)))) \quad \lambda x.x$$

Par β-réductions successives, nous obtenons :

$$(\lambda f.svs(\lambda v.f(\text{AnyV}(t_1, v)))) \quad \lambda v.\text{AnyV}(t_0, v)$$

Puis :

$$svs(\lambda v.\text{AnyV}(t_0, \text{AnyV}(t_1, v)))$$

Ce qui était attendu.

```
| ...
| [⟨ 'Keyword (_, "NULL"); (option SVSAux3) svs3Opt ⟩]
  → (match svs3Opt with
        Some svs3 → svs3
      | None → (fun f → Single (f NullV)))
| ...
```

Dans ce motif est lu le mot-clef NULL, qui peut avoir deux sémantiques différentes (type ou valeur ASN.1) selon la présence ou l'absence de contraintes de sous-typage à suivre. Dans le cas où il apparaît seul, il ne fait aucun doute qu'il dénote une valeur non spécifiée (*NullValue*), constituant ainsi la contrainte de sous-typage (*SingleValue*) recherchée. Sinon nous laissons le soin à la fonction d'analyse $\texttt{SVSAux}_3$ de lever l'ambiguïté.

```
| ...
| [⟨ 'Upper (_, id); SVSAux1 svs1 ⟩]
  → svs1 id
| ...
```

Nous laissons ici tout le travail à la fonction d'analyse `SVSAux`$_1$, qui devra donc avoir en premier argument une chaîne de caractères (dénotant l'identificateur `id`).

```
| ...
| [⟨ 'Lower (_, id); (option SVSAux2) svs2Opt ⟩]
  → (match svs2Opt with
       Some svs2 → svs2 id
     | None → (fun f → Single (f (IntEnumVRefV id))))
| ...
```

Dans ce motif, si la valeur optionnelle `svs2Opt` est « vide », alors nous sommes sûr de lire une contrainte de sous-typage réduite à une valeur (qui est d'ailleurs ambiguë). Dans le cas contraire, la fonction d'analyse `SVSAux`$_2$ se charge du travail. (Elle prend donc en premier paramètre l'identificateur.)

```
| ...
| [⟨ 'Number (_, n); (option (subValSetSuf Fail)) subOpt ⟩]
  → let uInt = IntegerV (SignNum (Plus, int_of_string n))
    in (match subOpt with
          Some sub → (fun f → sub (EndVal (f uInt)))
        | None → (fun f → Single (f uInt)))
| ...
```

Ici nous reconnaissons d'abord un entier (`uInt`). Le traitement ensuite est identique à celui du premier motif (cf. ci-dessus).

```
| ...
| [⟨ 'Symbol (_, "-"); term_num n; (option (subValSetSuf Fail)) subOpt ⟩]
  → let sInt = IntegerV (SignNum (Minus, int_of_string n))
    in (match subOpt with
          Some sub → (fun f → sub (EndVal (f sInt)))
        | None → (fun f → Single (f sInt)))
| ...
```

Par rapport au motif précédent, nous lisons ici un entier négatif (`sInt`).

```
| ...
| [⟨ strm ⟩] → match mode with
                 Fail → raise Parse_failure
               | Abort → syntax_error "Value expected" strm

and SVSAux1 = function
  [⟨ (plus subtypeSpec Fail) cons; (term_sym ":") _; (SVSAux Abort) svs ⟩]
  → (fun u f → svs (fun v → f (AnyV (Type ([ ], DefType ( !curMod, TRef u), cons), v))))
| ...
```

La fonction d'analyse $\texttt{SVSAux}_1$ est appelée seulement depuis `SVSAux`. La fonction renvoyée comme résultat de l'évaluation de ce motif prend en premier argument un identificateur `upper`. Par rapport au second motif de la fonction d'analyse `SVSAux` (cf. ci-dessus), nous savons ici que le type de la valeur instanciée (*AnyValue*) est un sous-type (dont la liste des contraintes est donnée par `cons`).

```
| ...
| [⟨ 'Symbol (_, ":"); (SVSAux Abort) svs ⟩]
  → (fun u f → svs (fun v → f (AnyV (Type ([ ], DefType ( !curMod, TRef u), [ ]), v))))
| ...
```

Ce cas est identique au précédent, excepté que la liste des constraintes de sous-typage est vide.

```
| ...
| [⟨ 'Symbol (_, "."); SVSAux11 svs11 ⟩]
  → (fun u → svs11 (MRef u))
| ...
```

Dans ce motif, nous savons que l'identificateur `upper` qui avait été reconnu dans le troisième motif de la fonction d'analyse `SVSAux` avant l'appel de $\texttt{SVSAux}_1$ était un identificateur de module. Nous passons donc à la fonction d'analyse $\texttt{SVSAux}_{11}$ cet identificateur, en lui laissant le soin de construire le sous-arbre de syntaxe abstraite approprié. Remarquons que nous ne passons pas ici une valeur de type *string*, mais directement un nœud `MRef`, car $\texttt{SVSAux}_{11}$ emploie toujours uniformément cet argument comme une référence de module.

```
| [⟨ strm ⟩]
  → syntax_error "Subtype specification or symbol ':' or symbol '.' expected" strm

and SVSAux2 = function
  [⟨ 'Symbol (_, ":"); (SVSAux Abort) svs ⟩]
  → (fun I f → svs (fun v → f (ChoiceV (Ident I, v))))
| ...
```

Le seul point d'appel de la fonction d'analyse $\texttt{SVSAux}_2$ est dans le cinquième motif de `SVSAux` (cf. ci-dessus). Lorsque le contrôle passe à $\texttt{SVSAux}_2$, nous avons déjà reconnu un identificateur `lower`.

```
| ...
| [⟨ 'Symbol (_, ".."); (option (function [⟨ 'Symbol (_, "<") ⟩] → ())) opt;
    upperEndValue up ⟩]
  → let fRange upMode I f = Range ((Large, EndVal (f (IntEnumVRefV I))), (upMode, up))
    in (match opt with
          Some _ → fRange Strict
        | None → fRange Large)
| ...
```

Puisque ce motif débute par "..", nous sommes maintenant sûr que l'identificateur lu avant l'appel à $\texttt{SVSAux}_2$ (`1`) correspondait à une valeur (ambiguë) de borne inférieure (au sens large) d'une contrainte de type par intervalle (de sous-typage). cf. le cinquième motif de la fonction d'analyse `SVSAux`.

```
| ...
| [⟨ 'Symbol (_, "<"); SVSAux21 svs21 ⟩]
  → svs21
```

Dans ce dernier motif, nous commençons par lire le symbole terminal "<", qui annonce donc que l'identificateur déjà lu juste avant l'appel à la fonction $\texttt{SVSAux}_2$ constitue une borne inférieure (au sens strict) ambiguë d'une contrainte de type par intervalle (de sous-typage). cf. le cinquième motif de la fonction d'analyse `SVSAux`.

```
and ...
and SVSAux3 = function
  [⟨ (plus subtypeSpec Fail) cons; (term_sym ":") _; (SVSAux Abort) svs ⟩]
  → (fun f → svs (fun v → f (AnyV (Type ([ ], NullT, cons), v))))
| ...
```

La fonction d'analyse $\texttt{SVSAux}_3$ n'est appelée que dans le troisième motif de `SVSAux`, après avoir reconnu le mot-clef NULL.
Dans ce premier motif, nous commençons par lire une liste de contraintes de sous-typage (`cons`), suivie du symbole ":". Donc le mot-clef NULL correspondait à un type, et nous pouvons d'ores et déjà former la valeur instanciée (*AnyValue*) dénotée ici. Nous renvoyons une fonction exactement dans le style présenté dans le deuxième motif de la fonction d'analyse `SVSAux`. (cf. ci-dessus.)

```
| ...
| [⟨ 'Symbol (_, ":"); (SVSAux Abort) svs ⟩]
  → (fun f → svs (fun v → f (AnyV (Type ([ ], NullT, [ ]), v))))
| ...
```

Ce cas est identique au précédent, excepté que la liste de contraintes est vide.

```
| ...
| [⟨ (subValSetSuf Fail) sub ⟩]
  → (fun f → sub (EndVal (f NullV)))
```

Dans ce dernier motif, le mot-clef NULL se révèle être une *valeur*, car il fait partie directement de la borne inférieure d'une contrainte de type par intervalle (appel à `subValSetSuf`. cf. ci-dessus.). Nous mettons donc en place la fonction qui fabriquera la borne inférieure, et l'enverra à `sub` qui construira l'intervalle. cf. le premier motif de la fonction d'analyse `SVSAux`.

```
and ...
and SVSAux11 = function
  [⟨ 'Upper (_, id); (star (subtypeSpec Fail)) cons; (term_sym ":") _; (SVSAux Abort) svs ⟩]
  → (fun m f → svs (fun v → f (AnyV (Type ([ ], DefType (m, TRef id), cons), v))))
| ...
```

La fonction d'analyse $\texttt{SVSAux}_{11}$ est appelée uniquement depuis le troisième motif de la fonction $\texttt{SVSAux}_1$. Nous avions lu avant l'appel un identificateur de module (`m`), suivi d'un point (pour la sélection).
Dans ce premier motif, le symbole " :" nous apprend que nous avons lu une valeur instanciée (*AnyValue*), et l'identificateur `upper` (`id`), suivi de contraintes de sous-typage, nous apprend que le type de la valeur instanciée est un sous-type dans un module externe. Par conséquent, nous mettons en place le mécanisme d'évaluation retardée correspondant.

```
| ...
| [⟨ 'Lower (_, id); (option (subValSetSuf Fail)) subOpt ⟩]
  → (match subOpt with
       Some sub → (fun m f → sub (EndVal (f (DefVal (m, VRef id)))))
     | None → (fun m f → Single (f (DefVal (m, VRef id)))))
| ...
```

Ce dernier motif débute par un identificateur `lower`, donc nous savons que nous avons affaire à une valeur dans un autre module. Si elle est seule (`None`), alors c'est une contrainte de sous-type réduite à une simple valeur (*SingleValue*) ; sinon (`Some sub`), c'est une contrainte de sous-typage par intervalle (`subValSetSuf`).

```
| [⟨ strm ⟩] → syntax_error "Type reference or value reference expected" strm


and SVSAux21 = function
  [⟨ (xType Fail) t; (term_sym ":") _; (SVSAux Abort) svs ⟩]
  → (fun l f → svs (fun v → f (AnyV (Type ([ ], SelectT (Ident l, t), [ ]), v))))
| ...
```

La fonction d'analyse `SVSAux`$_{21}$ est appelée uniquement à partir du troisième motif de `SVSAux`$_2$. Quand nous y entrons, nous avons déjà reconnu le préfixe `lower "<"`.
Ce premier motif débutant par un appel à `xType`, nous savons maintenant que nous avons reconnu un type sélectionné (*SelectionType*). Le symbole " :" qui suit nous apprend ensuite que ce type est celui d'une valeur instanciée (*AnyValue*). Nous envoyons alors une fonction mettant en place ces informations, exactement dans le style présenté dans le deuxième motif de la fonction d'analyse `SVSAux`. (cf. ci-dessus.)

```
| ...
| [⟨ 'Symbol (_, ".."); (option (function [⟨ 'Symbol (_, "<") ⟩] → ())) opt;
    upperEndValue up ⟩]
  → let fRange upMode l f = Range ((Strict, EndVal (f (IntEnumVRefV l))), (upMode, up))
    in (match opt with
          Some _ → fRange Strict
        | None → fRange Large)
| ...
```

Ce dernier motif débute par le symbole "..", ce qui dénote que le préfixe `lower "<"` jusqu'à présent reconnu correspondait en fait à une borne inférieure stricte d'une contrainte de sous-typage par intervalle. Nous renvoyons une fonction identique à celle retournée après l'évaluation du deuxième motif de la fonction d'analyse `SVSAux`$_2$, excepté qu'ici la borne inférieure est stricte au lieu d'être large.

```
| [⟨ strm ⟩] → syntax_error "Type or symbol '..' expected" strm
;;


let parser = specification;;                (* This function is exported from the module. *)
```

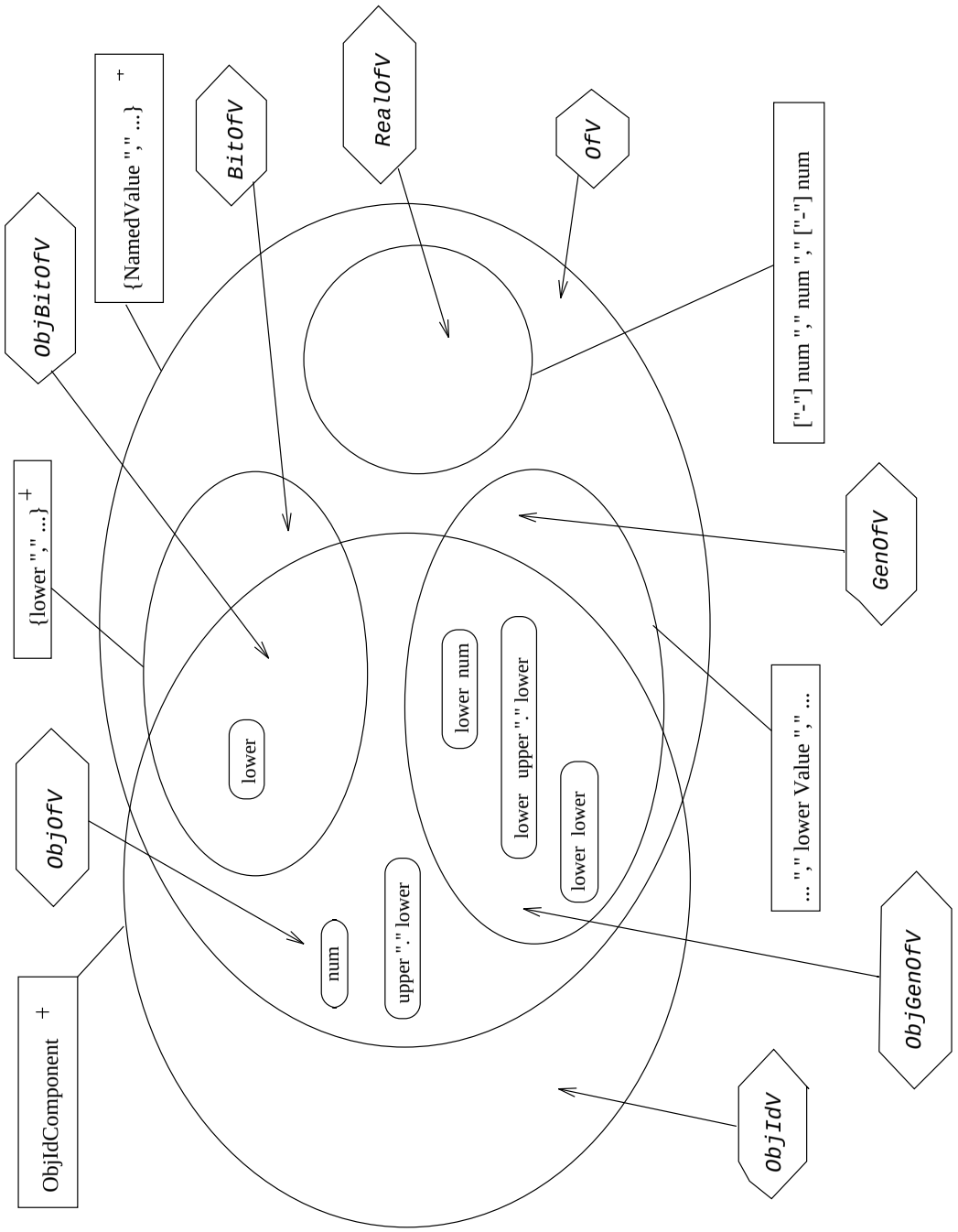


Figure 1 : Ambiguïtés des valeurs structurées ASN.1.

9 Macros ASN.1

Le langage ASN.1 inclut une construction destinée à étendre sa propre syntaxe : les macros. Typiquement, cela rend le langage ASN.1 *contextuel*, c'est-à-dire que la reconnaissance syntaxique ne peut se faire sans ambiguïté que si l'on dispose d'un contexte, soit une connaissance de ce qui précède ou suit l'élément à analyser. *Nous ne tenterons pas ici d'analyser les macros dans toute leur généralité*, mais des macros plus simples, un peu différentes, possédant l'avantage d'être clairement définies et suffisamment générales pour que le spécifieur ne se sente pas bridé. Pour une introduction et une petite critique des macros, nous invitons le lecteur à consulter [9]. Pour leur définition officielle on se rapportera bien sûr à [3]. Dans la présentation qui suit nous supposerons que le lecteur est familier avec les macros et leur vocabulaire.

9.1 Contraintes de réalisation

Nous devons tout d'abord fixer un ensemble de contraintes pour délimiter notre champ d'étude des macros.

1. Incrémentalité de l'intégration
   Nous voulons – et c'est là un critère majeur – pouvoir intégrer le traitement des macros de la façon la plus incrémentale possible dans l'analyseur de base, présenté dans les chapitres précédents. Entre autres choses nous voulons conserver l'arbre de syntaxe abstraite de base comme sous-ensemble du nouvel arbre.

2. Analyse en une passe
   Nous voulons conserver une analyse en une passe. Cela est réalisable grâce aux fonctions d'analyse d'ordre supérieur que nous fournit *gratis* Caml Light. L'idée est de générer dynamiquement des fonctions d'analyse chargées de la reconnaissance syntaxique des instances de macros (types et valeurs)[26].

3. Analyse descendante des macro-règles[27]
   Nous désirons utiliser le méchanisme d'analyse des flux Caml Light pour la reconnaissance des *instances* de macros. Pour relâcher un peu cette condition, nous adjoindrons en plus la possibilité de rebroussement limité[28], c'est-à-dire que si une erreur de syntaxe survient dans un membre droit, alors nous essayons le membre suivant plutôt que d'échouer définitivement. Si c'est le dernier membre d'une production, alors nous reportons l'échec à la règle appelante dont la fonction d'analyse agira de même.(cf. [7]). Puisque la reconnaissance des instances de macros est sous-tendue par les flux Caml Light, le rédacteur de macros devra prendre garde à l'ordre des membres droits

[26] Nous utiliserons les termes « instance de type » et « instance de valeur » pour qualifier respectivement un type et une valeur fournis par une syntaxe de macro.

[27] Dans [3] est utilisé le terme « production » dans le sens que nous avons défini (cf. 1.2) de « règle ». Attention en lisant le code Caml Light et les messages d'erreurs...

[28] *Limited back-tracking*

(notamment si l'un d'eux est vide, il faut le placer en dernier), et aux récursivités gauches (qui feront boucler indéfiniment l'analyseur). Par contre nous échappons à la contrainte exposée à la section (5.1). Enfin, pour terminer, rappelons que l'ambiguïté d'une grammaire est un problème indécidable.

4. Analyseur lexical non extensible
   Si l'analyseur syntaxique devient extensible, nous choisirons de ne pas rendre tel l'analyseur lexical. Cela étant dit, nous affaiblissons cette restriction en étendant l'analyseur lexical de base, pour qu'il accepte de nouveaux lexèmes (comme par exemple les symboles « > », « ! », etc.).

5. Détection d'erreurs très limitée
   La détection exhaustive des erreurs dans la définition des macros pose un problème épineux. En effet, elle doit se faire *pendant* l'analyse syntaxique alors qu'elle constitue intrinsèquement un diagnostic sémantique, ce qui rend les choses compliquées lorsque l'on veut construire strictement un analyseur syntaxique. Pour simplifier notre approche et montrer sa faisabilité, nous ne détecterons statiquement que les erreurs suivantes :

   - La non-définition d'une macro-règle.
   - La déclaration multiple d'une macro-règle.
   - L'absence de dénotation.

   Nous entendons par « dénotation » la sémantique d'une instance, que ce soit un type ou une valeur ; et la norme dit qu'à toute instance possible doit être associée une et une seule dénotation, c'est-à-dire un type ou une valeur.(cf. [3] §A.3.18) Cette erreur sera détectée au moment de la définition. Nous ne détecterons les types et les valeurs malformés dans les définitions de macros que dans la mesure où l'analyseur syntaxique de base le fait. Si nous disposions d'un analyseur sémantique pour le langage de base, il suffirait d'y faire appel pour s'assurer que la macro est correcte. Insistons sur le fait que, de toutes façons, une étude exhaustive des erreurs et de leur détection statique ou dynamique présuppose que l'on sache interpréter les arcanes de [3]. Sans compter que de nombreuses incohérences sémantiques ont depuis longtemps été mises en évidence.

6. Pas d'importation de macros
   Étant donné que Caml Light ne dispose d'aucun moyen direct pour exporter des valeurs fonctionnelles et que le résultat de l'analyse d'une définition de macro est une paire de fonctions d'analyse (une pour les instances de type et une pour celles de valeur), nous préférons ignorer les importations de macros. Pour réaliser une telle importation, il faudrait imaginer un codage des fonctions Caml Light vers un format symbolique, et l'opération réciproque de décodage. Cela poserait néanmoins des problèmes d'intégration à cause du typage. Néanmoins il existe une façon de simuler cette importation en allant analyser dans le module où elle est définie la macro importée, puis de poursuivre. La difficulté serait moindre car un tel analyseur est déjà disponible

(c'est lui-même !) et son adaptation peu coûteuse. L'inconvénient étant que ce n'est pas une importation *stricto sensu*, donc il faut analyser la définition de macro à chaque fois qu'elle est «importée».

7. Pas de définitions récursives
   Il semblerait que les définitions des dénotations de macros peuvent faire appel à d'autres macros. Nous conserverons cette possibilité mais en en restreignant l'usage : pas de définitions directement ou indirectement récursives.

8. Pas de macro-terminaux[29] `astring` et "`string`"
   Ces deux terminaux de macro sont définis respectivement dans §A.2.7 (puis §A.3.10) et §A.3.12 de [3]. Les deux définitions sont difficultueuses. Pour `astring`, le problème est le même que celui présenté à la section 9.2, point 3. Pour "`string`", l'exemple suivant montre la difficulté :

```
MY-MODULE DEFINITIONS ::= BEGIN

PB MACRO ::=
BEGIN
  TYPE  NOTATION ::= string
  VALUE NOTATION ::= value (VALUE BOOLEAN)
END

T     ::= TEST this is a string
val T ::= TRUE

END
```

   L'analyseur ne peut déterminer quand il doit s'arrêter de consommer des lexèmes (ici les chaînes de caractères "this", "is", "a", "string") et peut donc manger le début d'une déclaration de valeur qui suivrait (ici `val`). Le cas pire étant celui où `T` est littéralement un type sélectionné arbitrairement long.
   Nous proposons alors de supprimer le macro-terminal `astring` et de modifier la sémantique de "`string`" : il faudra maintenant mettre entre guillemets la chaîne dénotée. Ainsi, dans l'exemple précédent, il faut écrire à présent :

```
T     ::= TEST "this is a string"
val T ::= TRUE
```

[29] C'est-à-dire une production de la règle 'SymbolDefn' dénotant un terminal dans une macro-règle.

9.2 Quelques conséquences

1. Les points 1 et 4 de la section 9.1 nous conduisent à abandonner la convention des identificateurs de valeurs locales aux macros. En effet, celle-ci suit celle des identificateurs de types (cf. §A.2.8 de [3]), et rend par conséquent impossible la réutilisation de l'analyseur syntaxique de base pour reconnaître les valeurs ASN.1. Nous conserverons donc à l'intérieur des macros les mêmes conventions lexicales qu'à l'extérieur. Plus prosaïquement, nous confondrons le lexème `localvaluereference` avec `valuereference`, et non avec `typereference` (cf. 9.3.2). Toujours au sujet des conventions lexicales, notons que le terminal `macroreference` est un cas particulier de `typereference` : tous les caractères composant son identificateur doivent être en majuscules. C'est pourquoi nous ne pouvons détecter ces identificateurs de macros lors de l'analyse lexicale, et nous reporterons donc cette vérification à une éventuelle phase d'analyse sémantique en identifiant `macroreference` et `typereference`.

2. Une conséquence immédiate du point 2 de la section 9.1 est qu'il faut impérativement que les définitions de macros apparaissent *avant* les instances. Nous avons alors gratuitement la réalisation du point 7 de la section (9.1).

3. Le point 4 de la section 9.1 implique par exemple que le terminal « `"->"` », bien qu'accepté dans une définition de macro, ne se révèle pas valide car, lors de l'instanciation[30], l'analyseur lexical fournira successivement les lexèmes « - » et « > », et non pas « -> » qui n'est pas un lexème ASN.1. L'instance ne sera donc pas reconnue. Il faut subséquemment être prudent et décomposer soi-même le terminal de macro en une suite de terminaux compréhensibles par l'analyseur lexical. Ainsi, dans l'exemple donné cidessus, il convenait de spécifier « `"-" ">"` » pour qu'il n'y ait pas de problème. On pourrait suggérer de faire faire ce découpage des terminaux de macros par l'analyseur syntaxique pour les mettre en correspondance avec ceux de l'analyseur lexical. Cela serait aisé à faire, mais il ne faudrait pas oublier que dans ce cas les espaces dans les terminaux de macros ne seraient pas significatifs, car les espaces sont tous impitoyablement éliminés par l'analyseur lexical...

4. La méthode d'analyse adoptée au point 3 de la section 9.1 a une répercussion importante sur la détection des erreurs de syntaxe dans les définitions de macros. Pour intégrer la reconnaissance des instances de macros à l'analyseur de base, nous ajouterons un motif en tête des filtres des fonctions d'analyse reconnaissant les types et les valeurs ASN.1. En effet, si nous le placions en dernier, l'analyseur essaierait d'abord[31] de reconnaître un type ou une valeur de base en lieu et place d'une instance, et très probablement échouerait — sans avoir tenté de lire une instance de macro. Si l'analyseur échoue en tentant de lire une instance de macro, il croira que c'est peut-être une valeur ou un type de base qui est là, et il essaiera les motifs suivants. Si c'était bien une instance de macro que le spécifieur avait (mal) écrite, il obtiendra par conséquent

[30] C'est-à-dire la reconnaissance syntaxique d'une instance.
[31] Rappelons que l'ordre d'évalation des filtres de flux est de gauche à droite et de haut en bas.

une notification erreur comme si celle-ci s'était produite pour une valeur ou un type de base. C'est ici l'inconvénient du rebroussement : nous perdons *a priori* la propriété du plus long préfixe valide. Il reste cependant toujours possible de mettre au point une gestion d'erreurs syntaxiques plus précise, mais elle serait bien plus compliquée.

9.3 Transformations de la grammaire des macros

9.3.1 Étape 0

Nous donnons d'abord la forme normalisée, en introduisant la structuration en sections et sous-sections, *et en ayant supprimé la production* `astring` *de la règle 'SymbolDefn'*. cf. 9.1, point (8). D'autre part, la note 2 du §A.3.19 de [3] nous révèle que `localvaluereference` dans la règle 'LocalValueassignement' peut être « VALUE ». Or nous avions décidé implicitement depuis le début que nous analysions ASN.1 avec des *mots-clefs réservés* – dont VALUE fait partie. Par conséquent nous avons dû modifier légèrement la règle 'LocalValueassignment' pour faire apparaître explicitement le mot-clef VALUE.

MacroDefinition	→	`macroreference` MACRO "::=" MacroSubstance
MacroSubstance	→	BEGIN MacroBody END
	|	`macroreference`
	|	Externalmacroreference
MacroBody	→	TypeProduction ValueProduction SupportingProductions
Externalmacroreference	→	`modulereference` "." `macroreference`

TypeProduction	→	TYPE NOTATION "::=" MacroAlternativeList
ValueProduction	→	VALUE NOTATION "::=" MacroAlternativeList

SupportingProductions	→	ProductionList
	|	ε
ProductionList	→	Production
	|	ProductionList Production
Production	→	`productionreference` "::=" MacroAlternativeList
MacroAlternativeList	→	MacroAlternative
	|	MacroAlternativeList "|" MacroAlternative

MacroAlternative	→	SymbolList
SymbolList	→	SymbolElement
	|	SymbolList SymbolElement
SymbolElement	→	SymbolDefn
	|	EmbeddedDefinitions
SymbolDefn	→	`productionreference`
	|	"string"
	|	"identifier"
	|	"number"
	|	"empty"
	|	"type"
	|	"type" "(" `localtypereference` ")"
	|	"value" "(" MacroType ")"
	|	"value" "(" `localvaluereference` MacroType ")"
	|	"value" "(" VALUE MacroType ")"

EmbeddedDefinitions	→	"<" EmbeddedDefinitionList ">"
EmbeddedDefinitionList	→	EmbeddedDefinition
	|	EmbeddedDefinitionList EmbeddedDefinition
EmbeddedDefinition	→	LocalTypeassignment
	|	LocalValueassignment
LocalTypeassignment	→	`localtypereference` "::=" MacroType
LocalValueassignment	→	`localvaluereference` MacroType "::=" MacroValue
	|	VALUE MacroType "::=" MacroValue

MacroType	→	`localtypereference`
	|	Type
MacroValue	→	`localvaluereference`
	|	Value

9.3.2 Étape 1

Nous tiendrons compte des ambiguïtés lexicales qui nous amènent à confondre `macroreference`, `productionreference`, `localtypereference`, avec `typereference` ; et `localvaluereference` avec `valuereference`. Quand le contexte le permettra, nous préciserons si le terminal `upper` dénote un identificateur de macros ou un identificateur de production en l'indiçant (respectivement $\texttt{upper}_{mac}$ et $\texttt{upper}_{prod}$).

MacroDefinition	→	$\texttt{upper}_{mac}$ MACRO "::=" MacroSubstance
MacroSubstance	→	BEGIN MacroBody END
	|	`upper` ["." $\texttt{upper}_{mac}$]
MacroBody	→	TypeProduction ValueProduction [ProductionList]

Expansion globale de 'ExternalMacroreference' puis factorisation préfixe de 'MacroSubstance'.
Option de 'SupportingProductions' puis expansion globale.

TypeProduction	→	TYPE NOTATION "::=" MacroAlternativeList
ValueProduction	→	VALUE NOTATION "::=" MacroAlternativeList

ProductionList	→	Production^+
Production	→	$\texttt{upper}_{prod}$ "::=" MacroAlternativeList
MacroAlternativeList	→	$\{$ MacroAlternative "|" ... $\}^+$

Arden de 'ProductionList'.
Arden de 'MacroAlternativeList'.

MacroAlternative	→	SymbolElement$^+$
SymbolElement	→	SymbolDefn
	|	EmbeddedDefinitions
SymbolDefn	→	$\texttt{upper}_{prod}$
	|	"string"
	|	"identifier"
	|	"number"
	|	"empty"
	|	"type" ["(" $\texttt{upper}_{typ}$ ")"]
	|	"value" "(" Bind ")"
Bind	→	[$\texttt{lower}_{val}$] MacroType
	|	VALUE MacroType

Arden de 'SymbolList' et expansion globale.
Factorisation préfixe et bifixe de 'SymbolDefn' (Création de 'bind'.).

EmbeddedDefinitions	→	"<" EmbeddedDefinition$^+$ ">"
EmbeddedDefinition	→	$\texttt{upper}_{typ}$ "::=" MacroType
	|	$\texttt{lower}_{val}$ MacroType "::=" MacroValue
	|	VALUE MacroType "::=" MacroValue

Arden de 'EmbeddedDefinitionList' puis expansion globale.
Expansion globale de 'LocalTypeassignment'.
Expansion globale de 'LocalValueassignment'.

MacroType	→	Type
MacroValue	→	Value

Élimination de la variante upper_{typ} de la règle *'MacroType'* car Type $\Longrightarrow$ upper
Élimination de la variante lower_{val} de la règle *'MacroValue'* car Value $\Longrightarrow$ lower

9.3.3 Étape 2

MacroDefinition → $\texttt{upper}_{mac}$ MACRO "::=" MacroSubstance
MacroSubstance → BEGIN MacroBody END
| `upper` ["." $\texttt{upper}_{mac}$]
MacroBody → TypeProduction ValueProduction Production*

Expansion globale de 'ProductionList'.

TypeProduction → TYPE NOTATION "::=" { MacroAlternative "|" ...}+
ValueProduction → VALUE NOTATION "::=" { MacroAlternative "|" ...}+
Production → $\texttt{upper}_{prod}$ "::=" { MacroAlternative "|" ...}+

Expansion globale de 'MacroAlternativeList'.

MacroAlternative → SymbolElement+
SymbolElement → $\texttt{upper}_{prod}$
| SymbolDefn
| "<" EmbeddedDefinition+ ">"
SymbolDefn → "string"
| "identifier"
| "number"
| "empty"
| "type" ["(" $\texttt{upper}_{typ}$ ")"]
| "value" "(" Bind ")"
Bind → [$\texttt{lower}_{val}$] Type
| VALUE Type

Expansion partielle de la variante $\texttt{upper}_{prod}$ de la règle 'SymbolDefn' dans la règle 'SymbolElement'.
Expansion globale de 'MacroType'.
Expansion globale de 'EmbeddedDefinitions'.

EmbeddedDefinition → $\texttt{upper}_{typ}$ "::=" Type
| $\texttt{lower}_{val}$ Type "::=" Value
| VALUE Type "::=" Value

Expansion globale de 'MacroType' et 'MacroValue'.

9.3.4 Étape 3

MacroDefinition → $\texttt{upper}_{mac}$ MACRO "::=" MacroSubstance
MacroSubstance → BEGIN MacroBody END
| upper ["." $\texttt{upper}_{mac}$]
MacroBody → TypeProduction VALUE NOTATION "::="
{ MacroAlternative "|" ... }$^+$ Production*

Expansion globale de 'ValueProduction'.

TypeProduction → TYPE NOTATION "::=" { MacroAlternative "|" ... }$^+$
Production → $\texttt{upper}_{prod}$ "::=" { MacroAlternative "|" ... }$^+$

Expansion globale de 'ValueProduction'.

MacroAlternative → SymbolElement$^+$
SymbolElement → $\texttt{upper}_{prod}$
| PartElem
PartElem → SymbolDefn
| "<" EmbeddedDefinition$^+$ ">"
SymbolDefn → "string"
| "identifier"
| "number"
| "empty"
| "type" ["(" $\texttt{upper}_{typ}$ ")"]
| "value" "(" Bind ")"
Bind → NamedType
| VALUE Type

Réduction dans la règle 'SymbolElement' (Création de 'PartElem'.)
Nous savons que NamedType $\Longrightarrow$ [lower_{id}] Type
cf. (3.2.3). Donc, en réécrivant NamedType $\Longrightarrow$ [lower] Type
nous pouvons effectuer une expansion totale inverse dans la règle 'Bind',
et y faire apparaître une occurrence de 'NamedType'.

EmbeddedDefinition → $\texttt{upper}_{typ}$ "::=" Type
| $\texttt{lower}_{val}$ Type "::=" Value
| VALUE Type "::=" Value

9.3.5 Étape 4

Le lecteur attentif aura remarqué le problème suivant :

$$\mathcal{P}(\text{MacroAlternative}) \cap \mathcal{P}(\text{Production}) = \{\ \texttt{upper}\ \}$$

empêchant donc la règle 'MacroBody' d'être LL(1). Pour éliminer cette difficulté, nous allons transformer cette règle et c'est l'objet de cette section que d'en présenter les métamorphoses.

MacroBody → TypeProduction VALUE NOTATION ":=" MacroSuf
MacroSuf → { MacroAlternative "|" ... }$^+$ Production*

Expansion totale inverse. (Création de la règle 'MacroSuf'.)

MacroSuf → MacroAlternative ["|" { MacroAlternative "|" ... }$^+$] Production*

MacroSuf → SymbolElement$^+$ ["|" { MacroAlternative "|" ... }$^+$] Production*

Expansion totale de 'MacroAlternative'.

MacroSuf → SymbolElement Cont
Cont → SymbolElement* ["|" { MacroAlternative "|" ... }$^+$] Production*

MacroSuf → SymbolElement [Cont]
Cont → SymbolElement$^+$ ["|" { MacroAlternative "|" ... }$^+$] Production*
| "|" { MacroAlternative "|" ... }$^+$ Production*
| Production$^+$

Option de 'Cont'.

MacroSuf → SymbolElement [Cont]
Cont → SymbolElement [Cont]
| "|" MacroSuf
| Production$^+$

Nous reconnaissons 'Cont' et 'MacroSuf'. (Expansions totales inverses.)

MacroSuf → SymbolElement [Cont]
Cont → `upper` [Cont]
| PartElem [Cont]
| "|" MacroSuf
| Production Production*

Expansion totale de 'SymbolElement' dans 'Cont'.

MacroSuf → SymbolElement [Cont]
Cont → PartElem [Cont]
| "|" MacroSuf
| `upper` [Cont]
| `upper`$_{prod}$ "::=" { MacroAlternative "|" ...}$^+$ Production*

Expansion totale de 'Production' dans 'Cont'.

MacroSuf → SymbolElement [Cont]
Cont → PartElem [Cont]
| "|" MacroSuf
| `upper` [ContSuf]
ContSuf → Cont
| "::=" MacroSuf

Factorisation préfixe de 'Cont'. (Création de la règle 'ContSuf'.)

9.3.6 Bilan

MacroDefinition	→	$\texttt{upper}_{mac}$ MACRO "::=" MacroSubstance
MacroSubstance	→	BEGIN MacroBody END
	|	`upper` ["." $\texttt{upper}_{mac}$]
MacroBody	→	TypeProduction VALUE NOTATION "::=" MacroSuf
TypeProduction	→	TYPE NOTATION "::=" { MacroAlternative "|" ... }$^+$
MacroAlternative	→	SymbolElement$^+$
MacroSuf	→	SymbolElement [Cont]
Cont	→	PartElem [Cont]
	|	"|" MacroSuf
	|	`upper` [ContSuf]
ContSuf	→	Cont
	|	"::=" MacroSuf
SymbolElement	→	$\texttt{upper}_{prod}$
	|	PartElem
PartElem	→	SymbolDefn
	|	"<" EmbeddedDefinition$^+$ ">"
SymbolDefn	→	"string"
	|	"identifier"
	|	"number"
	|	"empty"
	|	"type" ["(" $\texttt{upper}_{typ}$ ")"]
	|	"value" "(" Bind ")"
Bind	→	NamedType
	|	VALUE Type
EmbeddedDefinition	→	$\texttt{upper}_{typ}$ "::=" Type
	|	$\texttt{lower}_{val}$ Type "::=" Value
	|	VALUE Type "::=" Value

9.4 Nouvelle grammaire complète d'ASN.1

Nous devons greffer cette nouvelle grammaire des macros sur celle de base, [3] ne disant pas explicitement comment s'effectue cette greffe. En fait il faut intégrer la définition des macros parmi les définitions de types et de valeurs de base. À partir de la forme initiale, nous obtenons les transformations suivantes :

Assignment → `upper`$_{typ}$ "::=" Type
| `lower`$_{val}$ Type "::=" Value

Assignment → `upper`$_{typ}$ "::=" Type
| `lower`$_{val}$ Type "::=" Value
| MacroDefinition

Assignment → `upper`$_{typ}$ "::=" Type
| `lower`$_{val}$ Type "::=" Value
| `upper`$_{mac}$ MACRO "::=" MacroSubstance

Expansion globale de 'MacroDefinition'.

Assignment → `upper` AssSuf
| `lower`$_{val}$ Type "::=" Value
AssSuf → MACRO "::=" MacroSubstance
| "::=" Type

Factorisation préfixe de 'MacroDefinition'.

Donc finalement, la forme finale de la nouvelle grammaire complète de *full*-ASN.1 est :

MODULES

ModuleDefinition	→	ModuleIdentifier DEFINITIONS [TagDefault TAGS] “::=” BEGIN [ModuleBody] END
ModuleIdentifier	→	`upper`$_{mod}$ [“{” ObjIdComponent$^+$ “}”]
ObjIdComponent	→	`number`
	|	`upper`$_{mod}$ “.” `lower`$_{val}$
	|	`lower` [“(” ClassNumber “)”]
TagDefault	→	EXPLICIT
	|	IMPLICIT
ModuleBody	→	[Exports] [Imports] Assignment$^+$
Exports	→	EXPORTS {Symbol “,” ...}* “;”
Imports	→	IMPORTS SymbolsFromModule* “;”
SymbolsFromModule	→	{Symbol “,” ...}$^+$ FROM ModuleIdentifier
Symbol	→	`upper`$_{typ}$
	|	`lower`$_{val}$
Assignment	→	`upper` AssSuf
	|	`lower`$_{val}$ Type “::=” Value
AssSuf	→	MACRO “::=” MacroSubstance
	|	“::=” Type

TYPES

Type	→	$\texttt{lower}_{id}$ "<" Type
	|	**upper** ["." $\texttt{upper}_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType
AuxType	→	"[" [Class] ClassNumber "]" [TagDefault] Type
	|	BuiltInType SubtypeSpec*
	|	SetSeq [TypeSuf]
SetSeq	→	SET
	|	SEQUENCE
TypeSuf	→	SubtypeSpec+
	|	"{" {ElementType "," ...}* "}" SubtypeSpec*
	|	[SIZE SubtypeSpec] OF Type

BuiltInType	→	BOOLEAN
	|	INTEGER ["{" {NamedNumber "," ...}+ "}"]
	|	BIT STRING ["{" {NamedBit "," ...}+ "}"]
	|	OCTET STRING
	|	CHOICE "{" {NamedType "," ...}+ "}"
	|	ANY [DEFINED BY $\texttt{lower}_{id}$]
	|	OBJECT IDENTIFIER
	|	ENUMERATED "{" {NamedNumber "," ...}+ "}"
	|	REAL
	|	"NumericString"
	|	"PrintableString"
	|	"TeletexString"
	|	"T61String"
	|	"VideotexString"
	|	"VisibleString"
	|	"ISO646String"
	|	"IA5String"
	|	"GraphicString"
	|	"GeneralString"
	|	EXTERNAL
	|	"UTCTime"
	|	"GeneralizedTime"
	|	"ObjectDescriptor"

NamedType	→	`lower` [“<”] Type
	|	`upper` [“.” $\texttt{upper}_{typ}$] SubtypeSpec*
	|	NULL SubtypeSpec*
	|	AuxType

NamedNumber	→	$\texttt{lower}_{id}$ “(” AuxNamedNum “)”
AuxNamedNum	→	[“-”] `number`
	|	[$\texttt{upper}_{mod}$ “.”] $\texttt{lower}_{val}$

NamedBit	→	$\texttt{lower}_{id}$ “(” ClassNumber “)”

ElementType	→	NamedType [ElementTypeSuf]
	|	COMPONENTS OF Type
ElementTypeSuf	→	OPTIONAL
	|	DEFAULT Value

Class	→	UNIVERSAL
	|	APPLICATION
	|	PRIVATE
ClassNumber	→	`number`
	|	[$\texttt{upper}_{mod}$ “.”] $\texttt{lower}_{val}$

VALEURS

Value	→	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	[“-”] `number`
AuxVal0	→	BuiltInValue
	|	AuxType “:” Value
	|	NULL [SpecVal]
AuxVal1	→	SpecVal
	|	“.” AuxVal11
AuxVal2	→	[“<” Type] “:” Value
AuxVal11	→	$\texttt{upper}_{typ}$ SpecVal
	|	$\texttt{lower}_{val}$
SpecVal	→	SubtypeSpec* “:” Value

BuiltInValue	→	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" [BetBraces] "}"

BetBraces	→	AuxVal0 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	`lower` [AuxBet1]
	|	`upper` AuxBet2
	|	`number` [AuxBet3]
AuxBet1	→	"(" ClassNumber ")" ObjIdComponent*
	|	AuxNamed
	|	AuxVal2 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	AuxVal0 [AuxNamed]
	|	`lower` [AuxBet11]
	|	`number` [AuxBet3]
	|	`upper` AuxBet2
AuxBet2	→	SpecVal [AuxNamed]
	|	"." AuxBet21
AuxBet3	→	ObjIdComponent$^+$
	|	AuxNamed
AuxBet11	→	"(" ClassNumber ")" ObjIdComponent*
	|	ObjIdComponent$^+$
	|	AuxVal2 [AuxNamed]
	|	AuxNamed
AuxBet21	→	`upper`$_{typ}$ SpecVal [AuxNamed]
	|	`lower`$_{val}$ [AuxBet3]
AuxNamed	→	"," {NamedValue "," ...}$^+$
NamedValue	→	`lower` [NamedValSuf]
	|	`upper` AuxVal1
	|	["-"] `number`
	|	AuxVal0
NamedValSuf	→	Value
	|	AuxVal2

SOUS-TYPES

SubtypeSpec	→	"(" {SubtypeValueSet "|" ...}$^+$ ")"
SubtypeValueSet	→	INCLUDES Type
	|	MIN SubValSetSuf
	|	FROM SubtypeSpec
	|	SIZE SubtypeSpec
	|	WITH InnerTypeSuf
	|	SVSAux
SubValSetSuf	→	["<"] ".." ["<"] UpperEndValue
UpperEndValue	→	Value
	|	MAX
InnerTypeSuf	→	COMPONENT SubtypeSpec
	|	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	→	"{" ["..." ","] {[NamedConstraint] "," ...} "}"
NamedConstraint	→	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	|	SubtypeSpec [PresenceConstraint]
	|	PresenceConstraint
PresenceConstraint	→	PRESENT
	|	ABSENT
	|	OPTIONAL
SVSAux	→	BuiltInValue [SubValSetSuf]
	|	AuxType ":" SVSAux
	|	NULL [SVSAux3]
	|	`upper` SVSAux1
	|	`lower` [SVSAux2]
	|	["-"] `number` [SubValSetSuf]
SVSAux1	→	SubtypeSpec* ":" SVSAux
	|	"." SVSAux11
SVSAux2	→	":" SVSAux
	|	".." ["<"] UpperEndValue
	|	"<" SVSAux21
SVSAux3	→	SubtypeSpec* ":" SVSAux
	|	SubValSetSuf
SVSAux11	→	`upper`$_{typ}$ SubtypeSpec* ":" SVSAux
	|	`lower`$_{val}$ [SubValSetSuf]
SVSAux21	→	Type ":" SVSAux
	|	".." ["<"] UpperEndValue

MACROS

MacroSubstance	→	BEGIN MacroBody END
	|	`upper` [“.” `upper`$_{mac}$]
MacroBody	→	TypeProduction VALUE NOTATION “::=” MacroSuf
TypeProduction	→	TYPE NOTATION “::=” { MacroAlternative “|” ...}$^+$
MacroAlternative	→	SymbolElement$^+$
MacroSuf	→	SymbolElement [Cont]
Cont	→	PartElem [Cont]
	|	“|” MacroSuf
	|	`upper` [ContSuf]
ContSuf	→	Cont
	|	“::=” MacroSuf
SymbolElement	→	`upper`$_{prod}$
	|	PartElem
PartElem	→	SymbolDefn
	|	“<” EmbeddedDefinition$^+$ “>”
SymbolDefn	→	“string”
	|	“identifier”
	|	“number”
	|	“empty”
	|	“type” [“(” `upper`$_{typ}$ “)”]
	|	“value” “(” Bind “)”
Bind	→	NamedType
	|	VALUE Type
EmbeddedDefinition	→	`upper`$_{typ}$ “::=” Type
	|	`lower`$_{val}$ Type “::=” Value
	|	VALUE Type “::=” Value

9.5 Preuve de la propriété LL(1) de la grammaire étendue

Nous n'allons pas vérifier la propriété LL(1) de la grammaire étendue (avec les macros) *ex nihilo*, mais incrémentalement, à partir des résultats donnés en (4).

9.5.1 Équation P1

Il est évident de constater que la sous-grammaire des macros n'engendre pas de récursivités gauches.

9.5.2 Équation P2

Il est aisé de vérifier que l'intersection des premiers (lexèmes) de chaque alternative est vide.

9.5.3 Équation P3

Avant de commencer, notons que la modification de la règle *'Assignment'* n'engendre pas de nouvelles contraintes. Pour ce qui est de la sous-grammaire des macros, il vient :

Règle	Contraintes
MacroSubstance	$\{\text{“.”}\} \cap \mathcal{S}(\text{MacroSubstance}) = \emptyset$
TypeProduction	$\{\text{“\|”}\} \cap \mathcal{S}(\text{TypeProduction}) = \emptyset$
MacroAlternative	$\mathcal{P}(\text{SymbolElement}) \cap \mathcal{S}(\text{MacroAlternative}) = \emptyset$
MacroSuf	$\mathcal{P}(\text{Cont}) \cap \mathcal{S}(\text{MacroSuf}) = \emptyset$
Cont	$\mathcal{P}(\text{Cont}) \cap \mathcal{S}(\text{Cont}) = \emptyset$ $\mathcal{P}(\text{ContSuf}) \cap \mathcal{S}(\text{Cont}) = \emptyset$
EmbeddedDefinition	$\mathcal{P}(\text{EmbeddedDefinition}) \cap \{\text{“>”}\} = \emptyset$
SymbolDefn	$\{\text{“(”}\} \cap \mathcal{S}(\text{SymbolDefn}) = \emptyset$

Soit :

(1) $\{\text{“.”}\} \cap \mathcal{S}(\text{MacroSubstance}) = \emptyset$
(2) $\{\text{“\|”}\} \cap \mathcal{S}(\text{TypeProduction}) = \emptyset$
(3) $\mathcal{P}(\text{SymbolElement}) \cap \mathcal{S}(\text{MacroAlternative}) = \emptyset$
(4) $\mathcal{P}(\text{Cont}) \cap \mathcal{S}(\text{MacroSuf}) = \emptyset$
(5) $\mathcal{P}(\text{Cont}) \cap \mathcal{S}(\text{Cont}) = \emptyset$
(6) $\mathcal{P}(\text{ContSuf}) \cap \mathcal{S}(\text{Cont}) = \emptyset$
(7) $\mathcal{P}(\text{EmbeddedDefinition}) \cap \{\text{“>”}\} = \emptyset$
(8) $\{\text{“(”}\} \cap \mathcal{S}(\text{SymbolDefn}) = \emptyset$

Calculons d'abord les premiers :

$\mathcal{P}$(SymbolElement) = { `upper`, "<", "`string`", "`identifier`", "`number`", "`empty`", "`type`", "`value`" }

$\mathcal{P}$(Cont) = { `upper`, "<", "`string`", "`identifier`", "`number`", "`empty`", "`type`", "`value`", "|" }

$\mathcal{P}$(ContSuf) = { `upper`, "<", "`string`", "`identifier`", "`number`", "`empty`", "`type`", "`value`", "|", "::=" }

$\mathcal{P}$(EmbeddedDefinition) = { `upper`, `lower`, VALUE }

Puis les suivants :

$$\begin{aligned}
\mathcal{S}(\text{MacroSubstance}) &= \mathcal{S}(\text{AssSuf}) \\
&= \mathcal{S}(\text{Assignment}) \\
&= \{ \text{END}, \texttt{upper}, \texttt{lower} \}
\end{aligned}$$

$$\mathcal{S}(\text{TypeProduction}) = \{ \text{VALUE} \}$$

$$\begin{aligned}
\mathcal{S}(\text{MacroAlternative}) &= \{\text{"|"}\} \cup \mathcal{S}(\text{TypeProduction}) \\
&= \{ \text{VALUE, "|"} \}
\end{aligned}$$

$$\mathcal{S}(\text{ContSuf}) = \mathcal{S}(\text{Cont})$$

$$\begin{aligned}
\mathcal{S}(\text{MacroSuf}) &= \mathcal{S}(\text{MacroBody}) \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{ContSuf}) \\
&= \{\text{END}\} \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{ContSuf}) \\
&= \{\text{END}\} \cup \mathcal{S}(\text{Cont})
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}(\text{Cont}) &= \mathcal{S}(\text{MacroSuf}) \cup \mathcal{S}(\text{ContSuf}) \\
&= \mathcal{S}(\text{MacroSuf}) \cup \mathcal{S}(\text{Cont}) \\
&= \mathcal{S}(\text{MacroSuf})
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}(\text{SymbolDefn}) &= \mathcal{S}(\text{PartElem}) \\
&= \mathcal{P}(\text{Cont}) \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{SymbolElement}) \\
&= \mathcal{P}(\text{Cont}) \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{MacroAlternative}) \cup \mathcal{S}(\text{MacroSuf}) \\
&= \mathcal{S}(\text{MacroSuf}) \cup \{\text{VALUE, "|", } \texttt{upper}\text{, "<", "}\texttt{string}\text{", "}\texttt{identifier}\text{", "}\texttt{number}\text{", "}\texttt{empty}\text{", "}\texttt{type}\text{", "}\texttt{value}\text{"} \}
\end{aligned}$$

D'où :

$$\mathcal{S}(\text{MacroSuf}) = \mathcal{S}(\text{Cont}) = \mathcal{S}(\text{ContSuf}) = \{\text{END}\}$$

Et :

$$\begin{aligned}\mathcal{S}(\text{SymbolDefn}) \quad = \quad & \{\ \text{END, VALUE, “|”, } \texttt{upper}\text{, “<”, } \texttt{“string”}, \\ & \texttt{“identifier”}\text{, “number”, “empty”, “type”, “value” }\}\end{aligned}$$

Donc le système (1)-(8) est vérifié.

Il reste à s'assurer que les non-terminaux apparaissant dans la sous-grammaire des macros *et* dans la grammaire de base n'induisent pas des ensembles $\mathcal{S}$ qui invalideraient la propriété LL(1) de la la grammaire de base. Les non-terminaux dans ce cas à examiner sont 'NamedType', 'Type' et 'Value'.

Nous avions : $\mathcal{S}(\text{NamedType}) = \{$ “,”, “}”, OPTIONAL, DEFAULT $\}$

Maintenant :

$$\begin{aligned}\mathcal{S}(\text{NamedType}) \quad &= \quad \mathcal{S}(\text{Bind}) \cup \{\text{“,”, “}\}\text{”, OPTIONAL, DEFAULT}\} \\ &= \quad \{\ \text{“)”, “,”, “}\}\text{”, OPTIONAL, DEFAULT }\}\end{aligned}$$

Considérons maintenant où est utilisé $\mathcal{S}$(NamedType). Il est utilisé d'une part pour vérifier les équations (18) et (19) données en 4.2.3, et d'autre part pour calculer $\mathcal{S}$(AuxType), qui sert lui-même uniquement à calculer $\mathcal{S}$(Type). Les équations (18) et (19) sont toujours vérifiées et pour ce qui est de $\mathcal{S}$(Type), celui-ci reste invariant car il contenait déjà “)”.

Les occurrences de 'Type' et 'Value' dans la sous-grammaire des macros introduit dans $\mathcal{S}$(Type) et $\mathcal{S}$(Value) le symbol terminal “>”. Or celui-ci n'existe pas dans la grammaire de base, donc il ne peut interférer dans les calculs de (4.2.3).

Conclusion : La grammaire étendue est LL(1).

9.6 Extension de l'arbre de syntaxe abstraite

Nous allons expliquer pourquoi il est nécessaire d'étendre l'arbre de syntaxe abstraite de base pour pouvoir traiter les macros et comment le faire simplement et incrémentalement. Le problème se divise en deux parties disjointes : le cas des instances de type et celui des instances de valeur.

9.6.1 Instances de types

Le sixième paragraphe de la section A.1 de [3] nous apprend qu'une instance de type peut dépendre d'une instance de valeur. Considérons l'exemple suivant :

```
SAMPLE DEFINITIONS ::=
BEGIN

TEST MACRO ::=
BEGIN
 TYPE  NOTATION ::= empty
 VALUE NOTATION ::= value (VALUE BOOLEAN) | value (VALUE INTEGER)
END

T ::= TEST

END
```

Quel est donc le type de `T` ? Réponse : cela dépend. Si une instance de valeur apparaissait dans le module, alors `T` pourrait être soit booléen, soit entier. En l'absence d'une telle instance de valeur, nous devons le rejeter comme étant incorrectement défini[32]...
C'est là qu'arrive au galop la fin de ce sixième paragraphe qui nous apprend qu'en fait nous devons considérer une instance de type comme un type choix[33] (« *[...] the use of the new type notation is similar to a CHOICE [...].* »). Ainsi dans notre exemple, cela nous amène à penser que `T` serait équivalent (dans un sens qui resterait à définir) à :

```
T ::= CHOICE { field-0 BOOLEAN,
               field-1 INTEGER }
```

Toutefois, les types possibles pour une instance de macro (c'est-à-dire les champs du type choix associé) peuvent contenir des identificateurs locaux à la macro. Par exemple :

[32] Attention ! `T` n'est même pas le type non spécifié NULL.
[33] *ChoiceType*

```
MACRO-INSIDE

DEFINITIONS ::= BEGIN

PAIR
MACRO ::= BEGIN
TYPE NOTATION ::= "TYPEX" "=" type(LT1) "TYPEY" "=" type(LT2)

VALUE NOTATION ::= "(" "X" "=" value(lv1 LT1) ","
                       "Y" "=" value(lv2 LT2)
                    <VALUE SEQUENCE {LT1, LT2} ::= {lv1, lv2}>
                   ")"
END

T1 ::= PAIR
         TYPEX = INTEGER
         TYPEY = BOOLEAN

END
```

Dans ce cas, il faut produire une fermeture de type, c'est-à-dire une paire constituée d'un type (choix) et d'un environnement (un ensemble de définitions locales à la macro qui contiendrait ici celles de `LT1` et `LT2`). Nous déciderons donc que *toute instance de type sera équivalente à une telle fermeture.* Nous allons pour ce faire étendre l'arbre de syntaxe abstraite en rajoutant un nœud «fermeture de type», c'est-à-dire un constructeur supplémentaire `TClos` au type Caml Light `Desc` :

```
type Spec = ...
and ...
and Desc = ...
         | TClos of Desc * Env
and Env == Def list
and ...
;;
```

cf. l'exemple en annexe.

9.6.2 Instances de valeurs

La section précédente consacrée aux instances de types nous amène naturellement à concevoir les instances de valeur comme des fermetures de valeurs, c'est-à-dire une paire formée d'une valeur et d'un environnement contenant des définitions locales à la macro. Plus exactement, ce sont des valeurs choisies (*ChoiceValue*) implicites, couplées à un environnement.

Pour réaliser cela, nous devons ajouter un nœud supplémentaire à l'arbre de syntaxe abstraite, c'est-à-dire un constructeur `VClos` au type Caml Light `Value` :

```
type Spec = ...
and ...
and Value = ...
          | VClos of Value * Env
and ...
;;
```

cf. l'exemple en annexe.

9.7 Modification de l'analyseur syntaxique de base

Nous présentons ici la liste des modifications à apporter à l'analyseur syntaxique de base pour «l'interfacer» avec l'analyseur dédié aux macros, présenté à la section (9.8).

Nous avons vu que l'arbre de syntaxe abstraite ne garde aucune trace de la définition d'une macro ; or nous devons toujours rendre une valeur de type Caml Light `Def` pour chaque définition analysée, donc la solution consiste à utiliser un type optionnel là où nous passions des valeurs de type `Def`. Nous définissons une fonction auxiliaire `useful` (cf. 9.9) qui extrait d'une liste de valeurs optionnelles les valeurs utiles (`None` correspondant à une définition de macro, et `Some` à celle d'un type ou d'une valeur). Ainsi la fonction d'analyse `moduleBody` doit tenir compte de ces valeurs optionnelles :

```
let rec specification strm = ...
and ...
and moduleBody = function
  [⟨ exports ex; (option imports) imOpt; (plus assignment Abort) decls ⟩]
  → (Scope (Import (list_of imOpt), Export ex), useful decls)
| [⟨ imports im; (plus assignment Abort) decls ⟩]
  → (Scope (Import im, Export [ ]), useful decls)
| [⟨ (plus assignment Fail) decls ⟩]
  → (Scope (Import [ ], Export [ ]), useful decls)
and ...
```

Nous devons modifier également la fonction `assignment` pour tenir compte de la nouvelle grammaire étendue :

```
and ...
and assignment mode = function
  [⟨ 'Upper (_, id); assSuf ass ⟩]
  → ass id
```

```
| [⟨ 'Lower (_, id); (xType Abort) t; (term_sym ":::=") _; (xValue Abort) v ⟩]
  → Some (ValDef (VRef id, t, v))
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Type definition or value definition expected" strm


and assSuf = function
  [⟨ 'Keyword (_, "MACRO"); (term_sym ":::=") _; macroSubstance ms ⟩]
  → ms
| [⟨ 'Symbol (_, ":::="); (xType Abort) t ⟩]
  → (fun id → Some (TypeDef (TRef id, t)))
| [⟨ strm⟩]
  → syntax_error "Keyword MACRO or symbol '::=' expected in assignment" strm
and ...
```

De même pour `xType` :

```
and ...
and xType mode = function
  [⟨ 'Lower (_, id); (term_sym "<") _; (xType Abort) t ⟩]
  → let (Type (tags, desc, cons)) = t
    in Type ([ ], SelectT (Ident id, Type (tags, desc, [ ])), cons)
| [⟨ 'Upper (_, id); (option (afterUpper id)) xOpt ⟩]
  → (match xOpt with
       Some x → x
     | None → Type ([ ], DefType (!curMod, TRef id), [ ]))
| [⟨ 'Keyword (_, "NULL"); (star (subtypeSpec Fail)) cons ⟩]
  → Type ([Tag (Universal, NumCat 5)], NullT, cons)
| [⟨ auxType t ⟩]
  → t
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Type expected" strm


and afterUpper id = function
  [⟨ (macroTypeInstance type_notations id) t ⟩]
  → t
| [⟨ accessType ext; (star (subtypeSpec Fail)) cons ⟩]
  → Type ([ ], DefType (MRef id, ext), cons)
```

```
| [⟨ (plus subtypeSpec Fail) cons ⟩]
  → Type ([ ], DefType ( !curMod, TRef id), cons)
and ...
```

Le lecteur remarquera le premier motif de la fonction `afterUpper` qui fait appel à une fonction d'analyse des instances de types (`macroTypeInstance` — cf. 9.9). Elle est en première position car nous essaierons toujours de lire d'abord une instance avant une valeur ou un type de base (cf. point 3 de la section 9.2). Son premier argument `type_notations` est une table de hachage mettant en correspondance les noms des macros déjà rencontrées avec les fonctions d'analyse de leurs instances de type. Son second argument `id` est intéressant : il correspond à un identificateur qui est peut-être celui d'une macro et il est nécessairement passé en argument à `afterUpper` car il apparaît *dans* le filtre de flux. C'est là qu'est mis en évidence le caractère intrinsèquement contextuel du langage.
Pour ce qui est de la fonction d'analyse `xValue`, les choses sont plus simples, car il suffit d'ajouter un premier motif dans lequel nous cherchons à lire une instance de valeur :

```
and ...
and xValue mode = function
  [⟨ (macroValueInstance  !value_notations) v ⟩]
  → v
| ...
and ...
;;
```

La fonction `macroValueInstance` essaie de reconnaître une instance de valeur, et son argument `value_notations` est la liste des fonctions d'analyse des instances de valeur dont la définition (macro) a déjà été rencontrée.

9.8 Extension pour l'analyse syntaxique des macros

```
let rec specification strm = ...
and ...
and macroSubstance = function
  [⟨ 'Keyword (_, "BEGIN"); macroBody mb; (term_kwd "END") _ ⟩]
  → (fun macId → let (tp, vp) = mb
                 in process_prods ();
                    check_instances macId  !macro_types;
                    let tmp = type_den  !macro_types
                    in add type_notations macId (macro_prod (tmp macId) tp);
                       value_notations  := (macro_prod (val_den macId) vp):: !value_notations;
                       macro_types  := [ ];
                       None)
```

La fonction `macroSubstance` lit une définition de macro. Elle retourne une fonction qui prend en argument le nom de la macro, lu par la fonction d'analyse `assSuf` (cf. 9.7). Nous récupérons une paire `mt` constituée d'une liste de fonctions d'analyse `tp` reconnaissant les instances de type, et d'une autre, `vp`, reconnaissant les instances de valeur. Le premier élément de la séquence Caml Light effectue la vérification de la complétude des macro-règles, et remet la table des macro-règles à zéro. Le deuxième élément vérifie si au moins une dénotation (c'est-à-dire un sens) existe pour les instances de la macro. Le troisième forme et stocke la fonction d'analyse des instances de type. Le quatrième fait de même pour les instances de valeur. Le cinquième remet à zéro la référence globale qui contient temporairement la liste des types possibles pour une instance de valeur. Le sixième et dernier élément (donc la valeur retournée) est la valeur optionnelle `None` pour indiquer que c'était une déclaration de macro. Ce `None` sera éliminé au niveau de la fonction `moduleBody` grâce à la fonction auxiliaire `useful`. Cela correspond au fait que les déclarations de macros ne doivent pas laisser de traces dans l'arbre de syntaxe abstraite (ce serait inutile).

```
| [⟨ 'Upper (_, up);
    (option (function [⟨ 'Symbol (_, "."); term_macro extMacId ⟩] → extMacId)) macOpt ⟩]
  → (fun _ → None)
```

Ici nous ignorons les importations de macros, comme dit dans le point 6 de (9.1).

```
| [⟨ strm ⟩]
  → syntax_error "Macro reference or macro definition expected" strm


and macroBody = function
  [⟨ typeProduction tp; (term_kwd "VALUE") _; (term_kwd "NOTATION") _;
    (term_sym "::=") _; macroSuf vp ⟩]
  → (tp, vp)
| [⟨ strm ⟩]
  → syntax_error "Type production in macro notation expected" strm
```

La fonction d'analyse `macroBody` retourne la paire constituée des listes d'analyseurs reconnaissant respectivement les instances de type et celles de valeurs.

```
and typeProduction = function
  [⟨ 'Keyword (_, "TYPE"); (term_kwd "NOTATION") _;
    (term_sym "::=") _; (list_plus symbolList "|" Abort) rhs ⟩]
  → rhs
```

La fonction d'analyse `typeProduction` reconnaît et retourne une liste de macro-productions (`symbolList`).

```
and macroSuf = function
  [⟨ (symbolElement Fail) sym; (option cont) cOpt ⟩]
  → (match cOpt with
        Some (raw, mat) → (useful (sym::raw))::mat
      | None → [useful [sym]])
| [⟨ strm ⟩]
  → syntax_error "Symbol element expected in macro-production" strm


and cont = function
  [⟨ partElem pe; (option cont) cOpt ⟩]
  → (match cOpt with
        Some (raw, mat) → (pe::raw, mat)
      | None → ([pe], []))
| [⟨ 'Symbol (_, "|"); macroSuf ms ⟩]
  → ([], ms)
| [⟨ 'Upper (_, id); (option contSuf) fOpt ⟩]
  → (match fOpt with
        Some f → f id
      | None → ([prod_call id], []))
```

La fonction d'analyse `prod_call` enregistre l'appel dans la macro-production d'une macro-règle associée au non-terminal d'identificateur `id`. Si celui-ci correspondait à une macro-règle déjà déclarée, alors une référence sur celle-ci est retournée. Sinon nous créons une fonction d'analyse de macro-règle vide (en attendant la définition) et nous retournons une référence sur celle-ci. cf. (9.9). Remarque : l'utilisation de ces références rend possibles les appels récursifs de macro-règles.

```
and contSuf = function
  [⟨ cont c ⟩]
  → let (raw, mat) = c in (fun pId → ((prod_call pId)::raw, mat))
| [⟨ 'Symbol (_,":::="); macroSuf ms ⟩]
  → fun pId → (prod_decl pId ms; ([], []))
```

La fonction d'analyse `prod_decl` enregistre une macro-règle qui a été reconnue.

```
and partElem = function
  [⟨ symbolDefn parser ⟩]
  → parser
| [⟨ 'Symbol (_, "<"); (plus embeddedDefinition Abort) defs; (term_sym ">") _ ⟩]
  → Some (ref (NTerm (fun cnt strm → (defs, cnt, strm))))
```

```
and symbolList mode = function
  [⟨ (plus symbolElement Fail) parsers ⟩]
  → useful parsers
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Right-hand of macro-production expected" strm


and symbolElement mode = function
  [⟨ 'Upper (_, pId) ⟩]
  → prod_call pId
| [⟨ partElem pe ⟩]
  → pe


and symbolDefn = function
  [⟨ 'XString (_, str) ⟩]
  → let conc = sub_string str 1 (string_length str − 2) in
    let get_syntax = function
      Keyword (_, syn) → syn
    | Lower (_, syn) → syn
    | Upper (_, syn) → syn
    | Number (_, syn) → syn
    | BasedNum (_, syn) → syn
    | XString (_, syn) → syn
    | Symbol (_, syn) → syn
    | _ → failwith "Fatal error in 'symbolDefn'. Please report." in
    let peep token = if conc = get_syntax token
                     then [ ]
                     else raise Parse_failure
    in Some (ref (Term peep))
| [⟨ 'Lower (_, "identifier") ⟩]
  → Some (ref (Term (function Lower (_, _) → [ ]
                            | _ → raise Parse_failure)))
| [⟨ 'Lower (_, "number") ⟩]
  → Some (ref (Term (function Number (_, _) → [ ]
                            | _ → raise Parse_failure)))
| [⟨ 'Lower (_, "string") ⟩]
  → Some (ref (Term (function XString (_, _) → [ ]
                            | _ → raise Parse_failure)))
| [⟨ 'Lower (_, "empty") ⟩]
  → None
```

```
| [⟨ 'Keyword (_, "type"); (option localTypeSuf) ltrOpt ⟩]
  → (match ltrOpt with
       Some ltr → Some (ref (STerm (function [⟨ (xType Fail) t ⟩] → [TypeDef (TRef ltr, t)])))
     | None → Some (ref (STerm (function [⟨ (xType Fail) _ ⟩] → [])))))
| [⟨ 'Keyword (_, "value"); (term_sym "(") _; bind lk; (term_sym ")") _ ⟩]
  → (match lk with
       NamedType (Some (Ident "@"), t)
       → let field = new_field t
         in Some (ref (STerm (function [⟨ (xValue Fail) v ⟩]
                                       → [ValDef (VRef "@", t, ChoiceV (field, v))])))
     | NamedType (Some (Ident id), t)
       → Some (ref (STerm (function [⟨ (xValue Fail) v ⟩] → [ValDef (VRef id, t, v)])))
     | NamedType (None, t)
       → Some (ref (STerm (function [⟨ (xValue Fail) v ⟩] → [ValDef (VRef "", t, v)])))))

and localTypeSuf = function
  [⟨ 'Symbol (_, "("); term_type ltr; (term_sym ")") _ ⟩] → ltr

and bind = function
  [⟨ 'Keyword (_, "VALUE"); (xType Abort) t ⟩]
  → NamedType (Some (Ident "@"), t)
| [⟨ (namedType Fail) nt ⟩]
  → nt
| [⟨ strm ⟩]
  → syntax_error "Macro-value binding expected" strm

and embeddedDefinition mode = function
  [⟨ 'Upper (_, id); (term_sym "::=") _; (xType Abort) t ⟩]
  → TypeDef (TRef id, t)
| [⟨ 'Lower (_, id); (xType Abort) t; (term_sym "::=") _; (xValue Abort) v ⟩]
  → ValDef (VRef id, t, v)
| [⟨ 'Keyword (_, "VALUE"); (xType Abort) t; (term_sym "::=") _; (xValue Abort) v ⟩]
  → let field = new_field t
    in ValDef (VRef "@", t, ChoiceV (field, v))
| [⟨ strm ⟩]
  → match mode with
      Fail → raise Parse_failure
    | Abort → syntax_error "Embedded definition expected" strm
;;
```

9.9 Module auxiliaire pour gérer les macros

```
type α Prod_stat =
       Call of α
     | Decl of α
;;

type (α, β) Component = Term of α → β
                      | NTerm of int → α stream → β * int * α stream
                      | STerm of α stream → β
;;

#open "ast";;

#open "lexer";;

#open "hashtbl";;

#open "parser";;

let prods = (new 7 : (string, (Token, Env) Component ref Prod_stat) t);;

let type_notations = (new 7 : (string, Token stream → Type) t);;

let value_notations = ref ([ ] : (Token stream → Value) list);;

let macro_types = ref ([ ] : NamedType list);;

let clear_macros _ = clear type_notations; clear prods; value_notations := [ ]; macro_types := [ ];;

let rec process_prods _ = do_table check prods; clear prods
and check pId = function
  Call _ → failwith ("Undefined macro-production '" ^ pId ^ "'")
| _ → ()
;;

let check_instances macId = function
  [ ] → failwith ("No semantics for instances is defined in macro `" ^ macId ^ "'")
| _ → ()
;;
```

```
let new_field t =
  let ident = Ident ("field-" ^ (string_of_int (list_length !macro_types)))
  in macro_types := !macro_types @ [NamedType (Some ident, t)]; ident
;;
```

```
let useful l = flat_map (function Some x → [x] | None → [ ]) l;;
```

```
let type_den l env = Type ([ ], TClos (ChoiceT l, env), [ ])
;;
let val_den defs = aux [ ] defs
                   where rec aux env = function
                     d::l → (match d with
                               ValDef (VRef "@", t, v) → VClos (v, env@l)
                             | _ → aux (env@[d]) l)
                   | [ ] → failwith "Fatal error in 'val_den'. Please report."
;;
```

```
let stream_copy strm =
  let r = ref strm in
  let c = ref 0
  in (c, stream_from (fun _ → let (v, s) = stream_get !r in incr c; r := s; v))
;;
let item elm cnt strm =
  match elm with
    Term f → let (t, s) = stream_get strm
             in (f t, succ cnt, s)
  | NTerm f → f cnt strm
  | STerm f → let (c, s) = stream_copy strm
              in try
                   let r = f s in let _ = stream_get s in (r, !c + cnt − 1, s)
                 with Parsing_err _ → raise Parse_failure
;;
let rec right_hand pl cnt strm =
  match pl with
    p::l → let (r, c, s) = item !p cnt strm in
           let (r', c', s') = right_hand l c s
           in (r@r', c', s')
  | [ ] → ([ ], cnt, strm)
;;
```

```
let rec prod pm cnt strm =
  match pm with
    [ ] → raise Parse_failure
  | parsers::l → try
                   let (r, c, _) = right_hand parsers cnt strm
                   in (r, c, strm)
                 with Parse_failure → prod l cnt strm
;;

let macro_prod act l strm =
  let (r, c, _) = prod l 0 strm
  in for i = 1 to c do stream_next strm done; act r
;;
```

```
let prod_call pId =
  try
    match find prods pId with
      Call r → Some r
    | Decl r → Some r
  with Not_found → let r = ref (NTerm (fun cnt strm → ([ ], cnt, strm)))
                   in add prods pId (Call r); Some r
;;

let prod_decl pId rhs =
  let p = ref (NTerm (prod rhs))
  in try
       match find prods pId with
         Call r → r := !p;
                  remove prods pId;
                  add prods pId (Decl r)
       | Decl _ → failwith ("Multiple declaration of macro-production '" ^ pId ^ "'")
     with Not_found → add prods pId (Decl p)
;;
```

```
let macroTypeInstance htbl pId =
  let macro = try
    find htbl pId
```

```
  with Not_found → raise Parse_failure
  in function [⟨ macro env ⟩] → env
;;

let rec macroValueInstance = function
  macro::l → (function [⟨ macro ast ⟩] → ast
                       | [⟨ strm ⟩] → macroValueInstance l strm)
| [ ] → raise Parse_failure
;;
```

Annexe

Code de l'analyseur lexical ASN.1

Tout d'abord, nous rappelons l'interface du module Caml Light `lexer.ml`, vu partiellement en (7.2).

```
type Location == int
and Syntax == string
;;

type Token = Keyword of Location * Syntax
           | Lower of Location * Syntax
           | Upper of Location * Syntax
           | Number of Location * Syntax
           | BasedNum of Location * Syntax
           | XString of Location * Syntax
           | Symbol of Location * Syntax
           | Sentry of Location
;;

exception Lexing_err of string * Location * int;;

value lexer : in_channel → Token stream;;

value pos : int ref;;

value incr : int ref → unit;;
```

Nous donnons à présent l'implémentation du module définissant l'analyseur lexical.

```
#open "hashtbl";;
```

Implements the keyword table as an hashtable.

```
let rec built_assoc = function
  key::t → (key, fun loc → Keyword (loc, key))::(built_assoc t)
| [] → []
;;
```

```
let keyword_list =
  [ "EXTERNAL"; "UTCTime"; "GeneralizedTime"; "ObjectDescriptor";
    "NumericString"; "PrintableString"; "TeletexString"; "VideotexString";
    "T61String"; "ISO646String";
    "VisibleString"; "IA5String"; "GraphicString"; "GeneralString";
    "BOOLEAN"; "INTEGER"; "BIT"; "STRING"; "OCTET"; "NULL"; "SEQUENCE";
    "OF"; "SET"; "IMPLICIT"; "CHOICE"; "ANY"; "OBJECT"; "IDENTIFIER";
    "OPTIONAL"; "DEFAULT"; "COMPONENTS"; "UNIVERSAL"; "APPLICATION";
    "PRIVATE"; "TRUE"; "FALSE"; "BEGIN"; "END"; "DEFINITIONS";
    "EXPLICIT"; "ENUMERATED"; "EXPORTS"; "IMPORTS"; "REAL"; "INCLUDES";
    "MIN"; "MAX"; "SIZE"; "FROM"; "WITH"; "PRESENT"; "ABSENT"; "DEFINED";
    "BY"; "PLUS-INFINITY"; "MINUS-INFINITY"; "TAGS"; "COMPONENT";
    "MACRO"; "TYPE"; "VALUE"; "NOTATION"; "type"; "value" ]
;;

let keyword_table =
  (new (list_length keyword_list) : (string, Location → Token) t)
;;

do_list (fun (str, tok) → add keyword_table str tok) (built_assoc keyword_list)
;;
```

Top-level reference as a character counter. Gives tokens' locations in the source file.

```
let pos = ref 0
and incr x = x := !x + 1
;;
```

Auxiliaries functions for the lexer.

```
let lower = function [⟨ '('a'..'z' as c) ⟩] → c;;

let upper = function [⟨ '('A'..'Z' as c) ⟩] → c;;

let letter = function
  [⟨ lower s ⟩] → s
| [⟨ upper u ⟩] → u
;;

let digit = function [⟨ '('0'..'9' as d) ⟩] → d;;

let alpha = function
```

```
  [⟨ letter l ⟩] → l
| [⟨ digit d ⟩] → d
;;

let ext_alpha = function
  [⟨ alpha c ⟩] → c
| [⟨ '('&' | '#' | ''' | '{' | '(' | '[' | '-' | '|' as c) ⟩] → c
| [⟨ '('\'' | '_' | '\\' | '^' | '@' | ')' | ']' | ' ' as c) ⟩] → c
| [⟨ '('+' | '=' | '}' | '$' | '%' | '*' | '!' | '~' as c) ⟩] → c
| [⟨ '('<' | '>' | '?' | ',' | '.' | ';' | '/' | ':' as c) ⟩] → c
;;
```

Parse comments.

```
let rec comment = function
  [⟨ '\n' ⟩] → ()
| [⟨ '-'; aux_com _ ⟩] → ()
| [⟨ '_; comment _ ⟩] → ()
| [⟨ ⟩] → ()
and aux_com = function
  [⟨ '-' ⟩] → ()
| [⟨ comment _ ⟩] → ()
;;
```

Parse a possibly empty concatenation of hexadecimal characters.

```
let rec star_hexa = function
  [⟨ '('0'..'9' | 'a'..'f' | 'A'..'F' as c); star_hexa sh ⟩] → (make_string 1 c) ^sh
| [⟨ ⟩] → ""
;;
```

Parse a possibly empty concatenation of decimal characters.

```
let rec star_dec = function
  [⟨ digit d; star_dec sd ⟩] → (make_string 1 d) ^ sd
| [⟨ ⟩] → ""
;;
```

Parse ASN.1 type references or value references, *except the first char.*

```
let rec aux_ref = function
  [⟨ '‘-‘; after_hyphen r ⟩] → r
| [⟨ alpha c; aux_ref ar ⟩] → (make_string 1 c) ^ ar
| [⟨ ⟩] → “”
and after_hyphen strm = match strm with
  [⟨ '‘-‘; comment _ ⟩] → “”
| [⟨ alpha c ⟩] → “-” ^ (make_string 1 c) ^ (aux_ref strm)
| [⟨ ⟩]
  → raise (Lexing_err (“Reference cannot end with hyphen”, !pos−1, 1))
;;
```

Parse ASN.1 strings, *except the first character* (i.e. the double-quote).

```
let rec aux_string strm = match strm with
  [⟨ ext_alpha c ⟩] → (make_string 1 c) ^ (aux_string strm)
| [⟨ '‘"‘; after_2quote s ⟩] → s
| [⟨ '‘\n‘ ⟩]
  → raise (Lexing_err (“String not terminated”, !pos, 1))
| [⟨ ⟩]
  → raise (Lexing_err (“Illegal character in string”, !pos, 1))
and after_2quote = function
  [⟨ '‘"‘; aux_string r ⟩] → “\"” ^ r
| [⟨ ⟩] → “”
;;
```

Parse base of ASN.1 numeric strings.

```
let hexa_bin = function
  [⟨ '(‘B‘| ‘H‘ as base) ⟩] → make_string 1 base
| [⟨ ⟩]
  → raise (Lexing_err (“Binary or hexadecimal base expected”, !pos, 1))
;;
```

Parse meaningless blanks.

```
let rec skip_blanks strm = match strm with
  [< ' (' ' | '\t' | '\n') >] -> skip_blanks strm
| [< >] -> ()
;;
```

Parse the end of a token beginning with a colon.

```
let rec four strm = match strm with
  [< '':=' >] -> [Symbol (!pos-2, ":=")]
| [< '':' >] -> let head = Symbol (!pos-2, ":") in head::(four strm)
| [< >] -> [Symbol (!pos-2, ":"); Symbol (!pos-1, ":")]
;;

let aux_colon = function
  [< '':'; four l >] -> l
| [< >] -> [Symbol (!pos-1, ":")]
;;
```

Parse the end of a token beginning with a dot.

```
let aux_dot strm = match strm with
  [< '''.' >] -> (match strm with
                    [< '''.' >] -> [Symbol (!pos-2, "...")]
                  | [< >] -> [Symbol (!pos-2, "..")])
| [< >] -> [Symbol (!pos-1, ".")]
;;
```

Predicate true if and only if the string is only made of 0 and 1.

```
let rec is_bin s =
  if string_length s = 0
  then true
  else let c = nth_char s 0 in
       if c = '0' or c = '1'
       then is_bin (sub_string s 1 (string_length s - 1))
       else false
;;
```

Parse ASN.1 numeric strings, *except the first char* (i.e. ').

```
let aux_quote str_num strm = match strm with
  [⟨ ''' ⟩] → let base = hexa_bin strm in
              if not (is_bin str_num) & (base = "B")
              then raise (Lexing_err ("Hexadecimal base expected", !pos, 1))
              else let con_syn = "'" ^ str_num ^ "'" ^ base
                   in [BasedNum (!pos−(string_length str_num)−2, con_syn)]
| [⟨ ''\n' ⟩] → raise (Lexing_err ("Numeric string not terminated", !pos, 1))
| [⟨ '_ ⟩] → raise (Lexing_err ("Illegal character in numeric string", !pos, 1))
| [⟨ ⟩] → raise (Lexing_err ("Numeric string not terminated", !pos, 1))
;;
```

This parser assumes that a number cannot start with 0 (except zero).

```
let aux_zero = function
  [⟨ digit _ ⟩] → raise (Lexing_err ("Left zero in number", !pos−1, 1))
| [⟨ ⟩] → [Number (!pos−1, "0")]
;;
```

Parse a token beginning with an hyphen, *except its first char.*

```
let rec aux_minus = function
  [⟨ ''-'; comment _ ⟩] → [ ]
| [⟨ ⟩] → [Symbol (!pos−1, "-")]
```

Parse all possible tokens.

```
and read_tokens strm =
  skip_blanks strm;
  match strm with
    [⟨ ''\'' ⟩] → [Symbol (!pos, "\\'")]
  | [⟨ ''\\' ⟩] → [Symbol (!pos, "\\")]
  | [⟨ ''-'; aux_minus t ⟩] → t
  | [⟨ ''.'; aux_dot t ⟩] → t
  | [⟨ ''0'; aux_zero t ⟩] → t
  | [⟨ '':'; aux_colon t ⟩] → t
  | [⟨ ''"'; aux_string s ⟩] → let con_syn = "\"" ^ s ^ "\""
                               in [XString (!pos − (string_length s) − 2, con_syn)]
```

```
  | [⟨ ''''; star_hexa s ⟩] → aux_quote s strm
  | [⟨ '('0'..'9' as d); star_dec l ⟩] → [Number (!pos − (string_length l) − 1, (make_string 1 d) ^ l)]
  | [⟨ lower h; aux_ref suf ⟩] → let id = (make_string 1 h) ^ suf in
                                 let loc = !pos − (string_length id)
                                 in (try
                                       [(find keyword_table id) loc]
                                     with Not_found → [Lower (loc, id)])
  | [⟨ upper h; aux_ref suf ⟩] → let id = (make_string 1 h) ^ suf in
                                 let loc = !pos − (string_length id)
                                 in (try
                                       [(find keyword_table id) loc]
                                     with Not_found → [Upper (loc, id)])
  | [⟨ 'c ⟩] → [Symbol (!pos, make_string 1 c)]
;;

let rec inject = function
  x::l → [⟨ 'x; inject l ⟩]
| [] → [⟨ ⟩]
;;
```

The main function.

```
let lexer chan =
  let in_stream =
    stream_from (fun () → let c = input_char chan in incr pos; c) in
  let rec aux_lexer strm =
    match strm with
      [⟨ read_tokens tok_lst ⟩] → [⟨ inject tok_lst; aux_lexer strm ⟩]
    | [⟨ ⟩] → [⟨ ⟩]
  in
  let out_stream = aux_lexer in_stream
  in try
       end_of_stream out_stream;
       [⟨ 'Sentry (1) ⟩]
     with Parse_failure
          → [⟨ out_stream; 'Sentry (1 + in_channel_length chan) ⟩]
;;
```

Quelques exemples sans macros

Nous donnons ici quelques exemples d'application de l'analyseur syntaxique pour ASN.1 présenté dans ce document. Pour rendre les choses plus explicites, la présentation se fait comme si l'on était au niveau de la boucle d'interaction de Caml Light[34]. Après avoir entré à l'invite système[35] `camllight`, nous effectuons les opérations suivantes :

```
##open "sys";;

##open "ast";;

##open "lexer";;

##open "parser";;

##open "errors";;

#load_object "lexer";;
- : unit = ()

#load_object "auxiliaries";;
- : unit = ()

#load_object "macros_aux";;
- : unit = ()

#load_object "parser";;
- : unit = ()

#load_object "errors";;
- : unit = ()

#let analyse filename =
  begin
    let my_file = open_in filename in
    let init_stream = stream_of_channel (open_in filename) in
    let ast = (try
                 parser (lexer my_file)
               with Lexing_err (msg, ofs, len)
                    → let header = "ASN.1:1990 lexer"
                      in print_error header init_stream filename msg ofs len; [ ]
                  | Parsing_err (msg, ofs, len)
                    → let header = "ASN.1:1990 parser"
                      in print_error header init_stream filename msg ofs len; [ ])
    in pos := 0;
       close_in my_file;
       ast
```

[34] *top-level loop*
[35] *prompt*

```
  end
;;
```
analyse : *string* → *Spec list* = ⟨**fun**⟩

Si nous supposons que nous avons un fichier source ASN.1 `ex1.asn1` dont le contenu est un extrait du protocole CMIP :

```
-- Common Management Information Protocol (CMIP)

CMIP {joint-iso-ccitt ms(9) cmip(1) modules(0) aAssociateUserInfo(1)}

DEFINITIONS ::=

BEGIN

FunctionalUnits ::= BIT STRING { multipleObjectSelection (0),
                                 filter (1),
                                 multipleReply (2),
                                 extendedService (3),
                                 cancelGet (4)
                               }

-- Functional unit i is supported if and only if bit i is one.
-- information carried in user-information parameter of A-ASSOCIATE

CMIPUserInfo ::= SEQUENCE { protocolVersion [0] IMPLICIT ProtocolVersion
                                                 DEFAULT { version1 },
                            functionalUnits [1] IMPLICIT FunctionalUnits
                                                 DEFAULT {},
                            accessControl   [2] EXTERNAL
                                                 OPTIONAL,
                            userInfo        [3] EXTERNAL
                                                 OPTIONAL }

ProtocolVersion ::= BIT STRING { version1 (0), version2 (1) }

END
```

Maintenant, nous lançons l'analyse de ce code :

```
#analyse "ex1.asn1";;
- : Spec list
  = [Spec (ModId (MRef "CMIP",
                  [IdVRefObj "joint-iso-ccitt";
                   IdObj (Ident "ms", NumForm 9);
                   IdObj (Ident "cmip", NumForm 1);
                   IdObj (Ident "modules", NumForm 0);
                   IdObj (Ident "aAssociateUserInfo",
                          NumForm 1)]),
          Scope (Import [], Export []),
          [TypeDef (TRef "FunctionalUnits",
                    Type ([Tag (Universal, NumCat 3, Explicit)],
                          BitStrT
                           [NamedBit (Ident "multipleObjectSelection",
                                      NumBit 0);
                            NamedBit (Ident "filter", NumBit 1);
                            NamedBit (Ident "multipleReply", NumBit 2);
                            NamedBit (Ident "extendedService", NumBit 3);
                            NamedBit (Ident "cancelGet", NumBit 4)], []));
           TypeDef (TRef "CMIPUserInfo",
                    Type ([Tag (Universal, NumCat 16, Explicit)],
                          SeqT [Default
                                  (NamedType
                                     (Some (Ident "protocolVersion")
                                      , Type ([Tag (Context, NumCat 0,
                                                    Implicit)],
                                              DefType
                                                (MRef "CMIP",
                                                 TRef "ProtocolVersion"), [])),
                                   ObjBitOfV "version1");
                                Default
                                  (NamedType
                                     (Some (Ident "functionalUnits")
                                      , Type ([Tag (Context, NumCat 1,
                                                    Implicit)],
                                              DefType
                                                (MRef "CMIP",
                                                 TRef "FunctionalUnits"), [])),
                                   EmptyV);
                                Optional
                                  (NamedType
                                     (Some (Ident "accessControl"),
```

```
                                    Type ([ Tag (Context, NumCat 2,
                                                 Implicit);
                                            Tag (Universal,
                                                 NumCat 8, Explicit) ],
                                          UsefulT External, [ ])));
                                Optional
                                 (NamedType
                                   (Some (Ident "userInfo"),
                                    Type ([ Tag (Context, NumCat 3,
                                                 Implicit);
                                            Tag (Universal,
                                                 NumCat 8, Explicit) ],
                                          UsefulT External, [ ]))) ], [ ]));
        TypeDef (TRef "ProtocolVersion",
                 Type ([ Tag (Universal, NumCat 3, Explicit) ],
                       BitStrT [ NamedBit (Ident "version1", NumBit 0);
                                 NamedBit (Ident "version2", NumBit 1) ],
                       [ ]))]) ]
```

Encore un exemple :

```
ASN1DefinedTypesModule {ccitt recommendation(0) m(13) gnm(3100)
                        informationModel(0) asn1Modules(2)
                        asn1DefinedTypesModule(0)}

DEFINITIONS IMPLICIT TAGS ::=

BEGIN

-- EXPORTS everything

IMPORTS
  ObjectInstance,ObjectClass
    FROM CMIP-1 {joint-iso-ccitt ms(9) cmip(1) modules(0) protocol(3)};

ConnectivityPointer ::= CHOICE { none         NULL,
                                 single       ObjectInstance,
                                 concatenated SEQUENCE OF ObjectInstance}

CTPUpstreamPointer ::= ConnectivityPointer(WITH COMPONENTS { ...,
      -- the other two choices are present
```

```
         concatenated ABSENT})

END -- end of ASN1DefinedTypesModule
```

Maintenant, nous lançons l'analyse de ce code :

```
#analyse "ex2.asn1";;
- : Spec list
  = [Spec (ModId (MRef "ASN1DefinedTypesModule",
                  [IdVRefObj "ccitt";
                   IdObj (Ident "recommendation", NumForm 0);
                   IdObj (Ident "m", NumForm 13);
                   IdObj (Ident "gnm", NumForm 3100);
                   IdObj (Ident "informationModel", NumForm 0);
                   IdObj (Ident "asn1Modules", NumForm 2);
                   IdObj (Ident "asn1DefinedTypesModule", NumForm 0)]),
          Scope (Import [SymMod
                           ([SymType (TRef "ObjectInstance");
                             SymType (TRef "ObjectClass")],
                            ModId (MRef "CMIP-1",
                                   [IdVRefObj "joint-iso-ccitt";
                                    IdObj (Ident "ms", NumForm 9);
                                    IdObj (Ident "cmip", NumForm 1);
                                    IdObj (Ident "modules", NumForm 0);
                                    IdObj (Ident "protocol", NumForm 3)]))],
                 Export []),
          [TypeDef (TRef "ConnectivityPointer",
                    Type ([], ChoiceT
                                [NamedType
                                   (Some (Ident "none"),
                                    Type ([Tag (Universal,
                                                NumCat 5, Explicit)],
                                          NullT, []));
                                 NamedType
                                   (Some (Ident "single"),
                                    Type ([], DefType
                                                (MRef "CMIP-1",
                                                 TRef "ObjectInstance"),
                                          []));
                                 NamedType
                                   (Some (Ident "concatenated"),
```

```
                                Type ([ Tag (Universal, NumCat 16,
                                        Explicit) ],
                                     SeqOfT
                                       (Type ([ ], DefType
                                                  (MRef "CMIP-1",
                                                   TRef "ObjectInstance"),
                                                [ ])), [ ])) ], [ ]));
            TypeDef
             (TRef "CTPUpstreamPointer",
              Type ([ ], DefType (MRef "CMIP-1",
                                  TRef "ConnectivityPointer"),
                    [ Constraint
                       [ Inner (MultConst
                                  (Partial
                                    [ Some (NamedConst
                                              (Some (Ident "concatenated"), None,
                                               Some Absent)) ])) ] ])) ])
```

Un dernier pour la route :

```
Attribute-ASN1Module {joint-iso-ccitt ms(9) smi(3) part2(2) asn1Module(2) 1}

DEFINITIONS  IMPLICIT TAGS::=

BEGIN

AvailabilityStatus ::= SET OF INTEGER
                           { inTest(0), failed(1), powerOff(2),
                             offLine(3), offDuty(4), dependency(5),
                             degraded(6), notInstalled (7) , logFull(8)}

LogAvailability ::= AvailabilityStatus (WITH COMPONENT (logFull | offDuty))

END
```

Maintenant, nous lançons l'analyse de ce code :

```
#analyse "ex3.asn1";;
- : Spec list
  =
```

```
[Spec (ModId (MRef "Attribute-ASN1Module",
              [IdVRefObj "joint-iso-ccitt";
               IdObj (Ident "ms", NumForm 9);
               IdObj (Ident "smi", NumForm 3);
               IdObj (Ident "part2", NumForm 2);
               IdObj (Ident "asn1Module", NumForm 2);
               NumObj 1]), Scope (Import [], Export []),
      [TypeDef
        (TRef "AvailabilityStatus",
         Type ([Tag (Universal, NumCat 17,
                     Explicit)],
               SetOfT (Type ([Tag (Universal, NumCat 2,
                                   Explicit)],
                             IntegerT
                              [NamedNum (Ident "inTest",
                                         NumNum (Plus, 0));
                               NamedNum (Ident "failed",
                                         NumNum (Plus, 1));
                               NamedNum (Ident "powerOff",
                                         NumNum (Plus, 2));
                               NamedNum (Ident "offLine",
                                         NumNum (Plus, 3));
                               NamedNum (Ident "offDuty",
                                         NumNum (Plus, 4));
                               NamedNum (Ident "dependency",
                                         NumNum (Plus, 5));
                               NamedNum (Ident "degraded",
                                         NumNum (Plus, 6));
                               NamedNum (Ident "notInstalled",
                                         NumNum (Plus, 7));
                               NamedNum (Ident "logFull",
                                         NumNum (Plus, 8))], [])), []));
       TypeDef (TRef "LogAvailability",
                Type ([], DefType (MRef "Attribute-ASN1Module",
                                   TRef "AvailabilityStatus"),
                      [Constraint [Inner (SingleConst
                                          [Single (IntEnumVRefV
                                                   "logFull");
                                           Single (IntEnumVRefV
                                                   "offDuty")])]])])]
```

Un exemple de macro

L'exemple du §E.3 de [3] :

```
SAMPLE
DEFINITIONS ::= BEGIN

PAIR
MACRO ::= BEGIN
TYPE NOTATION ::= "TYPEX" "=" type(LT1) "TYPEY" "=" type(LT2)

VALUE NOTATION ::= "(" "X" "=" value(lv1 LT1) ","
                       "Y" "=" value(lv2 LT2)
                    <VALUE SEQUENCE {LT1, LT2} ::= {lv1, lv2}>
                   ")"
END

T1 ::= PAIR
         TYPEX = INTEGER
         TYPEY = BOOLEAN

T2 ::= PAIR
         TYPEX = VisibleString
         TYPEY = T1

v1 T1 ::= (X = 3, Y = TRUE)

v2 T2 ::= (X = "Name", Y = (X = 4, Y = FALSE))

END
```

```
- : Spec list
  = [Spec (ModId (MRef "SAMPLE", [ ]),
           Scope (Import [ ], Export [ ]),
           [TypeDef
              (TRef "T1",
               Type ([ ], TClos
                           (ChoiceT
                              [NamedType
                                 (Some (Ident "field-0"),
                                  Type ([Tag (Universal,
                                              NumCat 16,
                                              Explicit)],
```

```
                        SeqT
                         [Mandatory
                            (NamedType
                               (None, Type ([ ], DefType
                                              (MRef "SAMPLE",
                                               TRef "LT1"), [ ])));
                          Mandatory
                            (NamedType
                               (None, Type ([ ], DefType
                                              (MRef "SAMPLE",
                                               TRef "LT2"), [ ])))],
                        [ ]))],
                [TypeDef (TRef "LT1",
                        Type ([Tag (Universal, NumCat 2,
                                    Explicit)], IntegerT [ ], [ ]));
                 TypeDef
                  (TRef "LT2",
                   Type ([Tag (Universal, NumCat 1, Explicit)],
                         BooleanT, [ ]))]), [ ]));
TypeDef
 (TRef "T2",
  Type ([ ], TClos
              (ChoiceT
                [NamedType
                   (Some (Ident "field-0"),
                    Type ([Tag (Universal,
                                NumCat 16,
                                Explicit)],
                          SeqT
                           [Mandatory
                              (NamedType
                                 (None, Type ([ ], DefType
                                                (MRef "SAMPLE",
                                                 TRef "LT1"), [ ])));
                            Mandatory
                              (NamedType
                                 (None, Type ([ ], DefType
                                                (MRef "SAMPLE",
                                                 TRef "LT2"), [ ])))],
                          [ ]))],
                [TypeDef (TRef "LT1",
                        Type ([Tag (Universal, NumCat 26,
```

```
                                        Explicit)], CharStrT Visible,
                                    []));
                    TypeDef (TRef "LT2",
                             Type ([], DefType (MRef "SAMPLE",
                                                TRef "T1"), [])), []));
ValDef (VRef "v1",
        Type ([], DefType (MRef "SAMPLE", TRef "T1"), []),
        VClos (ChoiceV (Ident "field-0",
                        BitOfV [Ident "lv1"; Ident "lv2"]),
               [ValDef (VRef "lv1",
                        Type ([], DefType (MRef "SAMPLE",
                                           TRef "LT1"), []),
                        IntegerV (SignNum (Plus, 3)));
                ValDef (VRef "lv2",
                        Type ([], DefType (MRef "SAMPLE",
                                           TRef "LT2"), []),
                        BooleanV true)]));
ValDef (VRef "v2",
        Type ([], DefType (MRef "SAMPLE", TRef "T2"), []),
        VClos (ChoiceV (Ident "field-0",
                        BitOfV [Ident "lv1"; Ident "lv2"]),
               [ValDef (VRef "lv1",
                        Type ([], DefType (MRef "SAMPLE",
                                           TRef "LT1"), []),
                        CharStrV "Name");
                ValDef
                 (VRef "lv2",
                  Type ([], DefType (MRef "SAMPLE",
                                     TRef "LT2"), []),
                  VClos (ChoiceV (Ident "field-0",
                                  BitOfV [Ident "lv1";
                                          Ident "lv2"]),
                         [ValDef (VRef "lv1",
                                  Type ([], DefType
                                            (MRef "SAMPLE",
                                             TRef "LT1"), []),
                                  IntegerV (SignNum (Plus, 4)));
                          ValDef (VRef "lv2",
                                  Type ([], DefType
                                            (MRef "SAMPLE",
                                             TRef "LT2"), []),
                                  BooleanV false)])])])]
```

Références

[1] A. Aho et J. Ullman. *Theory of parsing, Translation and Compiling*, vol. 1 (Parsing) de *Automatic Computation*. Prentice Hall, 1972, ch. 2.6.3, p. 199.

[2] A. Aho, R. Sethi, et J. Ullman. *Compilateurs : Principes, techniques et outils*. InterÉditions, 1992.

[3] ISO/IEC. *Information technology – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1)*, seconde ed., Décembre 1990. Référence ISO/IEC 8824 : 1990 (E).

[4] J.R. Hindley et J.P. Seldin. *Introduction to Combinators and λ-calculus*, vol. 1 de *LMS Students Texts*. Cambridge University Press, 1986.

[5] X. Leroy. *The Caml Light system, release 0.6 – Documentation and user's manual*. INRIA-Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France, Septembre 1993.

[6] M. Serrano et P. Weis. 1+1=1 : an Optimizing Caml Compiler. Dans *ACM-SIGPLAN Workshop on ML and its Applications* (25-26 Juin 1994). Orlando, Florida (USA).

[7] M. Mauny et D. de Rauglaudre. Parsers in ml. Dans *Proceedings of the ACM International Conference on Lisp and Functional Programming* (San Francisco, USA, 1992).

[8] P. Weis et X. Leroy. *Le langage Caml*. IIA. InterÉditions, 1993.

[9] D. Steedman. *Abstract Syntax Notation One (ASN.1) - The Tutorial and Reference*. Technology Appraisals, 1990, ch. 5 Macros, p. 71–78.

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur
Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
ISSN 0249-6399