INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Parsing ASN.1:1990 with Caml Light

Christian Rinderknecht

N° 171

August 1995

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

Rapport technique

ISSN 0249-6399

Parsing ASN.1:1990 with Caml Light

Christian Rinderknecht

Programme 2 — Calcul symbolique, programmation et génie logiciel

Projet Cristal

Rapport technique n ° 171 — August 1995 — 188 pages

Abstract: ASN.1 is a language for network protocol specification, 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 parser writing in Caml Light is also given.

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

(Résumé : tsvp)

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

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

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 détails.

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

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.

Acknowledgements

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.*

Contents

Introduction

OSI service and protocol specifications[1] are the backbone for design and implementation of distributed heterogeneous systems. These OSI communication standards are specified in natural language and the data exchanged through the application-layout of the ISO model are described using the ASN.1 notation.

This *Abstract Syntax Notation One* is a common standard of ISO [4] and CCITT (X.208) allowing a formal description of types and values. It can be compared in this way to the type and value definitions in programming languages like Pascal or C. The notation includes predefined types like integers, real numbers, booleans, etc., and make possible the definition of structured types like records, ordered sets, or even unions of types (like in C).

A tag may be used to distinguish between different optional elements in a set, or in a union of types; basically it identifies each value type, and each value is transmitted on-line with its tag type.

This paper presents the results of a research study of ASN.1 grammar, in its ISO version 1990 [4]. We showh that we can get another ASN.1 grammar satisfying the LL(1) property[2], using a sequence of transformations from its standard form, and making no compromises with the latter (Three ISO *Technical Corrigenda* were nevertheless added.). A grammar of a context-free language can be considered as a set of rational equations whose lowest solution (in terms of inclusion of sets) is a context-free language. Each transformation is therefore described in function of rational operations (union, product and star), assuring step-by-step the invariance of the language generated by the resulting grammar.
ASN.1 macros are handled, but not in their full generality because it would not be realistic. An on-the-fly macro parsing and dynamic generation (the result of a macro parsing *is* a specific parser) is presented, and also its orthogonal integration to the base parser (without macro-processing facility). So the complete parser works in *one pass*.

We first present a grammar for ASN.1 tokens recognition, and follows a study of ASN.1 syntax. The constraint which guided the election of transformations was the will to make the final grammar top-down parsable, that implies it had *not* to be ambiguous. But there is no algorithm able to decide whether a given grammar is ambiguous or not (it's a theoretical bound too), so it became necessary to make *ad hoc* choices in order to get rid of ambiguities. Moreover, we even didn't know *a priori*, assuming these ambiguities eliminated, if an LL(1) grammar existed for ASN.1 (another theoretical bound). The two preceding reasons lead us to a "hand-made" work. Next, it was known that, given two grammars, the problem of the equality of their generated language is undecidable[1]. Thus the proof of this equality will be a step-by-step fully commented presentation of each syntactic transformation. For

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

sake of simplicity the initial grammar will be split into sections. The proof of the LL(1) property is given after. The technical corrigenda were taken in account from the initial grammar.

This study led to the implementation of a complete parser, fully written in the Caml Light programming language, currently designed and developed at INRIA[8, 5], and freely distributed by anonymous *ftp*. It's a functional language, dialect of ML, which strong static typing assures the good structure and the consistence of values at run-time. Its compiler with type inference makes it a good and fast prototyping language, and it's compilation (to C[6]) assures efficiency. Caml is small and highly portable (PC, Macintosh, MS-DOS, UNIX, MS-Windows). Another point is that it allows direct parser writings, with a special abstract data type called *stream*[7]. These streams behave like lazy lists[3] with destructive semantics. It means that when you access the element at the head, this one is then evaluated (hence laziness) and is bound to be extracted from the list (hence destruction). A specific pattern matching construct allows an exact description of what is happening at run-time, particularly when parsing, and with a Caml syntax very close to a standard grammar notation (for instance BNF). The advantage, compared to YACC, is that we continue taking the benefits of full functionality and formal semantics ("We know *what* is parsed and computed, *when* it is parsed and computed and *how* it is parsed and computed."). The disadvantage is that the grammar must be top-down parsable, with one token of look-ahead and no back-tracking (cf. section 5).

A systematic method for parser writing in Caml Light from a LL(1) grammar is also presented. The parser hence produced is *not* a "black box", because of the particular readability of stream pattern matchings and the code uniformity.

[3]Thus maybe infinite.

1 Notations

The grammar notation used in this document is an extended BNF[4] (rational operators).

1.1 ASN.1 tokens

- Token identifiers appear in lower case.
- Keywords are in upper case.
- Nonterminals identifiers are the concatenation of identifiers in lower case, each one beginning with an upper case letter.
- Extracts of concrete syntax (terminals) lay between inverted commas — if the inverted commas belong to the concrete syntax, they are preceded by a back-slash.

1.2 Grammars

We'll call "production rule" (or for short "rule"), the couple formed by a nonterminal and all the words that can be derived from in one step.

We'll call "right-hand of a production rule" (or for short "right-hand") one of the words (containing or not tokens) that can be derived in one step. For instance, $\mathrm{X}_0\ \mathrm{X}_1\ \ldots \mathrm{X}_n$ is a right-hand of the production rule X in: $\mathrm{X} \rightarrow \mathrm{X}_0\ \mathrm{X}_1\ \ldots \mathrm{X}_n \mid \ldots$

We'll name "production" the set of right-hands of a production rule.

Structure The grammar is divided in sections which are themselves divided in subsections separated by a line. The purpose is to gather the rules which have a close semantics or that contribute to the same semantics. A subsection A separated by the double-line of a subsection B means that A is transformed into B. The order of the rules inside the same subsection is meaningful: it's a breadth-first traversal from an entry rule (see below), considering productions in writing order. Order between entry rules is not significant.

Entry rules Entry rules are rules which are called from outside of the subsection where they are defined. If calls are limited to the current section, the rule is said "local" (to the section); if they are restricted to the outside of the defining section, the rule is said "global", and if they are calls both in the current section and in the other sections, the rule is said "mixed". In the first case the defining nonterminal identifier will appear in italics; in the second it will appear underlined, and in the last in underlined italics. The axiom (or entry point) will be in bold font. Multiple entry rules in the same subsection are allowed.

[4] *Backus-Naur Form*

Remarks In the presentation of temporary grammars, comments will be boxed after the rules. It is mandatory to read them in the order of appearance in the subsection, because they can describe composed transformations (hence sequencing). For the same reason one must read the sections in the order of presentation. On the other hand the rules created by a transformation will be given between brackets.

Conventions It is sometimes hard to find a pertinent identifier for a newly created rule, suggesting its semantics. In this situation the rule will be given the concatenation of the calling rule name (maybe shortened) with a prefix and/or suffix, often numbered. In the examples contained in this document, we'll adopt moreover the following lexicographical conventions:

- The empty word is denoted ε.
- A latin lower case always denotes a terminal.
- A latin upper case always denotes a nonterminal.
- A greek lower case always denotes any concatenation of terminals and nonterminals.
- We'll note "A $\Longrightarrow \alpha$" the relation "A produces α".

1.3 Rational operators

The following rational operators were added to the BNF, in order to gain in compaction and readability (it is not a gain of expressivity): α^*, α^+, $[\alpha]$, $\{$ A a $\ldots\}^*$, $\{$ A a $\ldots\}^+$. Any occurrence of these expressions can be replaced by a nonterminal which defining rule is specific. As we'll see in the section devoted to the LL(1) property checking (cf. 4), we are even allowed to consider that there is no sharing between these rules. Here is the array giving their semantics.

Notation	Definition	Validity constraint
$\mathrm{X} \rightarrow \alpha^*$	$\mathrm{X} \rightarrow \alpha\ \mathrm{X} \mid \varepsilon$	$\neg(\alpha \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \rightarrow \alpha^+$	$\mathrm{X} \rightarrow \alpha\alpha^*$	$\neg(\alpha \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \rightarrow [\alpha]$	$\mathrm{X} \rightarrow \alpha \mid \varepsilon$	$\neg(\alpha \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \rightarrow \{\mathrm{A\ a} \ldots\}^*$	$\mathrm{X} \rightarrow \varepsilon \mid \mathrm{A\ (a\ A)}^*$	$\neg(\mathrm{A} \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \rightarrow \{\mathrm{A\ a} \ldots\}^+$	$\mathrm{X} \rightarrow \mathrm{A\ (a\ A)}^*$	$\neg(\mathrm{A} \overset{*}{\Longrightarrow} \varepsilon)$
$\mathrm{X} \rightarrow \{[\mathrm{A}]\ \mathrm{a} \ldots\}$	$\mathrm{X} \rightarrow \varepsilon \mid \mathrm{A\ (a\ [A])}^* \mid \mathrm{(a\ [A])}^+$	$\neg(\mathrm{A} \overset{*}{\Longrightarrow} \varepsilon)$

The elected definitions are LL(1), and the validity constraints allow a direct definition of associated parsers in Caml Light (suitable argument types).

2 Grammar transformations

The transformations are merely the application of basic properties of rational expressions, expressed in a slightly different way in the frame of grammars.

Factorisation The factorisation forms prefixes and/or suffixes of a production subset of a rule.

Prefix $X \rightarrow \beta\gamma_1 \mid \beta\gamma_2 \mid \ldots \mid \beta\gamma_n \mid B$ *becomes* $\begin{array}{lcl} X & \rightarrow & \beta\ Y \mid B \\ Y & \rightarrow & \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n \end{array}$

Suffix $X \rightarrow \alpha_1\beta \mid \alpha_2\beta \mid \ldots \mid \alpha_n\beta \mid B$ *becomes* $\begin{array}{lcl} X & \rightarrow & Y\ \beta \mid B \\ Y & \rightarrow & \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \end{array}$

Bifix $X \rightarrow \alpha\beta_1\gamma \mid \alpha\beta_2\gamma \mid \ldots \mid \alpha\beta_n\gamma \mid B$ *becomes* $\begin{array}{lcl} X & \rightarrow & \alpha\ Y\ \gamma \mid B \\ Y & \rightarrow & \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{array}$

We can apply these factorisations if $n = 1$, or partially on the factorizable productions.

Reduction The reduction gather productions of the same rule and create a new nonterminal:

$$X \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad \textit{becomes} \quad \begin{array}{lcl} X & \rightarrow & Y \mid \alpha_{i+1} \mid \alpha_{i+2} \mid \ldots \mid \alpha_n \\ Y & \rightarrow & \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_i \end{array}$$

Elimination Each useless rule (typically after a global expansion — see below) may be suppressed. We can also remove redundant productions inside a rule, and just keep one.

Remark 1 A global or mixed entry rule now useless inside a section will be moved and therefore disappear — but is *not* removed from the whole grammar!

Remark 2 The following transformation is *not* an elimination:
$X \rightarrow X \mid A$ *becomes* $X \rightarrow A$
Cf. paragraph about the Arden lemma (2.1).

Remark 3 the following elimination is legal:
$X \rightarrow \texttt{lower}_{val} \mid \texttt{lower}_{id}$ *becomes* $X \rightarrow \texttt{lower}$

Renaming For sake of readability identifiers may be globally renamed in all the grammar. We'll try to avoid such an operation choosing with care from the beginning suitable names (to avoid clashes).

Expansion An expansion can be *total*, *partial*, *prefix*, *suffix* or *global*.

Total A total expansion is a textual substitution of a whole rule right-hand at *a particular* occurrence of the corresponding nonterminal. For example:

$$\begin{array}{lcl} \mathrm{A} & \rightarrow & \mathrm{a\ B\ C} \\ & | & \mathrm{B} \\ \mathrm{B} & \rightarrow & \mathrm{b\ A} \\ & | & \mathrm{b} \\ \mathrm{C} & \rightarrow & \mathrm{c} \end{array} \quad \textit{becomes} \quad \begin{array}{lcl} \mathrm{A} & \rightarrow & \mathrm{a\ b\ A\ C} \\ & | & \mathrm{a\ b\ C} \\ & | & \mathrm{B} \\ \mathrm{B} & \rightarrow & \mathrm{b\ A} \\ & | & \mathrm{b} \\ \mathrm{C} & \rightarrow & \mathrm{c} \end{array}$$

It will be the default expansion in the following, in absence of indication.

Global A global expansion is the **total** expansion of a rule for all possible occurrences of its corresponding nonterminal in the whole grammar (this rule can therefore be eliminated). The transformations presentation being modular, the reader might be disconcerted by this global scope. Actually a global expansion will be equivalent to a total expansion applied to the whole subsection, followed by an elimination; else a note will be supplied.

Partial A partial expansion is the composition of a **global** expansion, a partial bifix factorisation and a renaming. To sum up:

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

Prefix A prefix expansion is the composition of the **global** expansion of a left-factorisable rule, a bifix factorisation and a renaming. Briefly:

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

Suffix A suffix expansion is the composition of the **global** expansion of a right-factorizable rule, a bifix factorisation and a renaming. That is to say:

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

Option The option (transformation) is a particular case of the **partial** expansion of an empty production, followed by a "square bracketing". For instance:

$$\begin{array}{lcl} \mathrm{A} & \rightarrow & \mathrm{a\ B\ C} \\ & | & \mathrm{B} \\ \mathrm{B} & \rightarrow & \mathrm{b\ A} \\ & | & \varepsilon \\ \mathrm{C} & \rightarrow & \mathrm{c} \end{array} \quad \textit{becomes} \quad \begin{array}{lcl} \mathrm{A} & \rightarrow & \mathrm{a\ [B]\ C} \\ & | & \mathrm{[B]} \\ \mathrm{B} & \rightarrow & \mathrm{b\ A} \\ \mathrm{C} & \rightarrow & \mathrm{c} \end{array}$$

The same we'll assimilate to an option (for short) the transformation:

$$
\begin{array}{lcl}
\mathrm{X} & \rightarrow & \mathrm{A} \\
& | & \mathrm{b} \\
\mathrm{A} & \rightarrow & \{\ \mathrm{B}\ \text{“,”}\ \ldots\}^* \\
& | & [\mathrm{a\ B\ c}] \\
& | & \mathrm{a}
\end{array}
\quad \textit{becomes} \quad
\begin{array}{lcl}
\mathrm{X} & \rightarrow & [\mathrm{A}] \\
& | & \mathrm{b} \\
\mathrm{A} & \rightarrow & \{\ \mathrm{B}\ \text{“,”}\ \ldots\}^+ \\
& | & \mathrm{a\ B\ c} \\
& | & \mathrm{a}
\end{array}
$$

In order to ease the LL(1) checking of the resulting grammar, we'll apply the option each time it will be possible. Thus the resulting form will not contain *explicit* empty productions (Beware: we do *not* remove these productions!). In other words, no production will produce explicitly ε.

Note that the converses of factorisations, reductions, expansions and options are also legal transformations.

2.1 The Arden lemma

Suppose we have a rational equation of languages of the form $\mathrm{X} = \alpha\ \mathrm{X} + \beta$, with $\varepsilon \notin \alpha$, and where $+$ denotes disjunction ($|$). Then the Arden lemma states that $\mathrm{X} = \alpha^*\beta$ is the sole solution of the equation. If $\varepsilon \in \alpha$, then $\mathrm{X} = \alpha^*(\beta + \gamma)$ is a solution, for all γ (which can even be a context-dependent language). We thus have an infinity of solutions, and we'll choose in this case the lowest solution language ("the minimal fix-point"): $\mathrm{X} = \alpha^*\beta$. For short, we'll always allow the transformation: $\mathrm{X} \rightarrow \alpha\ \mathrm{X} \mid \beta$ *becomes* $\mathrm{X} \rightarrow \alpha^*\beta$.
We so have in particular: $\mathrm{X} \rightarrow \mathrm{X} \mid \beta$ *becomes* $\mathrm{X} \rightarrow \beta$. We'll call "arden" all the elementary transformations that imply the application of the Arden lemma:

$\mathrm{X} \rightarrow \alpha\ \mathrm{X} \mid \varepsilon$ *or* $\mathrm{X} \rightarrow \mathrm{X}\ \alpha \mid \varepsilon$	*becomes*	$\mathrm{X} \rightarrow \alpha^*$
$\mathrm{X} \rightarrow \alpha\ \mathrm{X} \mid \alpha$ *or* $\mathrm{X} \rightarrow \mathrm{X}\ \alpha \mid \alpha$	*becomes*	$\mathrm{X} \rightarrow \alpha^+$
$\mathrm{X} \rightarrow \mathrm{A} \mid \mathrm{A\ a\ X}$	*becomes*	$\mathrm{X} \rightarrow \{\mathrm{A\ a} \ldots\}^+$
$\mathrm{X} \rightarrow \varepsilon \mid \mathrm{A} \mid \mathrm{A\ a\ X}$	*becomes*	$\mathrm{X} \rightarrow \{\mathrm{A\ a} \ldots\}^*$

Remark 1 We get: $\mathrm{X} \rightarrow \mathrm{X}\ \alpha \mid \beta$ *becomes* $\mathrm{X} \rightarrow \beta\alpha^*$.

Remark 2 The Arden lemma allows us to find non trivial equalities, as for example: $(a + cb^*d)^* = a^* + a^*c(b + da^*c)^*da^*$.

2.2 Syntactic ambiguities

The question of grammar ambiguity is always hard. A grammar is said to be ambiguous if and only if one can build two different derivations for a word of the language. A sufficient condition of ambiguity is the double-recursion of a rule.
When double-recursion is inside a production, we can sometimes solve this problem applying the Arden lemma. A typical case is arithmetic expressions grammars, where some operators are both binary and infix: $\mathrm{E} \rightarrow \mathrm{E}$ "+" $\mathrm{E} \mid \mathrm{E}$ "$*$" $\mathrm{E} \mid$ "(" E ")" $\mid$ **id**.
If double-recursion is left-side and right-side, on *two* productions, we can proceed as it follows. Assume:

$\mathrm{Z} \rightarrow \mathrm{A\ Z} \mid \mathrm{Z\ B} \mid \mathrm{C}$

The Arden lemma implies here: $\mathrm{Z} \rightarrow \mathrm{A\ Z\ B}^* \mid \mathrm{C\ B}^*$
But it is trivial that: $\mathrm{X} \rightarrow \alpha\ \mathrm{X}\ \beta \mid \gamma$ *becomes* $\forall n \geq 0.\mathrm{X} \rightarrow \alpha^n \gamma \beta^n$
Thus: $\forall n \geq 0.\mathrm{Z} \rightarrow \mathrm{A}^n\ (\mathrm{C\ B}^*)\ (\mathrm{B}^*)^n$ *simplified in* $\mathrm{Z} \rightarrow \mathrm{A}^*\ \mathrm{C\ B}^*$
Rewriting $\mathrm{Z} \rightarrow \mathrm{A}^+\ \mathrm{C\ B}^* \mid \mathrm{C\ B}^*$ *it follows* $\mathrm{Z} \rightarrow \mathrm{A\ Z} \mid \mathrm{C\ B}^*$
To conclude, we'll keep the following transformation:

$\boxed{\mathrm{Z} \rightarrow \mathrm{A\ Z} \mid \mathrm{Z\ B} \mid \mathrm{C} \quad \textit{becomes} \quad \mathrm{Z} \rightarrow \mathrm{A\ Z} \mid \mathrm{C\ B}^*}$

3 Transforming the ASN.1:1990 grammar

3.1 Modules

3.1.1 Step 0

We present first the section corresponding to the ASN.1 modules specification. We introduce rule gathering.

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	$\rightarrow$	Exports Imports AssignmentList
	$\mid$	ε
Exports	$\rightarrow$	EXPORTS SymbolsExported ";"
	$\mid$	ε
Imports	$\rightarrow$	IMPORTS SymbolsImported ";"
	$\mid$	ε
SymbolsExported	$\rightarrow$	SymbolList
	$\mid$	ε
SymbolsImported	$\rightarrow$	SymbolsFromModuleList
	$\mid$	ε
SymbolList	$\rightarrow$	Symbol
	$\mid$	Symbol "," SymbolList
SymbolsFromModuleList	$\rightarrow$	SymbolsFromModule
	$\mid$	SymbolsFromModuleList SymbolsFromModule
Symbol	$\rightarrow$	`typereference`
	$\mid$	`valuereference`
SymbolsFromModule	$\rightarrow$	SymbolList FROM ModuleIdentifier

AssignmentList	$\rightarrow$	Assignment
	$\mid$	AssignmentList Assignment
Assignment	$\rightarrow$	TypeAssignment
	$\mid$	ValueAssignment
TypeAssignment	$\rightarrow$	`typereference` "::=" Type
ValueAssignment	$\rightarrow$	`valuereference` Type "::=" Value

<u>DefinedType</u>	$\rightarrow$	Externaltypereference
	$\mid$	`typereference`
<u>*DefinedValue*</u>	$\rightarrow$	Externalvaluereference
	$\mid$	`valuereference`
Externaltypereference	$\rightarrow$	`modulereference` "." `typereference`
Externalvaluereference	$\rightarrow$	`modulereference` "." `valuereference`

3.1.2 Step 1

We substitute (lexically) ambiguous token identifiers, trying when possible to keep information on their original "semantics" (which then appears in subscript). On the other hand we apply option until all ε disappear, and we use the Arden lemma to introduce rational operators.

ModuleDefinition	$\rightarrow$	ModuleIdentifier DEFINITIONS [TagDefault TAGS] "::=" BEGIN [ModuleBody] END

Cf. 'TagDefault' and 'ModuleBody'.

ModuleIdentifier	$\rightarrow$	`upper`$_{mod}$ ["{" ObjIdComponent$^+$ "}"]
ObjIdComponent	$\rightarrow$	`lower`$_{id}$
	|	NumberForm
	|	`lower`$_{id}$ "(" NumberForm ")"
NumberForm	$\rightarrow$	`number`
	|	DefinedValue

Option and then global expansion of 'AssignedIdentifier'.
Arden of 'ObjIdComponentList'.
Global expansion of 'NameForm'.
Global expansion of 'NameAndNumberForm'.
Elimination of the second production of 'ObjectIdentifierValue' because:
ObjIdComponentList $\Rightarrow$ ObjIdComponent ObjIdComponentList
$\Rightarrow$ NumberForm ObjIdComponentList $\Rightarrow$ DefinedValue ObjIdComponentList.
Global expansion of 'ObjIdComponentList'.
Global expansion of '*ObjectIdentifierValue*'.
Beware! We keep ObjectIdentifierValue $\rightarrow$ "{" ObjIdComponent$^+$ "}"
in the section (3.3).

TagDefault	$\rightarrow$	EXPLICIT
	$\mid$	IMPLICIT

Option and then suffix expansion of 'TagDefault'.

ModuleBody	$\rightarrow$	[Exports] [Imports] AssignmentList
Exports	$\rightarrow$	EXPORTS [SymbolList] ";"
Imports	$\rightarrow$	IMPORTS [SymbolsFromModuleList] ";"
SymbolList	$\rightarrow$	$\{$Symbol "," $\ldots\}^+$
SymbolsFromModuleList	$\rightarrow$	SymbolsFromModule$^+$
Symbol	$\rightarrow$	`typereference`
	$\mid$	`valuereference`
SymbolsFromModule	$\rightarrow$	SymbolList FROM ModuleIdentifier

Option of 'Exports' and of 'Imports'.
Option and then global expansion of 'SymbolsExported'.
Option and then global expansion of 'SymbolsImported'.
Arden of 'SymbolList' and 'SymbolsFromModuleList'.

AssignmentList	$\rightarrow$	Assignment$^+$
Assignment	$\rightarrow$	$\texttt{upper}_{typ}$ "::=" Type
	$\mid$	$\texttt{lower}_{val}$ Type "::=" Value

Arden of '*AssignmentList*'.
Global expansion of 'TypeAssignment' and 'ValueAssignment'.

DefinedType	$\rightarrow$	$\texttt{upper}_{mod}$ "." $\texttt{upper}_{typ}$
	$\mid$	$\texttt{upper}_{typ}$
DefinedValue	$\rightarrow$	$\texttt{upper}_{mod}$ "." $\texttt{lower}_{val}$
	$\mid$	$\texttt{lower}_{val}$

Global expansion of 'ExternalTypeReference' and 'ExternalValueReference'.

3.1.3 Step 2

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

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

> Total expansion of *'DefinedValue'*.
> Prefix factorisation (lower).
> Elimination of 'NumberForm' because '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`$_{val}$

> Global expansion of *'AssignmentList'*.
> Global expansion of 'SymbolList'.
> Global expansion of 'SymbolsFromModuleList'.

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

> 'DefinedType' is moved to the section (3.2).
> *'DefinedValue'* is moved to the sections 3.2 and (3.3).

3.2 Types

3.2.1 Step 0

We first give the standard notation devoted to ASN.1 types specification, after restructuring. Notice we recover the rules *'DefinedType'* and *'DefinedValue'* from the section (3.1). *'Subtype'* and *'ParentType'* rules are showed in this section, and not in the section 3.4, in order to facilitate reading. *'SubtypeSpec'* is presented in the section (3.4). Notice also that *'ClassNumber'* is a mixed entry rule (Cf.3.1.3).

Type	→	DefinedType
	|	BuiltInType
	|	Subtype

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

> Prefix factorisation of *'DefinedType'*.
> Suffix factorisation of *'DefinedValue'*.

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

	⊏	
	|	RealType

NamedType	→	`identifier` Type
	|	Type
	|	SelectionType

Subtype	→	ParentType SubtypeSpec
	|	SET SizeConstraint OF Type
	|	SEQUENCE SizeConstraint OF Type
ParentType	→	Type
<u>SizeConstraint</u>	→	SIZE SubtypeSpec

IntegerType	→	INTEGER
	|	INTEGER "{" NamedNumberList "}"
NamedNumberList	→	NamedNumber
	|	NamedNumberList "," NamedNumber
NamedNumber	→	`identifier` "(" SignedNumber ")"
	|	`identifier` "(" DefinedValue ")"
<u>SignedNumber</u>	→	["-"] `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 Step 1

NamedType → $\texttt{lower}_{id}$ Type
| Type

> Elimination of the production 'SelectionType' because:
> Type ⇒ BuiltInType ⇒ SelectionType

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

> Total expansion of *'DefinedType'*.
> Partial expansion of productions *'SetType'*, *'SequenceType'*, *'SetOfType'*, *'SequenceOfType'*, *'NullType'*, *'SelectionType'* and *'TaggedType'* of the rule *'BuiltInType'*.
> Global expansion of '*Subtype*' after the global expansion of 'ParentType':
> Subtype ⇒ ParentType SubtypeSpec ⇒ Type SubtypeSpec
> the goal is to left-factorise the rule *'Type'* and to make clear the double-recursion, source of ambiguity.
> Global expansion of 'SizeConstraint' which is moved to the section (3.4).

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

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

> Global expansion of *'BooleanType'*, *'IntegerType'*, *'BitStringType'*, *'OctetStringType'*, *'TaggedType'*, *'AnyType'*, *'ObjectIdentifierType'*, *'UsefulType'*, *'CharacterStringType'*, *'RealType'*.
> Arden of 'Enumeration' and global expansion, and then global expansion of *'EnumeratedType'*.
> Arden of 'AlternativeTypeList' and global expansion, and then global expansion of *'ChoiceType'*.
> Cf. *'Type'*, *'NamedNumber'*, *'NamedBit'*.

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

> Arden of 'NamedNumberList' and global expansion.
> Prefix factorisation of *'IntegerType'* and then global expansion.
> Bifix factorisation of 'NamedNumber' ('AuxNamedNum').

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

> Bifix factorisation: we recognize the rule 'ClassNumber'.

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

> Prefix factorisations.

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

> Bifix factorisation of *'SequenceType'* and *'SetType'*.
> Arden of 'ElementTypeList' and global expansion.
> Prefix factorisation of 'ElementType'.

SelectionType → `lower`$_{id}$ "<" Type

NullType → NULL

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

> Bifix factorisation of *'TaggedType'*: we recognize the option of the rule *'TagDefault'* (Cf. 3.1).
> Global expansion of 'Tag'.
> Option of 'Class'.

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

> Total expansion of *'DefinedValue'*.

3.2.3 Step 2

NamedType	$\rightarrow$	$\mathtt{lower}_{id}$ Type
	|	Type

<u>*Type*</u>	$\rightarrow$	upper [“.” $\mathtt{upper}_{typ}$]
	|	BuiltInType
	|	SET “{” {ElementType “,” ...}* “}”
	|	SEQUENCE “{” {ElementType “,” ...}* “}”
	|	SET [OF Type]
	|	SEQUENCE [OF Type]
	|	$\mathtt{lower}_{id}$ “<” Type
	|	“[” [Class] ClassNumber “]” [TagDefault] Type
	|	NULL
	|	Type SubtypeSpec
	|	SET SIZE SubtypeSpec OF Type
	|	SEQUENCE SIZE SubtypeSpec OF Type

> Global expansion of *‘SetType’*, *‘SequenceType’*, *‘SetOfType’*, *‘SequenceOfType’*, *‘SelectionType’*, *‘TaggedType’* and *‘NullType’*.

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

$\square$

	⊏	
	|	EXTERNAL
	|	“UTCTime”
	|	“GeneralizedTime”
	|	“ObjectDescriptor”

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

> Total expansion of ‘SignedNumber’ and *‘DefinedValue’* in ‘AuxNamedNum’.
> ‘SignedNumber’ and *‘DefinedValue’* moved to the section (3.3).

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

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

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

3.2.4 Step 3

We get rid here of the *'Type'* rule ambiguity, applying the transformation of the section (2.2). Then we'll make LL(1) the rule *'NamedType'*.

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

Suffix factorisations ('SetSeq').

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

Application of section (2.2) transformation.

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

Prefix factorisation (‘TypeSuf’).

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

Option of ‘TypeSuf’.
Reduction of ‘*Type*’ (‘AuxType’). We'll understand why at the fourth step of section (3.3).

$$\begin{array}{lcl} \textit{NamedType} & \rightarrow & \texttt{lower}_{id}\ \text{Type} \\ & | & \text{Type} \end{array}$$

$$\begin{array}{lcl} \textit{NamedType} & \rightarrow & \texttt{lower}_{id}\ \text{Type} \\ & | & \texttt{lower}_{id}\ \text{``<''}\ \text{Type} \\ & | & \texttt{upper}\ [\text{``.''}\ \texttt{upper}_{typ}]\ \text{SubtypeSpec}^{*} \\ & | & \text{NULL SubtypeSpec}^{*} \\ & | & \text{AuxType} \end{array}$$

Total expansion of *'Type'*.

$$\begin{array}{lcl} \textit{NamedType} & \rightarrow & \texttt{lower}_{id}\ [\text{``<''}]\ \text{Type} \\ & | & \texttt{upper}\ [\text{``.''}\ \texttt{upper}_{typ}]\ \text{SubtypeSpec}^{*} \\ & | & \text{NULL SubtypeSpec}^{*} \\ & | & \text{AuxType} \end{array}$$

Bifix factorisation.

3.2.5 Step 4

Here's the resulting grammar of the section (3.2).

$$
\begin{array}{lcl}
\underline{\mathit{Type}} & \rightarrow & \texttt{lower}_{id}\ \text{“<” Type} \\
 & | & \texttt{upper}\ [\text{“.”}\ \texttt{upper}_{typ}]\ \text{SubtypeSpec}^{*} \\
 & | & \text{NULL SubtypeSpec}^{*} \\
 & | & \text{AuxType} \\
\text{AuxType} & \rightarrow & \text{“[” [Class] ClassNumber “]” [TagDefault] Type} \\
 & | & \text{BuiltInType SubtypeSpec}^{*} \\
 & | & \text{SetSeq [TypeSuf]} \\
\text{SetSeq} & \rightarrow & \text{SET} \\
 & | & \text{SEQUENCE} \\
\text{TypeSuf} & \rightarrow & \text{SubtypeSpec}^{+} \\
 & | & \text{“\{” \{ElementType “,” \ldots\}}^{*}\ \text{“\}” SubtypeSpec}^{*} \\
 & | & \text{[SIZE SubtypeSpec] OF Type}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{BuiltInType} & \rightarrow & \text{BOOLEAN} \\
 & | & \text{INTEGER [“\{” \{NamedNumber “,” \ldots\}}^{+}\ \text{“\}”]} \\
 & | & \text{BIT STRING [“\{” \{NamedBit “,” \ldots\}}^{+}\ \text{“\}”]} \\
 & | & \text{OCTET STRING} \\
 & | & \text{CHOICE “\{” \{NamedType “,” \ldots\}}^{+}\ \text{“\}”} \\
 & | & \text{ANY [DEFINED BY}\ \texttt{lower}_{id}] \\
 & | & \text{OBJECT IDENTIFIER} \\
 & | & \text{ENUMERATED “\{” \{NamedNumber “,” \ldots\}}^{+}\ \text{“\}”} \\
 & | & \text{REAL} \\
 & | & \text{“NumericString”} \\
 & | & \text{“PrintableString”} \\
 & | & \text{“TeletexString”} \\
 & | & \text{“T61String”} \\
 & | & \text{“VideotexString”} \\
 & | & \text{“VisibleString”} \\
 & | & \text{“ISO646String”} \\
 & | & \text{“IA5String”} \\
 & | & \text{“GraphicString”} \\
 & | & \text{“GeneralString”}
\end{array}
$$

$\sqsubset$

	$\sqsubset$	
	|	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
<u>*ClassNumber*</u>	→	`number`
	|	[`upper`$_{mod}$ “.”] `lower`$_{val}$

3.3 Values

3.3.1 Step 0

Here is the standard notation of the ASN.1 values grammar. Notice we recover the rules *'ObjectIdentifierValue'*, coming from the section 3.1, and *'DefinedValue'* and *'SignedNumber'*, from the 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

And not: *ChoiceValue* → NamedValue (**Technical corrigendum**).

SelectionValue	→	Value

And not: *SelectionValue* → NamedValue (**Technical corrigendum**).

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

TaggedValue	→	Value

AnyValue	→	Type ":" Value

And not: *AnyValue* → Type Value (**Technical corrigendum**).

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

CharacterStringValue	→	`cstring`

3.3.2 Step 1

We'll take care that the tokens `bstring` and `hstring` are merged to `basednumber`. Cf. (6.2). On the other hand, the productions in double are not immediately merged for sake of clarity. Notice that the terminals "0", "2" and "10" become the token `number`. (It's just what does a lexical analyser.)

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

> Partial expansion of *'IntegerValue'*, *'NullValue'*, *'ChoiceValue'*, *'SelectionValue'*, *'TaggedValue'*, *'AnyValue'* and *'EnumeratedValue'* of the rule *'BuiltInValue'*.
> Global expansion of *'DefinedValue'*.

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

Global expansion of *'BooleanValue'*, *'ObjectIdentifierValue'*, *'CharacterStringValue'* and *'OctetStringValue'*.
Arden of 'IdentifierList' and global expansion, and then bifix factorisation in *'BitStringValue'*.
Global expansion of *'BitStringValue'*.
Arden of 'ElementValueList' and global expansion, and then bifix factorisation in *'SequenceValue'* and *'SetValue'*.
Global expansion of *'SequenceValue'* and *'SetValue'*.
Arden of 'ValueList' and global expansion, and then bifix factorisation in *'SequenceOfValue'* and *'SetOfValue'*.
Global expansion of *'SequenceOfValue'* and *'SetOfValue'*.
Subsection *'RealValue'*: total expansion of *'SignedNumber'* in 'Mantissa' and 'Exponent'.
Global expansion of 'Mantissa', 'Base' (Base → number) and 'Exponent'.
Global expansion of 'NumericRealValue', 'SpecialRealValue' and *'RealValue'*.

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

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

Global expansion of *'SignedNumber'*.

EnumeratedValue	→	$\texttt{lower}_{id}$
NullValue	→	NULL
ChoiceValue	→	[$\texttt{lower}_{id}$ ":"] Value
SelectionValue	→	Value
TaggedValue	→	Value
AnyValue	→	Type ":" Value

3.3.3 Step 2

Value	$\rightarrow$	BuiltInValue
	$\mid$	[“-”] `number`
	$\mid$	$\texttt{lower}_{id}$
	$\mid$	NULL
	$\mid$	[$\texttt{lower}_{id}$ “:”] Value
	$\mid$	Value
	$\mid$	Value
	$\mid$	Type “:” Value
	$\mid$	$\texttt{lower}_{id}$
	$\mid$	[$\texttt{upper}_{mod}$ “.”] $\texttt{lower}_{val}$
	$\mid$	`number`

> Global expansion of *‘IntegerValue’*, *‘NullValue’*, *‘ChoiceValue’*, *‘SelectionValue’*, *‘TaggedValue’*, *‘AnyValue’*, and *‘EnumeratedValue’*.
> Partial expansion of *BuiltInValue* $\rightarrow$ number

BuiltInValue	$\rightarrow$	TRUE
	$\mid$	FALSE
	$\mid$	PLUS-INFINITY
	$\mid$	MINUS-INFINITY
	$\mid$	`basednum`
	$\mid$	`string`
	$\mid$	“{” BetBraces “}”
BetBraces	$\rightarrow$	{NamedValue “,” ...}*
	$\mid$	$\text{ObjIdComponent}^{+}$
	$\mid$	[“-”] `number` “,” `number` “,” [“-”] `number`

> Arden lemma (Elimination of redundant Value $\rightarrow$ Value $\mid$ ...).
> Elimination of the production “{” {lower_{id} “,” ...}* “}”
> because “{” {Value “,” ...}* “}” $\Rightarrow$ “{” {lower_{id} “,” ...}* “}”
> Elimination of the production “{”{Value “,” ...}* “}”
> because “{” {NamedValue “,” ...}* “}” $\Rightarrow$ “{”{Value “,” ...}* “}”
> Bifix factorisation (‘BetBraces’).

NamedValue	$\rightarrow$	$\texttt{lower}_{id}$ Value
	$\mid$	Value

3.3.4 Step 3

<u>*Value*</u>	→	BuiltInValue
	|	[“-”] `number`
	|	NULL
	|	`lower` [“:” Value]
	|	Type “:” Value
	|	$\texttt{upper}_{mod}$ “.” $\texttt{lower}_{val}$

> Elimination of redundancies.
> Arden of ‘<u>*Value*</u>’.
> Prefix factorisation of lower.

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

> Option of ‘BetBraces’.
> Elimination of the third production of ‘BetBraces’ because:
> {NamedValue “,” …}$^+$ $\stackrel{+}{\Longrightarrow}$ [“-”] number “,” number “,” [“-”] number

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

3.3.5 Step 4

We make LL(1) the rule *'<u>Value</u>'*, and then *'NamedValue'*.

<u>Value</u>	$\rightarrow$	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

Total expansion of *'Type'*.

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

Prefix factorisations.
Reduction ('AuxVal0').
The reason is given in (3.3.6).

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

Converse expansion ('SpecVal').
Prefix factorisation of 'AuxVal1' ('AuxVal11').

NamedValue	→	`lower`$_{id}$ Value
	|	AuxVal0
	|	`upper` AuxVal1
	|	`lower` [AuxVal2]
	|	["-"] `number`

Expansion of '*Value*'.

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

Prefix factorisation.

3.3.6 Step 5

We now make LL(1) the rule 'BetBraces'.

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

Rational operations.

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

Converse expansion ('AuxNamed').
Total expansion of '*NamedValue*'.
Total expansion of '*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]
	|	"." $\texttt{lower}_{val}$ ObjIdComponent*
AuxBet3	→	ObjIdComponent$^+$
	|	AuxNamed

Prefix factorisations and development of 'AuxBet1'.

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

Rational operation in ‘AuxBet1’.
Total expansion of ‘NamedValSuf’.
Total expansion of ‘AuxVal1’ in ‘AuxBet2’, and then of ‘AuxVal11’.

AuxBet1	$\rightarrow$	“(” 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	$\rightarrow$	SpecVal [AuxNamed]
	|	“.” AuxBet21
AuxBet21	$\rightarrow$	$\texttt{upper}_{typ}$ SpecVal [AuxNamed]
	|	$\texttt{lower}_{val}$ [AuxBet3]

Total expansion of ‘ObjectIdComponent’.
Total expansion of ‘*Value*’.
Prefix factorisation of ‘AuxBet2’.
We recognize ‘AuxBet3’.

AuxBet1	$\rightarrow$	"(" ClassNumber ")" ObjIdComponent*
	|	AuxNamed
	|	AuxVal2 [AuxNamed]
	|	"-" `number` [AuxNamed]
	|	AuxVal0 [AuxNamed]
	|	`lower` [AuxBet11]
	|	`number` [AuxBet3]
	|	`upper` AuxBet2
AuxBet11	$\rightarrow$	"(" ClassNumber ")" ObjIdComponent*
	|	ObjIdComponent+
	|	AuxVal2 [AuxNamed]
	|	AuxNamed

Prefix factorisations in 'AuxBet1' ('AuxBet11') where we recognize 'AuxBet3' and 'AuxBet2'.
We develop 'AuxBet11'.
NOTA: We did not put here the rules 'AuxBet2' and 'AuxBet21'.

3.3.7 Step 6

We recall the result of previous transformations.

Value	$\rightarrow$	AuxVal0
	$\mid$	`upper` AuxVal1
	$\mid$	`lower` [AuxVal2]
	$\mid$	["-"] `number`
AuxVal0	$\rightarrow$	BuiltInValue
	$\mid$	AuxType ":" Value
	$\mid$	NULL [SpecVal]
AuxVal1	$\rightarrow$	SpecVal
	$\mid$	"." AuxVal11
AuxVal2	$\rightarrow$	["<" Type] ":" Value
AuxVal11	$\rightarrow$	$\texttt{upper}_{typ}$ SpecVal
	$\mid$	$\texttt{lower}_{val}$
SpecVal	$\rightarrow$	SubtypeSpec* ":" Value
BuiltInValue	$\rightarrow$	TRUE
	$\mid$	FALSE
	$\mid$	PLUS-INFINITY
	$\mid$	MINUS-INFINITY
	$\mid$	`basednum`
	$\mid$	`string`
	$\mid$	"{" [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

3.4 Subtypes

3.4.1 Step 0

We present here the standard notation for ASN.1 subtypes grammar, after restructuring. Notice we recover the rule ‘SizeConstraint’, coming from the 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	$\rightarrow$	WITH COMPONENT SingleTypeConstraint
	$\mid$	WITH COMPONENTS MultipleTypeConstraints
SingleTypeConstraint	$\rightarrow$	SubtypeSpec
MultipleTypeConstraints	$\rightarrow$	FullSpecification
	$\mid$	PartialSpecification
FullSpecification	$\rightarrow$	"{" TypeConstraints "}"
PartialSpecification	$\rightarrow$	"{" "..." "," TypeConstraints "}"
TypeConstraints	$\rightarrow$	NamedConstraint
	$\mid$	NamedConstraint "," TypeConstraints
NamedConstraint	$\rightarrow$	`identifier` Constraint
	$\mid$	Constraint
Constraint	$\rightarrow$	ValueConstraint PresenceConstraint
ValueConstraint	$\rightarrow$	SubtypeSpec
	$\mid$	ε
PresenceConstraint	$\rightarrow$	PRESENT
	$\mid$	ABSENT
	$\mid$	OPTIONAL
	$\mid$	ε

3.4.2 Step 1

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

Arden of ‘SubtypeValueSetList’, and then global expansion.

SingleValue → Value

ContainedSubtype → INCLUDES Type

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

Prefix factorisation of ‘LowerEndPoint’.
Suffix factorisation of ‘UpperEndPoint’.
Global expansion of ‘LowerEndPoint’ and ‘UpperEndPoint’.

SizeConstraint → SIZE SubtypeSpec

PermittedAlphabet → FROM SubtypeSpec

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

Global expansion of 'SingleTypeConstraint'.
Prefix factorisation of *'InnerTypeConstraints'* ('InnerTypeSuf').
Global expansion of 'FullSpecification' and 'PartialSpecification'.
Bifix factorisation of 'MultipleTypeConstraints'.
Arden of 'TypeConstraints'.
Suffix factorisation of 'NamedConstraint'.
Option of 'ValueConstraint', and then global expansion.
Option of 'PresenceConstraint'.

3.4.3 Step 2

<u>*SubtypeSpec*</u> $\rightarrow$ “(” {SubtypeValueSet “|” ...}$^+$ “)”
SubtypeValueSet $\rightarrow$ Value
$\mid$ INCLUDES Type
$\mid$ LowerEndValue [“<”] “..” [“<”] UpperEndValue
$\mid$ FROM SubtypeSpec
$\mid$ SIZE SubtypeSpec
$\mid$ WITH InnerTypeSuf
LowerEndValue $\rightarrow$ Value
$\mid$ MIN
UpperEndValue $\rightarrow$ Value
$\mid$ MAX
InnerTypeSuf $\rightarrow$ COMPONENT SubtypeSpec
$\mid$ COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints $\rightarrow$ “{” [“...” “,”] {[NamedConstraint] “,” ...} “}”
NamedConstraint $\rightarrow$ $\texttt{lower}_{id}$ [SubtypeSpec] [PresenceConstraint]
$\mid$ SubtypeSpec [PresenceConstraint]
$\mid$ PresenceConstraint
PresenceConstraint $\rightarrow$ PRESENT
$\mid$ ABSENT
$\mid$ OPTIONAL

Global expansion of ‘SingleValue’.
Global expansion of ‘ContainedSubtype’.
Global expansion of ‘ValueRange’.
Global expansion of ‘PermittedAlphabet’.
Global expansion of ‘SizeConstraint’.
Global expansion of ‘InnerTypeConstraints’.
Global expansion of ‘TypeConstraints’.
Global expansion of ‘Constraint’ and then option of ‘NamedConstraint’.

3.4.4 Step 3

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

Global expansion of ‘LowerEndValue’.
Prefix factorisation of ‘SubtypeValueSet’ (‘SubValSetSuf’).

3.4.5 Step 4

A subtle problem arises, impeding the grammar to be LL(1). We indeed previously obtain (Cf. 3.3):

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

The third production of *‘Value’* implies that the terminal “<” must not be a possible symbol after an occurrence of *‘Value’*, otherwise we did not comply with the third condition defining an LL(1) grammar — cf. (4). But we happen to have formed:

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

That implies that “<” *is* a possible symbol after *‘Value’*...
We are going to show that we are able to remove this difficulty. The sketch is to make appear the sub-word ‘Value [SubValSetSuf]’ everywhere it’s possible thanks to some expansions, and to form a rule with it. We’ll try next to transform this rule in order to get a definition never producing a word containing ‘Value [SubValSetSuf]’: at each occurrence of it (in the dependant rules or in the rule itself) we’ll substitute a call to this rule. Notice it was not sure *a priori* that such a solution existed.

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

Converse factorisation (‘SVSAux’).

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

Total expansion of '*Value*'.

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

SVSAux1  →  AuxVal1 [SubValSetSuf]

SVSAux2  →  AuxVal2 [SubValSetSuf]
         |  SubValSetSuf
```

Total expansion of 'AuxVal0'.

Converse expansions ('SVSAux1' and '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
```

Converse expansion ('SVSAux3').

Total expansion of 'AuxVal1'.

Total expansion of 'AuxVal2' and 'SubValSetSuf'.

Converse expansions ('SVSAux1' and 'SVSAux2').

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

We recognize 'SVSAux' in 'SVSAux2', and then prefix factorisation ('SVSAux21').
Total expansion of 'SpecVal' in 'SVSAux1' and 'SVSAux3'.
Converse total expansion of 'SVSAux11' ('SVSAux11'):
SVSAux11 $\rightarrow$ AuxVal11 [SubValSetSuf]
and then total expansion of 'AuxVal11' in '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

Total expansion of 'SpecVal' in 'SVSAux11'.
We recognize 'SVSAux' in 'SVSAux1', 'SVSAux3' and 'SVSAux11'.

3.4.6 Step 5

Here stands the result of the previous sequence of transformations.

SubtypeSpec	$\rightarrow$	“(” {SubtypeValueSet “$\vert$” ...}$^{+}$ “)”
SubtypeValueSet	$\rightarrow$	INCLUDES Type
	$\vert$	MIN SubValSetSuf
	$\vert$	FROM SubtypeSpec
	$\vert$	SIZE SubtypeSpec
	$\vert$	WITH InnerTypeSuf
	$\vert$	SVSAux

InnerTypeSuf	$\rightarrow$	COMPONENT SubtypeSpec
	$\vert$	COMPONENTS MultipleTypeConstraints
MultipleTypeConstraints	$\rightarrow$	“{” [“...” “,”] {[NamedConstraint] “,” ...} “}”
NamedConstraint	$\rightarrow$	`lower`$_{id}$ [SubtypeSpec] [PresenceConstraint]
	$\vert$	SubtypeSpec [PresenceConstraint]
	$\vert$	PresenceConstraint
PresenceConstraint	$\rightarrow$	PRESENT
	$\vert$	ABSENT
	$\vert$	OPTIONAL

SubValSetSuf	$\rightarrow$	[“<”] “..” [“<”] UpperEndValue
UpperEndValue	$\rightarrow$	Value
	$\vert$	MAX

SVSAux	$\rightarrow$	BuiltInValue [SubValSetSuf]
	$\vert$	AuxType “:” SVSAux
	$\vert$	NULL [SVSAux3]
	$\vert$	`upper` SVSAux1
	$\vert$	`lower` [SVSAux2]
	$\vert$	[“-”] `number` [SubValSetSuf]
SVSAux1	$\rightarrow$	SubtypeSpec* “:” SVSAux
	$\vert$	“.” SVSAux11
SVSAux2	$\rightarrow$	“:” SVSAux
	$\vert$	“..” [“<”] UpperEndValue
	$\vert$	“<” SVSAux21
SVSAux3	$\rightarrow$	SubtypeSpec* “:” SVSAux
	$\vert$	SubValSetSuf
SVSAux11	$\rightarrow$	`upper`$_{typ}$ SubtypeSpec* “:” SVSAux
	$\vert$	`lower`$_{val}$ [SubValSetSuf]
SVSAux21	$\rightarrow$	Type “:” SVSAux
	$\vert$	“..” [“<”] UpperEndValue

3.5 The new ASN.1:1990 grammar

We give here the new ASN.1:1990 grammar we got thanks to the previous transformations. *It describes exactly the same language as the standard grammar of ISO (modulo the three corrigenda).*

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`$_{typ}$ "::=" Type
	|	`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	$\rightarrow$	$\texttt{lower}_{id}$ [“<”] Type
	$\mid$	`upper` [“.” $\texttt{upper}_{typ}$] SubtypeSpec*
	$\mid$	NULL SubtypeSpec*
	$\mid$	AuxType

NamedNumber	$\rightarrow$	$\texttt{lower}_{id}$ “(” AuxNamedNum “)”
AuxNamedNum	$\rightarrow$	[“-”] `number`
	$\mid$	[$\texttt{upper}_{mod}$ “.”] $\texttt{lower}_{val}$

NamedBit	$\rightarrow$	$\texttt{lower}_{id}$ “(” ClassNumber “)”

ElementType	$\rightarrow$	NamedType [ElementTypeSuf]
	$\mid$	COMPONENTS OF Type
ElementTypeSuf	$\rightarrow$	OPTIONAL
	$\mid$	DEFAULT Value

Class	$\rightarrow$	UNIVERSAL
	$\mid$	APPLICATION
	$\mid$	PRIVATE
<u>*ClassNumber*</u>	$\rightarrow$	`number`
	$\mid$	[$\texttt{upper}_{mod}$ “.”] $\texttt{lower}_{val}$

VALUES

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

<table>
<tr><td><u>BuiltInValue</u></td><td>→</td><td>TRUE</td></tr>
<tr><td></td><td>|</td><td>FALSE</td></tr>
<tr><td></td><td>|</td><td>PLUS-INFINITY</td></tr>
<tr><td></td><td>|</td><td>MINUS-INFINITY</td></tr>
<tr><td></td><td>|</td><td><code>basednum</code></td></tr>
<tr><td></td><td>|</td><td><code>string</code></td></tr>
<tr><td></td><td>|</td><td>“{” [BetBraces] “}”</td></tr>
</table>

<table>
<tr><td>BetBraces</td><td>→</td><td>AuxVal0 [AuxNamed]</td></tr>
<tr><td></td><td>|</td><td>“-” <code>number</code> [AuxNamed]</td></tr>
<tr><td></td><td>|</td><td><code>lower</code> [AuxBet1]</td></tr>
<tr><td></td><td>|</td><td><code>upper</code> AuxBet2</td></tr>
<tr><td></td><td>|</td><td><code>number</code> [AuxBet3]</td></tr>
<tr><td>AuxBet1</td><td>→</td><td>“(” ClassNumber “)” ObjIdComponent*</td></tr>
<tr><td></td><td>|</td><td>AuxNamed</td></tr>
<tr><td></td><td>|</td><td>AuxVal2 [AuxNamed]</td></tr>
<tr><td></td><td>|</td><td>“-” <code>number</code> [AuxNamed]</td></tr>
<tr><td></td><td>|</td><td>AuxVal0 [AuxNamed]</td></tr>
<tr><td></td><td>|</td><td><code>lower</code> [AuxBet11]</td></tr>
<tr><td></td><td>|</td><td><code>number</code> [AuxBet3]</td></tr>
<tr><td></td><td>|</td><td><code>upper</code> AuxBet2</td></tr>
<tr><td>AuxBet2</td><td>→</td><td>SpecVal [AuxNamed]</td></tr>
<tr><td></td><td>|</td><td>“.” AuxBet21</td></tr>
<tr><td>AuxBet3</td><td>→</td><td>ObjIdComponent$^+$</td></tr>
<tr><td></td><td>|</td><td>AuxNamed</td></tr>
<tr><td>AuxBet11</td><td>→</td><td>“(” ClassNumber “)” ObjIdComponent*</td></tr>
<tr><td></td><td>|</td><td>ObjIdComponent$^+$</td></tr>
<tr><td></td><td>|</td><td>AuxVal2 [AuxNamed]</td></tr>
<tr><td></td><td>|</td><td>AuxNamed</td></tr>
<tr><td>AuxBet21</td><td>→</td><td><code>upper</code>$_{typ}$ SpecVal [AuxNamed]</td></tr>
<tr><td></td><td>|</td><td><code>lower</code>$_{val}$ [AuxBet3]</td></tr>
<tr><td>AuxNamed</td><td>→</td><td>“,” {NamedValue “,” …}$^+$</td></tr>
<tr><td>NamedValue</td><td>→</td><td><code>lower</code> [NamedValSuf]</td></tr>
<tr><td></td><td>|</td><td><code>upper</code> AuxVal1</td></tr>
<tr><td></td><td>|</td><td>[“-”] <code>number</code></td></tr>
<tr><td></td><td>|</td><td>AuxVal0</td></tr>
<tr><td>NamedValSuf</td><td>→</td><td>Value</td></tr>
<tr><td></td><td>|</td><td>AuxVal2</td></tr>
</table>

SUBTYPES

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	→	$\mathtt{lower}_{id}$ [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	→	$\mathtt{upper}_{typ}$ SubtypeSpec* “:” SVSAux
	|	$\mathtt{lower}_{val}$ [SubValSetSuf]
SVSAux21	→	Type “:” SVSAux
	|	“..” [“<”] UpperEndValue

4 LL(1) checking

We bring here the proof that the previous grammar, obtained after a lot of transformations, is LL(1). We first give an exact definition of the LL(1) property.

4.1 Definition of the LL(1) property

LL(1) grammars generate languages that can be top-down parsed, with one token of look-ahead and no back-tracking (*Left to right scanning of the input, building a Leftmost derivation with one token of look-ahead*). In order to define them formally, it is necessary to introduce before two functions. We'll call $\mathcal{N}$ the set of the nonterminals and Σ the set of the terminals.

4.1.1 The *First* function

The *first* function $\mathcal{P}$ is defined as:

$$\forall \mathrm{A} \in \mathcal{N}, \quad \mathcal{P}(\mathrm{A}) = \{x \in \Sigma \cup \{\varepsilon\} \mid \mathrm{A} \overset{*}{\Longrightarrow} x\alpha \;\; \textit{and} \quad x \neq \varepsilon \;\; \textit{or} \;\; \mathrm{A} \overset{*}{\Longrightarrow} x\}$$

4.1.2 The *Follow* function

The *follow* function $\mathcal{S}$ is defined as:

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

4.1.3 LL(1) definition

We note here the implication relation by "$\models$", in order to avoid any confusion with the "$\Longrightarrow$" relation. A grammar is LL(1) if and only if it satisfies the following properties:

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

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

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

4.1.4 Extension to rational operators

We saw at section1.3 the definition of the rational operators used in this ASN.1 grammar. We pointed out we could have considered rational expressions as being produced by a specific rule. We're going to extend the functions $\mathcal{P}$ et $\mathcal{S}$ to these expressions, and give a recursive and algorithmic definition.

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

Notice that in order to ease the definition we extended $\mathcal{P}$ to ε (in spite of ε never appears explicitly in the grammar) and γ can be ε.

New definition of $\mathcal{S}$:

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

Everywhere it is possible we can read the previous array substituting A by [A], A^* or A^+.

It was the general definition but, in our study, grammars don't own explicit empty productions (Cf. 2). We can thus replace equation P3 by the following algorithm: at each occurrence of the rational expressions α^*, α^+, $[\alpha]$, $\{A\ a\ \ldots\}^*$, et $\{A\ a\ \ldots\}^+$, consider it as being produced by a specific rule (without sharing) and check the following associated constraints:

Rational rule	Constraint
$X \rightarrow \alpha^*$	$\mathcal{P}(\alpha) \cap \mathcal{S}(X) = \emptyset$
$X \rightarrow \alpha^+$	$\mathcal{P}(\alpha) \cap \mathcal{S}(X) = \emptyset$
$X \rightarrow [\alpha]$	$\mathcal{P}(\alpha) \cap \mathcal{S}(X) = \emptyset$
$X \rightarrow \{A\ a\ \ldots\}^*$	$(\mathcal{P}(A) \cup \{a\}) \cap \mathcal{S}(X) = \emptyset$
$X \rightarrow \{A\ a\ \ldots\}^+$	$\{a\} \cap \mathcal{S}(X) = \emptyset$
$X \rightarrow \{[A]\ a\ \ldots\}$	$(\mathcal{P}(A) \cup \{a\}) \cap \mathcal{S}(X) = \emptyset$

The fact to do not consider rule sharing ease the work if it is "hand-made": the cost is the same but the task is more localized at each step. A software could get rid of this election, of course.

4.2 LL(1) property checking of the new ASN.1:1990 grammar

4.2.1 Equation P1

We present in the following array the nonterminal which are generated at the head of each production. We easily check by transitive closure that no left recursion appears.

Rule	Head
ModuleDefinition	ModuleIdentifier
ModuleIdentifier	
ObjIdComponent	
TagDefault	
ModuleBody	Exports, Imports, Assignment
Exports	
Imports	
SymbolsFromModule	Symbol
Symbol	
Assignment	

Rule	Head
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 Equation P2

For all the rules we write the constraint imposed by equation P2, and then we calculate the sets $\mathcal{P}(\alpha_i)$ necessary to solve it. For easy reading we won't write the constraint if all the α_i are tokens (trivial). The same we'll reduce directly the $\mathcal{P}(\alpha_i)$, where α_i is a rational expression, to their form without operator (Cf. 4.1.4).

Rule	Constraint
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	

Rule	Constraint
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	$\mathcal{P}(\text{Type}) \cap \{\text{“..”}\} = \emptyset$

Finally, after simplification, the following set of equations must be satisfied:

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

We have:

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

Thus equation (1) is satisfied.

$$\mathcal{P}(\text{SubtypeSpec}) = \{\ \text{``(''}\ \}$$

Thus equation (2) is satisfied.

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

Thus equation (3) is satisfied.

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

Thus equation (4) is satisfied.

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

Thus equations (5) and (11) are satisfied.

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

Thus equation (6) is satisfied.

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

Thus equations (7) and (8) are satisfied.

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

Thus equation (9) is satisfied.

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

Thus equation (10) is satisfied.

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

Thus equation (12) is satisfied.

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

Thus equation (13) is satisfied.

4.2.3 Équation P3

We give here for each production of each rule the constraints imposed by equation P3 (Cf. 4.1.4). In order to not overload the array, redundant equations inside a rule and trivial equations don't appear. We simplify also all that can be locally simplified.

Rule	Constraint
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	$\mathcal{P}(\text{SubtypeSpec}) \cap \mathcal{S}(\text{TypeSuf}) = \emptyset$ $(\mathcal{P}(\text{ElementType}) \cup \{\text{“,”}\}) \cap \{\text{“\}”}\} = \emptyset$
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	

Rule	Constraint
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	$\{\text{“...”, “\}”}\} \cap \mathcal{P}(\text{NamedConstraint}) = \emptyset$
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	$\{\text{“<”}\} \cap \mathcal{P}(\text{UpperEndValue}) = \emptyset$

Finally, after simplifications, the following set of equations must be satisfied:

$$
\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{``\}''}\} = \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}
$$

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

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

Equations (10), (14) and (40) are immediately satisfied thanks to the sets $\mathcal{P}$ previously computed (Cf. 4.2.2). We calculate hence the missing sets $\mathcal{P}$:

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

Thus equations (1) and (13) are satisfied.

$$
\begin{array}{l}
\mathcal{P}(\text{Exports}) = \{\text{EXPORTS}\} \\
\mathcal{P}(\text{Imports}) = \{\text{IMPORTS}\}
\end{array}
$$

Thus equation (5) is satisfied.

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

Thus equations (6) and (7) are satisfied.

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

Thus equation (2) is satisfied.

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

Thus equation (9) is satisfied.

$$\begin{array}{lcl}
\mathcal{P}(\text{Class}) & = & \{\ \text{UNIVERSAL, APPLICATION, PRIVATE}\ \} \\
\mathcal{P}(\text{ClassNumber}) & = & \{\ \texttt{number}, \texttt{upper}, \texttt{lower}\ \}
\end{array}$$

Thus equation (12) is satisfied.

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

Thus equation (16) is satisfied.

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

Thus equation (38) is satisfied.

$$\begin{array}{lcl}
\mathcal{P}(\text{SubtypeSpec}) & = & \{\ \text{“(”}\ \} \\
\mathcal{P}(\text{NamedConstraint}) & = & \{\texttt{lower}\} \cup \mathcal{P}(\text{SubtypeSpec}) \cup \mathcal{P}(\text{PresenceConstraint}) \\
 & = & \{\ \texttt{lower}, \text{“(”, PRESENT, ABSENT, OPTIONAL}\ \}
\end{array}$$

Thus equation (39) is satisfied.

Moreover:

$$\begin{array}{lcl}
\mathcal{P}(\text{TypeSuf}) & = & \mathcal{P}(\text{SubtypeSpec}) \cup \{\text{“\{”, SIZE, OF}\} \\
 & = & \{\ \text{“(”, “\{”, SIZE, OF}\ \} \\
 & & \\
\mathcal{P}(\text{ElementTypeSuf}) & = & \{\ \text{OPTIONAL, DEFAULT}\ \} \\
 & & \\
\mathcal{P}(\text{AuxVal2}) & = & \{\ \text{“<”, “:”}\ \} \\
 & & \\
\mathcal{P}(\text{SpecVal}) & = & \mathcal{P}(\text{SubtypeSpec}) \cup \{\text{“:”}\} \\
 & = & \{\ \text{“(”, “:”}\ \} \\
 & & \\
\mathcal{P}(\text{AuxBet1}) & = & \{\text{“(”, “-”}, \texttt{upper}, \texttt{lower}, \texttt{number}\} \cup \mathcal{P}(\text{AuxNamed}) \\
 & & \cup\ \mathcal{P}(\text{AuxVal2}) \cup \mathcal{P}(\text{AuxVal0}) \\
 & = & \{\ \text{“(”, “-”}, \texttt{upper}, \texttt{lower}, \texttt{number}, \text{“,”, “<”, “:”,} \\
 & & \text{TRUE, FALSE, PLUS-INFINITY, MINUS-INFINITY,} \\
 & & \texttt{basednum}, \texttt{string}, \text{“\{”, NULL, “[”, SET, SEQUENCE,} \\
 & & \text{BOOLEAN, INTEGER, BIT, OCTET, CHOICE,} \\
 & & \text{ANY, OBJECT, ENUMERATED, REAL, EXTERNAL,} \\
 & & \text{“NumericString”, “PrintableString”, “TeletexString”,} \\
 & & \text{“T61String”, “VideotexString”, “VisibleString”,} \\
 & & \text{“ISO646String”, “IA5String”, “GraphicString”,} \\
 & & \text{“GeneralString”, “UTCTime”, “GeneralizedTime”,} \\
 & & \text{“ObjectDescriptor”}\ \}
\end{array}$$

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

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

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

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

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

$$
\mathcal{P}(\text{SVSAux2}) \ = \ \{\ \text{“:”, “..”, “<”}\ \}
$$

Thus $\mathcal{P}(\text{SubValSetSuf}) \subset \mathcal{P}(\text{SVSAux2}) \subset \mathcal{P}(\text{SVSAux3})$,
which allow the removal of equations(43) and (45), because they are implied by (44).

Moreover, it is pertinent to notice that:

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

Thus we can remove equation (46) because it is implied by (44).

We compute now some sets $\mathcal{S}$ that, thanks to previously calculated sets $\mathcal{P}$, allow us to conclude in one step.

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

Thus equation (3) is satisfied.

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

Thus equation (20) is satisfied.

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

Thus equations (18) and (19) are satisfied.

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

Thus equations (23), (24) and (25) are satisfied

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

Thus equations (26), (27), (28), and (29) are satisfied.

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

Thus equations (32) and (33) are satisfied.

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

Thus equation (30) is satisfied.

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

Thus equations (34) and (35) are satisfied.

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

Thus equation (31) is satisfied.

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

Thus equations (41) and (42) are satisfied.

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

Thus equation (36) is satisfied.

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

Thus equation (37) is satisfied.

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

Thus equation (4) is satisfied.

We only have now to check the following system (we replace the sets $\mathcal{P}$ by their value, except $\mathcal{P}$(NamedValSuf)):

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

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

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

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

We can therefore merge equations (8), (11), (15), (17) and (47) in one single, and the system is equivalent to:

(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)	$\{$"("$,$ "<"$,$ ".."$,$ ":"$\} \cap \mathcal{S}(\text{SVSAux}) = \emptyset$

We have:

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

And it follows, with the remark that $\mathcal{S}(\text{ElementType}) \subset \mathcal{S}(\text{NamedType}) \subset \mathcal{S}(\text{AuxType})$:

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

On the other hand:

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

And finally:

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

Thus equation (X) is satisfied.

Note that $\left\{ \begin{array}{lcl} \mathcal{S}(\text{Value}) & \subseteq & \mathcal{S}(\text{AuxVal0}) \\ \mathcal{S}(\text{AuxVal0}) & \subseteq & \mathcal{S}(\text{Value}) \end{array} \right.$

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

We can therefore merge equations (21) and (22), and then the system is equivalent to:

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

Moreover:

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

We have:

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

Thus $\mathcal{S}(\text{SVSAux}) = \{\text{“|”}, \text{“)”}\}$ *and so equation (44) is satisfied.*

Now we have to compute:

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

We have:

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

Hence

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

Moreover:

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

It follows:

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

On the other hand:

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

We have

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

Hence

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

Therefore:

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

And finally:

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

Hence

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

Thus equation (Y) is satisfied.

Conclusion The system of equations is entirely satisfied, i-e. the new ASN.1:1990 grammar is LL(1).

5 Designing parsers in Caml Light

We describe in this section a method for writing parsers of LL(1) grammars in Caml Light, *but it intends to universal.* It is based upon the format presented at section 1.2 and some additional constraints imposed by the stream pattern matching semantics. (Cf. Introduction) We show how to produce error messages (without help of the context) solely for each rule susceptible of analysis abortion, and in a systematic way. We'll see moreover how partial application and full functionality allow high-order parsers building. It is recommended to the interested reader to read before [7], [8] (for beginners) or [5].

5.1 Stream constraint

In general it is not possible to translate *directly* a LL(1) grammar to its corresponding correct parser.
Rules that are a problem have the form $\mathrm{A} \rightarrow \mathrm{X\ B} \mid \mathrm{C}$ where $\mathrm{X} \stackrel{*}{\Longrightarrow} \varepsilon$.
Suppose indeed X recognise ε but B fails after; in this case the exception `Parse_failure` raised by (the parser associated to) B becomes at the level of A in `Parse_error` *because B has not been called in head of the stream pattern*, and so stop the parsing process although C could have matched. It's obvious that there was no danger to go on parsing because no token was removed from the stream. A simpler semantics for streams justifies this drawback, but it implies a previous analysis and maybe modification of LL(1) grammars.
The first step thus is to put the LL(1) grammar to the format presented in this document (cf. 1.2) *and* apply the following additional transformations, until it's impossible:

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

This way we comply with the constraint imposed by Caml Light streams.

Remark If the empty word belongs to the language, we note the appearance of the unique explicit empty production of the grammar (let $\beta = \varepsilon$).

5.2 Plea for streams

The stream constraint previously presented upper do not have to make us forget the great number of advantages they bring. First, it is erroneous that the streams only allow us to parse LL(1) grammars[5].

[5]Modulo the constraint given in the previous section

Let's consider the famous "the dangling `else`" case. Let the ambiguous grammar [2]:

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

This grammar features the `if then else` construct. It leads down-top parsers like YACC to a shift/reduce conflict in presence of the `else` clause. Usually in this case one privilege the shift action in order to associate the `else` to the last non-closed `then`. With Caml Light this is done naturally, although there is no underlying stack-automaton for parsing. One has just to write the pattern of S' *with* **else** *S' as the first alternative*. The patterns indeed in a Caml Light matching are evaluated in writing order (from left to right and up to bottom); hence the parser S' will privilege (due to pattern matching semantics) the alternative **else** S' instead of ε.

Now we're going to show, following the example given in [7, 2], that we can also parse context-sensitive languages thanks to the Caml Light streams and to full functionality.
Let the language $\{wcw \mid w \in (a+b)^*\}$, where a, b and c are tokens. It has been proved that it is context-dependent. The idea used to parse this language is to parse the prefix w and then built *dynamically* a list of parsers matching each character composing w. Hence, after c reading, we use this list to parse the suffix w. Look at the details.
So we first need a function `wd` (standing for "word definition") that parse w and built the latter list. In Caml Light the characters (of type `char`) a and b are respectively noted `‘a‘` and `‘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⟩
```

The second parser[6] take the parsers list produced by `wd` and match them to the current stream. It is called `wu` (for "word usage").

```
#let rec wu = function
    p::pl → (function [⟨ p x; (wu pl) w ⟩] → x^w)
  | [ ] → (function [⟨ ⟩] → "")
;;
wu : (α stream → string) list → α stream → string = ⟨fun⟩
```

[6] A parser is merely a function.

Finally a parser for our language is:

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

Application:

#wcw (stream_of_string "abaacabaa");;
− : *string* = "abaa"

5.3 Error handling

An important concern when parsing is the earliest detection of errors (the so called property of "the longest valid prefix"). Moreover messages should be the more informative as possible. It is to be inferred from the latter that the more context we have at the time of error detection, the more pertinent the message will be. When designing and implementing with Caml Light, it means that we should add parameters to parsers, dedicated to this purpose. We prefer here for sake of simplicity to ignore context. On the other hand we take care to raise only one message for a syntax error; that is to say that in the path (of the derivation tree — see 5.4) from the erroneous token to the root (*axiom*), no other message will be produced. Notice that error recovering, even in panic mode[7] is not easy *a priori*. Generally the semi-colon (as separator or terminator) is used to this recovering, but the absence of such a mark in ASN.1 allows nothing akin. We could imagine to re-synchronise parsing on symbol "::=", but in case of value declaration the value identifier may be arbitrary far behind "::="... There is hence no simple, satisfactory and general solution in case of only one module parsing. If an error is encountered while structured value parsing, we can re-synchronise on the next closing brace; otherwise if there are many modules in the same source file we can go to the END of the current module where appeared the error. Although this strategy is not difficult, it has not been implemented.
From the implementation point of view, we have a module called `errors` which exports a function `syntax_error` of which type is `string` → α `stream` → β, where the first argument is an error message and the second the current token stream which head is the erroneous token. It suspend the execution raising the exception `Parsing_error`.

We propose now a general method for error message display. Basically there is no difference of handling between lexical and syntactic errors: what is meaningful is that an error occurred and we want to inform the user the most precisely as possible. Modulo a little constraint upon lexical analysis, the functions presented below are *independent* from the compilers where we want to reuse them. We could also use them for semantic errors, for example.

[7] Tokens are ignored until one which belongs to a previously fixed set.

5.3.1 Elections for the lexer

The basic principle is the adjunction to each lexed token of its *location* in the source text, that is to say of both its position measured in characters from the beginning, and its concrete syntax, i-e. the character string identifying the token in the source. In fact, for error displays, the concrete syntax is not necessary because the length (in characters) of the erroneous source fragment is enough (For a token it would be its number of characters.). The convention for lexing are the following.

1. the location of the first character of the source text is 1.
2. We use a fictive token (also called "virtual") as a sentry in the stream produced by the lexer, that is to say it always exists at eh end of the produced stream this special token of which unique argument is a location. This latter is 1 if the (real) token stream is empty, and equal to the length of the source text *plus one* otherwise.
3. We always keep an original copy of the token stream where the error occurred.

5.3.2 Display format of error messages

The display format is the same as Caml Light one:

- The name of source file.
- The number of the erroneous line.
- The location of the first erroneous character, counted from the beginning of the line.
- The erroneous line, with the error underlined.
- The specific error message.
- The location of the first erroneous character, counted from the beginning of the source file.

For instance, let `err.asn` the source file:

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

Then we'll get:

```
Asno'90 0.1 parser
File "err.asn", line 1, char 6
> ERR {} DEFINITIONS ::=
>      ^
> Object identifier component expected at char 6
```

5.3.3 The module errors

First here's three auxiliary functions.

- The function `tabulation` counts the number of tabulations in its argument string (it's used to underline correctly the error).
- The function `out_string` send to the standard output the string passed as argument and flush it.
- The function `get_til_eol` takes as sole argument a character stream and returns the concatenation of the read characters until an *End Of Line*, or else until stream exhaustion.

```
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
| [⟨ ⟩] → ""
;;
```

We give now an auxiliary function `find_error` which main role is, from an absolute location (i-e. counted from the beginning of the source text) and the original character stream, to return a triple formed by the number of the erroneous line, the relative location (i-e. counted from the beginning of the line), and a string representing this line.

```
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)
;;
```

To terminate, here's the main function `print_error` which is the sole exported outside the module `errors`, and therefore is the sole usable in the compilers. Its arguments denote:

- A header (`header`) qualifying the part of the compiler raising the error (for instance: "Asno'90 0.1 lexer").
- The original character stream (`strm`).
- The filename of the analysed source (`filename`).
- The error message (`msg`).
- The absolute location of the first character of the erroneous zone (`ofs`).
- The length of the erroneous zone (`len`).

```
let print_error header strm filename msg ofs len =
 begin
  out_string ("\n" ^ header);
  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 Analysis method

Viewing an analysis method has to do with the election of the programming language used for implementation. Caml Light is a statically strong-typed language, what implies that functions may take as argument a function and return a function. In attribute grammars terminology the values passed to the parsers (i-e. analysis functions) are called *inherited attributes*, and their result *synthesised attributes*. In operational terms, the tree of function calls is named *derivation tree*.

As it is said in the introduction, Caml Light allows a *descendent* analysis, that is to say that the derivation tree is built from nodes to leaves. We call *abstract-syntax tree* the result of the highest-level parser (i-e. the synthesised attribute of the grammar axiom). The result of a parser can be either an abstract-syntax subtree or a *function* of which later application will produce an abstract-syntax tree.

In the frame of our present study, we voluntarily restrict us to a *purely* synthesised analysis, that is to say information for parser result building circulate in the derivation tree from the leaves to the nodes — in other words: there's no inherited attributes. This is done thanks to functional synthesised attributes. Assume indeed that a node of the derivation tree has an inherited attribute, we remove it and we abstract[8] the synthesised attribute with regards to this inherited attribute. Thus the final computation will be done at the level of the father of this node, where is all the needed information.

This method presents the advantage, in the eventual frame of an automatic parser generation, to separate well syntax and calculation of the syntactic tree. Indeed, if the first pass is devoted to the generation of a parser which answers yes or no, according to syntactic correction, then the second pass completes this parser instead of rewriting the argument declarations. Moreover we visually separate the nature of attributes: inherited for parsing and syntax errors handling and synthesised for abstract-syntax tree computation. One could object that if the number of inherited attributes at the beginning is large, we lose in readability at the end. The example of applying this method to ASN.1:1990 shows it is nevertheless a worthwhile approach. The sole concession to functionality is two top-level references representing the current module name and the default type-tagging mode, because these informations can be useful at any moment and it would have been cumbersome to abstract *all* the synthesised attributes over these values.

The semantics of stream pattern matchings impose on the other hand that attribute evaluation is from left to right. Hence, if we have the rule:

$\mathrm{Z} \rightarrow \mathrm{X}_0\ \mathrm{X}_1\ \ldots\ \mathrm{X}_n$, each X_i having as synthesised attributes s_i, then s_i can be function of the $s_{j\leq i}$, but not of the $s_{j>i}$.

[8] *To abstract* an expression e with regards to a variable x consists in forming the function `fun x -> e`.

5.5 General form of parsers

5.5.1 Code structuring

We have to define first a Caml Light type with two constant constructors, and of which values passed to parsers indicate in case of syntactic failure whether the parser must stop (and give an error message) or not.

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

In order to allow eventual partial applications of these parsers, we place the argument of type `Parsing_mode` in first position. Therefore the general forma of parsers is:

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

If know that the parser will never stop the analysis (we'll see in 5.7 how to decide this point), it's enough to put for the moment an empty error message.

We understand now better one of the advantages to have no rule producing explicitly ε: we use the empty stream pattern for error handling.

5.5.2 Renaming rules

A small problem about naming conventions of parser was til now hidden. *A priori* we take as identifiers the names of the associated grammatical rules, but we must take care to not generate illegal Caml Light identifiers, like keywords. So, if this situation occurs, we must treat specially these names. For instance, here we chose to prefix the generated identifier by the character `x`.

5.6 Rational operators coding

The rational operators correspond to high-order parsers: they take as first argument the necessary parsers, then the failure mode and finally the token stream. This way we can partially apply an operator to its first argument and thus we obtain a parser which can be in turn be used to apply another operator. In other words: we can *combine* arbitrarily operators.

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

This operator definition was: X $\rightarrow$ α X | ε. Its semantics is the list of the semantics of the α read:

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

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

This operator definition was: X $\rightarrow$ $\alpha\alpha^*$. Its semantics is the list of the semantics of the α read:

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

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

Let's recall the definition:X $\rightarrow$ α | ε. A first approach consistent with the other operators is to return the empty list (empty is ε is read).

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

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

The definition was: X $\rightarrow$ | A (a A)*. We first code an auxiliary function which is also useful to operator { A a ...}*, and which corresponds to (a 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
     | [⟨ ⟩] → [ ]
;;
```

The function `elm` parses the nonterminal “A”, and `term` reads the token “a”. `Symbol` is token constructor of which first argument is its location in the text source (`int` type), and of which second argument is its concrete syntax (`string` type).
It follows:

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

5.6.5 $\{$ A a $\ldots\}^+$

The definition was: X $\rightarrow$ A (a A)*. We then get directly:

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

5.6.6 $\{$[A] a $\ldots\}$

Recall the definition: X $\rightarrow \varepsilon$ | A (a [A])* | (a [A])$^+$, what can be rewritten:
X $\rightarrow \varepsilon$ | A (a [A])* | a [A] (a [A])*. We first define an auxiliary parser which recognise (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
     | [⟨ ⟩] → [ ]
;;
```

Therefore:

```
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

Until now parsers have the same format that makes necessary a parameter indicating the behaviour in case of syntactic failure (Cf. 5.5). But it is clear that some functions, in case of failure, can never stop the parsing process, and others make it stop always. These are the parsers we want now to optimise, suppressing their useless behaviour-parameters, and we do the same with rational operators.
The advantage of such an optimisation is double: in one hand we create less closures, and on the other hand we remove useless code. The disadvantage is also double: in one hand we need an additional previous analysis of the grammar, and on the other hand we lose the ability to combine arbitrarily operators (for typing reasons).

5.7.1 X → α^*

We had:

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

We note that argument `mode` is useless. Indeed we must always accept ε, and hence `my_parser` must never interrupt execution. We assume therefore from now that we pass as argument a parser which never aborts the process, because it doesn't have any behaviour-argument and it raises `Parse_failure` on ε, or because it owns a behaviour-argument and this one has been applied to `Fail`. The code becomes:

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

5.7.2 X → α^+

We got:

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

We cannot here remove the behaviour-parameter in the operator: it is `my_parser` which determines its usage. The optimisation of `star` implies nevertheless the modifications of its calls:

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

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

We had:

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

We understand here we can suppress the behaviour-parameter because it is left unused inside the function body (and ε must always be readable). We thus modify the operator as we did for `star`, taking care of changing all the calls to `option` in accordance. On the other hand since we chose to optimise (and therefore to lose the common format for operator types), it would be better to change the type of the returned value from `option`. We see indeed that the semantics of this operator is particular: on one hand we should express "the semantics of α", and on the other hand "no semantics". In order to do so, we define a polymorphic type which allow us to build such optional values:

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

Hence, it follows:

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

5.7.4 $\{ \mathbf{A}\ \mathbf{a} \ldots \}^*$

We gave:

```
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
    | [⟨ ⟩] → [ ]
;;
```

It's obvious that the behaviour-parameter is useless because it serves only to the recursive call (and ε must always be readable). So:

```
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
    | [⟨ ⟩] → [ ]
;;
```

Moreover we had:

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

Hence:

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

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

We got:

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

Here we cannot remove the behaviour-parameter.

5.7.6 {[A] a ...}

We had:

```
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
     | [⟨ ⟩] → [ ]
;;
```

We have to remove the argument `mode` from the call to `option`. Thus the behaviour-parameter can be suppress because it is only used in the recursive call (and ε is accepted). Therefore:

```
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
     | [⟨ ⟩] → [ ]
;;
```

On the other hand we gave:

```
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
| [⟨ ⟩] → [ ] ;;
```

We have to remove the argument `mode` from the call to `option`. Thus the behaviour-parameter can be suppress because it is only used in the recursive call (and ε is accepted). Therefore:

```
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 Nonterminal analysis

Suppose here we work with the non-implementative ASN.1:1990 grammar (Cf. 3.5), although the optimisation method works fine also with the implementative grammar (Cf. 7.1). We classify in three kinds the parsers, along with their behaviour in case of failure of the stream pattern matching.

Passing There are parsers which never abort execution.

Blocking There are parsers which always abort execution.

Mixed There are parsers which can abort or not execution, along with their context call.

If the axiom (i-e. the entry point of the grammar) is never called *inside* the grammar, then its associated parser is considered as being passing or blocking, in accordance whether the empty word (ε) belongs to the language.

First of all, we traverse the grammar ignoring the expressions which involve the rational operators seen in (1.3). If a parser is always called in head of a pattern, then it is passing; if it is always called after the head of a pattern, then it is blocking; otherwise it is mixed. *Parsers of nonterminals which* only *appear inside rational expressions are considered as being passing for the moment.*

And now we take a look to the rational expressions:

1. α^*
   We must distinguish the first element of α. If it's a nonterminal of which associated parser was blocking, then it becomes mixed. For each following nonterminal, if their parser was passing, it then becomes mixed.

2. α^+
   Let aside the first element of α. For each following nonterminal, if their parser was passing, then it becomes mixed. We must distinguish now along the rational expression position. If α^+ is at the head of a stream pattern and the parser of the first element of α was blocking, then it becomes mixed. If α^+ is not at the head and the parser of the parser of the first element of α was passing, then it becomes mixed.

3. $[\alpha]$
   The same as α^*.

4. $\{$ **A a** $\ldots\}^*$
   If the parser of A was passing or blocking, then it becomes mixed.

5. $\{$ **A a** $\ldots\}^+$
   The same as A^+.

6. { **[A] a** ...}
   The same as [A].

To sum up, and assuming in the previous enumeration that $\alpha = \text{A}$:

<table>
<tr><th></th><th>Head</th><th>A before</th><th>A after</th></tr>
<tr><td>A^*</td><td></td><td>Blocking</td><td>Mixed</td></tr>
<tr><td rowspan="2">A^+</td><td>Yes</td><td>Blocking</td><td rowspan="2">Mixed</td></tr>
<tr><td>No</td><td>Passing</td></tr>
<tr><td>[A]</td><td></td><td>Blocking</td><td>Mixed</td></tr>
<tr><td>$\{\text{A a} \ldots\}^*$</td><td></td><td></td><td>Mixed</td></tr>
<tr><td rowspan="2">$\{\text{A a} \ldots\}^+$</td><td>Yes</td><td>Blocking</td><td rowspan="2">Mixed</td></tr>
<tr><td>No</td><td>Passing</td></tr>
<tr><td>$\{[\text{A}]\ \text{a} \ldots\}$</td><td></td><td>Blocking</td><td>Mixed</td></tr>
</table>

All the necessary mixed parsers need a behaviour-argument; nonetheless some passing or blocking parsers, though they not use such an argument, can need it for typing reasons. For instance it's enough that one of these parsers is passed to an operator α^+ or $\{ \text{A a} \ldots\}^+$, because they need a behaviour-parameter in order to be applied uniformly. So the last step of the method resides in detect which are these parsers and to impose to them a behaviour-parameter. It's this phenomenon that restricts partially the scope of the optimisation process, but the application to ASN.1:1990 shows that only one parser among sixty has a useless parameter.

The blocking parsers have the form:

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

Note that the optional behaviour-parameter *mode* is put in italics with an exponent *opt*.

The passing parsers have the form:

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

Idem for the optional parameter notation. Notice in this case there's no empty pattern: `Parse_failure` is automatically raised when all the head-patterns fail.

The mixed parsers keep of course the form given in (5.5).

5.7.8 Token analysis

We analyse the grammar and we build the set of tokens which do not start any rule. For each one we define a *blocking* parser. Thus, at parser writing-time, we'll take care of reading explicitly the tokens appearing in head of the stream patterns, and the other terminals with their specific blocking parser. This way we remove the possibility of a `Parse_error` raising when a failure on a token inside a pattern.

6 Lexical analysis of ASN.1:1990

6.1 A grammar for the ASN.1:1990 lexicon

From the ISO documentwe can extract a grammar (non LL(1)) for the lexicon. For more details, have a look to the annex below.

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 Lexical ambiguities

According to the ISO document, several terminals semantically different are not lexically distinguishable: only the context of use can tell us which they are. For example, a type identifier `typereference` is identical to a module identifier `modulereference`. The same, a value identifier `valuereference` is identical to a field identifier `identifier` in a SEQUENCE type. In the ASN.1:1990 grammar they will be denoted respectively by the identifiers `upper` and `lower`. When there's no ambiguity, we put in underscript the nature of the terminal:

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

The terminals `bstring` and `hstring` have the same semantics (which is to denote a number in binary or hexadecimal base) and thus are merged under the terminal `basednum`.

Moreover, `cstring` which denotes a character string is renamed simply `string`.

7 Parsing ASN.1:1990

We here carry out all that was said in the previous section in order to write a parser for ASN.1:1990.

7.1 An ASN.1:1990 grammar for implementation

We apply the previous transformations and also the following, for rational operators uniform coding (Cf. 1.3):

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

except if β is actually a nonterminal.

It is obvious that the resulting grammar is still LL(1). Here's the result for ASN.1:1990:

MODULES

ModuleDefinition	$\rightarrow$	ModuleIdentifier DEFINITIONS [Tagging] "::=" BEGIN [ModuleBody] END

ModuleIdentifier → $\mathtt{upper}_{mod}$ [ObjIdCompLst]
ObjIdCompLst → “{” ObjIdComponent$^+$ “}”
ObjIdComponent → `number`
| $\mathtt{upper}_{mod}$ “.” $\mathtt{lower}_{val}$
| `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 → $\mathtt{upper}_{typ}$
| $\mathtt{lower}_{val}$

Assignment → $\mathtt{upper}_{typ}$ “::=” Type
| $\mathtt{lower}_{val}$ Type “::=” Value

TYPES

Type → $\mathtt{lower}_{id}$ “<” Type
| `upper` [AccessType] SubtypeSpec*
| NULL SubtypeSpec*
| AuxType
AccessType → “.” $\mathtt{upper}_{typ}$
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 $\texttt{lower}_{id}$

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

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

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

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

Class	→	UNIVERSAL
	|	APPLICATION
	|	PRIVATE
<u>*ClassNumber*</u>	→	`number`
	|	`lower`$_{val}$
	|	`upper`$_{mod}$ "." `lower`$_{val}$

VALUES

<u>*Value*</u>	→	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

SUBTYPES

<table>
<tr><td>SubtypeSpec</td><td>→</td><td>“(” {SubtypeValueSet “|” . . . }$^+$ “)”</td></tr>
<tr><td>SubtypeValueSet</td><td>→</td><td>INCLUDES Type</td></tr>
<tr><td></td><td>|</td><td>MIN SubValSetSuf</td></tr>
<tr><td></td><td>|</td><td>FROM SubtypeSpec</td></tr>
<tr><td></td><td>|</td><td>SIZE SubtypeSpec</td></tr>
<tr><td></td><td>|</td><td>WITH InnerTypeSuf</td></tr>
<tr><td></td><td>|</td><td>SVSAux</td></tr>
</table>

<table>
<tr><td>SubValSetSuf</td><td>→</td><td>“..” [“<”] UpperEndValue</td></tr>
<tr><td></td><td>|</td><td>“<” “..” [“<”] UpperEndValue</td></tr>
<tr><td>UpperEndValue</td><td>→</td><td>Value</td></tr>
<tr><td></td><td>|</td><td>MAX</td></tr>
</table>

<table>
<tr><td>InnerTypeSuf</td><td>→</td><td>COMPONENT SubtypeSpec</td></tr>
<tr><td></td><td>|</td><td>COMPONENTS MultipleTypeConstraints</td></tr>
<tr><td>MultipleTypeConstraints</td><td>→</td><td>“{” [“...” “,”] {[NamedConstraint] “,” . . . } “}”</td></tr>
<tr><td>NamedConstraint</td><td>→</td><td>lower$_{id}$ [SubtypeSpec] [PresenceConstraint]</td></tr>
<tr><td></td><td>|</td><td>SubtypeSpec [PresenceConstraint]</td></tr>
<tr><td></td><td>|</td><td>PresenceConstraint</td></tr>
<tr><td>PresenceConstraint</td><td>→</td><td>PRESENT</td></tr>
<tr><td></td><td>|</td><td>ABSENT</td></tr>
<tr><td></td><td>|</td><td>OPTIONAL</td></tr>
</table>

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

7.2 Parser optimisation

We carry out here the optimisation method we previously saw for parsers, using the grammar for implementation. Note that we add a rule (Specification $\rightarrow$ ModuleDefinition$^+$) which plays the role of "super-axiom" (i-e. "super entry-point"), in order to allow the parsing of various ASN.1 modules in the same source file. We point out for each parser if it's need the behaviour-parameter *mode* (even if it does not use it). We get:

Parser	Status	Mode	Error message
Specification	Blocking	No	Module definition expected
ModuleDefinition	Passing	Yes	
ModuleIdentifier	Mixed	Yes	Module reference expected
ObjIdCompLst	Passing	No	
ObjIdComponent	Mixed	Yes	Object identifier component expected
ClassAttr	Passing	No	
Tagging	Passing	No	
TagDefault	Passing	No	
ModuleBody	Passing	No	
Exports	Passing	No	
Imports	Passing	No	
SymbolsFromModule	Passing	No	
Symbol	Mixed	Yes	Type reference or value reference expected
Assignment	Mixed	Yes	Type definition or value definition expected
Type	Mixed	Yes	Type expected
AccessType	Passing	No	
AuxType	Passing	No	
SetSeq	Passing	No	
TypeSuf	Passing	No	
BuiltInType	Passing	No	
NamedNumLst	Mixed	Yes	Left braces beginning a named number list expected
NamedBitLst	Passing	No	
AnySuf	Passing	No	
NamedType	Mixed	Yes	Named type expected
NamedNumber	Mixed	Yes	Named number expected
AuxNamedNum	Blocking	No	Number or external value reference expected
NamedBit	Mixed	Yes	Named bit expected
ElementType	Mixed	Yes	Named type or inclusion clause expected
ElementTypeSuf	Passing	No	
Class	Passing	No	
ClassNumber	Blocking	No	Unsigned number or external value reference expected
Value	Mixed	Yes	Value expected
AuxVal0	Passing	No	
AuxVal1	Blocking	No	Subtype specification or symbol ':' or symbol '.' expected
AuxVal2	Passing	No	
AuxVal11	Blocking	No	Type reference or value reference expected
SpecVal	Mixed	Yes	Subtype specification or symbol ':' expected
BuiltInValue	Passing	No	

Parser	Status	Mode	Error message
BetBraces	Passing	No	
AuxBet1	Passing	No	
AuxBet2	Blocking	No	Subtype specification or symbol ':' or symbol '.' expected
AuxBet3	Passing	No	
AuxBet11	Passing	No	
AuxBet21	Blocking	No	Type reference or value reference expected
AuxNamed	Passing	No	
NamedValue	Mixed	Yes	Named value expected
NamedValSuf	Passing	No	
SubtypeSpec	Mixed	Yes	Left bracket beginning a subtype specification expected
SubtypeValueSet	Mixed	Yes	Subtype value set expected
SubValSetSuf	Mixed	Yes	Symbol '..' or symbol '<' expected
UpperEndValue	Blocking	No	Value or MAX clause expected
InnerTypeSuf	Blocking	No	Keyword COMPONENT or keyword COMPONENTS expected
MultipleTypeConstraints	Blocking	No	Multiple type constraints expected
NamedConstraint	Passing	No	
PresenceConstraint	Passing	No	
SVSAux	Mixed	Yes	Value expected
SVSAux1	Blocking	No	Subtype specification or symbol ':' or symbol '.' expected
SVSAux2	Passing	No	
SVSAux3	Passing	No	
SVSAux11	Blocking	No	Type reference or value reference expected
SVSAux21	Blocking	No	Type or symbol '..' expected

We give now a part of the interface of the lexer, in order to understand the coding of parsers for tokens.

```
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
;;
```

The first parameter of the constructors correspond to the location of the first character of the token in the ASN.1 source file, and the second one id for the concrete syntax of the token, i-e. its characteristic character string. The four first ones are obvious. `BasedNum` correspond to the `basednum` of the grammar and `XString` to `string` (Cf. 6.2). `Symbol` gather all ASN.1:1990 symbols, as ':', '..', '(', '{', etc. `Sentry` is fictive token for private use.

Here is now the code for token parsing:

```
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
;;
```

Notice the absence of `string` or `basednum`, due to their exclusive presence in head of productions.

7.3 A YACC specification

It is easy to obtain a YACC specification from the grammar for implementation (just expand rational operator definitions). That means, as far as YACC accepts this inputs without reporting errors, that the new ASN.1:1990 grammar is also LALR(1)[9] Note that a YACC specification is interesting only if it allows a rather easy abstract-syntax tree building in C. But because the new grammar is LL(1), it is very far from the initial one, and so we lose the intuitive semantics helpful for this aim. That's why it was necessary to employ all the features of the Caml Light language to implement the parser, and among them high-order functions[10]. This functionality allows also on-the-fly macro-processing, and thus to keep a one-pass parser. This feature is not available in most imperative languages, like C. It means more precisely that results need sometimes to be themselves functions which return abstract-syntax nodes, and not directly nodes (for instance in the (sub-)grammar of subtypes). *Maybe* transforming this specification (taking care to let the generated language invariant, of course) without generating conflicts, one will get a grammar suitable for easy abstract-syntax tree building in C (but forgetting macro-processing). Another approach for the reader who wants to develop a software in C interfaced with this Caml Light front-end, is to use some compiler like *Bigloo*, an optimising Caml[11] to C compiler, freely distributed by the INRIA. Cf. [6]. Another possible way is to write in Caml Light a back-end which maps the Caml abstract-syntax tree into a C one.

[9] A LL(1) grammar is not always an LALR(1) one.
[10] Functions may take as argument a function and return a function.
[11] and Scheme!

8 An abstract-syntax tree for ASN.1:1990

We present the abstract-syntax tree designed for ASN.1:1990. We try to extract the maximum information from the parsing process without making calculations with the context, without exploring the produced tree to check out for instance if a referenced type exists, if a subtype is empty, if a value has a correct type, or if a module exists — all things relevant to semantic analysis. Nevertheless, we check out (with *no* computations) the produced subtree in order to get rid of some *semantic* ambiguities, but we do not examine outside of this current subtree (it's a kind of "local-scope syntactic type-checking"!). Figure p.130 sum up all the syntactic constructions that may appear in the structured ASN.1:1990 values (i-e. values specified between braces — see rule 'BetBraces'). Each set is characterised by a pseudo-production generating its elements. For example, considering the set tagged "…"," lower Value "," …" one must understand that its superset is {NamedValue "," ...}$^+$: thus it is the set of non-empty lists of named values *of which at least one is explicitly named*. Each intersection of these sets therefore stress a syntactic ambiguity: we associate to each domain a specific Caml Light constructor, denoting the semantics of the syntactic construction, the best we are able to. This denotation is graphically shown with tags (constructor identifiers) pointing into each domain (for instance `GenOfV` denotes the syntactic domain "…"," lower Value "," …"). Some set elements are boxed: that means that the domain which contains them is finite and that all elements are written in the figure (all the other elements are also enclosed).

The definition of an abstract-syntax tree implies various difficulties.

Firstly, it's obvious that one must interpret the most correctly as possible a semantics given in natural language, with all the well-known risks of ambiguities and non-consistence it carries.

Secondly, it's compulsory to have a very unform naming convention for Caml Light identifiers. For example, integer numbers appear in numerous contexts, semantically different, and we have to generate a distinct identifier for each context. Following the example: `NumCat` denotes the number of a type tag, and `NumBit` qualify the position of a named bit in a *BitString* value. The rule consists in abbreviate the local meaning ("*It's a number, thus* `Num`), then to concatenate the abbreviation of its context ("*...qualifying the position of a named bit, so* `NumBit`.")

Thirdly, type definitions in Caml Light are sometimes afferent to the computation of the tree in itself, which may necessitate temporary nodes, i-e. nodes which don't serve to tree building, and never occur in the final tree.

Finally, the main difficulty is that several definitions are justified by the limits upon the desambiguisation of ASN.1:1990 types *during parsing*: we have ambiguous nodes which denote several semantics. Some subtrees denote an ASN.1:1990 value which has more than one types, after parsing. From the implementation point of view, this leads to constructor identifiers (the nodes) which are the concatenation of the possible type abbreviations (for the subtree of which they are the root). For instance: `BitOctStrV` denotes a value of type *BitString* or *OctetString* — ambiguity only could be removed thanks to their concrete syntax.

We found the following Caml Light type definitions, corresponding to the abstract-syntax tree of ASN.1:1990 in module `ast`.

8.1 Elementary definitions

First we present some basic definitions which correspond essentially to tokens. Note we here distinguish between `valuereference` and
`identifier`, and between `typereference` and `modulereference`.

We also found type `Option` (see 5.6.3). Beware! In the following, we use the term "identifier" in a larger sense than ISO does (`identifier`). In order to avoid ambiguities we flank it with a complement (e.g. "type identifier"), and otherwise it must be taken in the general sense of "string of alphanumeric characters denoting an ASN.1 type, value, module or identifier".

```
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
;;
```

8.2 Auxiliary values

We give the temporary node definitions (Caml Light value constructors) always used to build other nodes. They never settle in the abstract-syntax tree.

```
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` allows the transmission of some tokens of which semantics is not yet available, and that will be found in dependent (calling) parser. `TagMode` serves to point out how to compute a type tag.

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

Taking a look to ASN.1:1990 grammar transformations, we realize that some rules where identified (for they generate the same sub-language), and therefore we have to recover their original different semantics, of which we didn't care about, to construct the tree. It's the case for example for the rule '*ClassNumber*' (Cf. 3.1.3). Its associated subtree is a Caml Light value of type `ClassNum`: it may be a literal integer (`NumCl`) or an integer value exported from another module (`DefValCl`). It's the calling context of '*ClassNumber*' that decides between the two semantics.

8.3 Modules

Here's the nodes corresponding to the ASN.1:1990 module specification, without details about types, subtypes and values (for syntax, see 3.1).

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

The module specification is a triple made of a module identifier (`ModId`) — which references it in a unique way in the ISO tree —, a clause (`Scope`) defining identifier scoping, and a list of type and value definitions (`Def`).

```
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 ...
```

A module identifier is a pair made up of a module identifier (`MRef`) and node list (`ObjIdComp`) from the ISO tree. The scoping clause (`Scope`) gather imported identifiers (`Import`) and exported ones (`Export`). Definitions (`Def`) are of two kinds: type definitions

(`TypeDef`) or value definitions (`ValDef`). A type definition is a pair made up of a type identifier (`TRef`) and a type definition (`Type`), strictly speaking. A value definition is a triple made of a value identifier (`VRef`), a type (`Type`) and value definition (`Value`), strictly speaking.

```
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 ...
```

A node in the ISO tree is qualified

- by a number (`NumObj`).
- or by an external value reference (`EVRObj`), that is to say a value reference exported from another module.
- or by `IdObj`: an explicit node name (`Ident`) specified directly by an ISO subtree number (`NumForm`), or indirectly (in another module: `DefValForm`).
- or by a value identifier or value reference (`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 ...
```

An importation clause (`Import`) is made up of as list of exported identifiers from other modules (`SymMod`). Each element of this list is a pair made up of the identifier list (`Sym`) strictly speaking and of the exporting module identifier.
An exportation clause consists of an identifier list (qualifying a list of entities defined inside the current module: `Sym`).
An identifier (`Sym`) denotes either a value (`SymVal`) or a type (`SymType`).

8.4 Types

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

An ASN.1:1990 type is fully determined by a tag list (`Tag list`), an effective description of the type structure (`Desc`) and a list of sub-typing constraints (`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 ...

An ASN.1:1990 tag (`Tag`) can either be *undefined* (`Undefined`) for any type (*AnyType*) tagging, or a triple composed of a normalised class (`Class`), a reference to the ISO catalogue (`Cat`) and a tagging mode (`TagMode`). This reference may be indirect (`DefValCat`), under the form of an exported value from another 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 ...

The structural description of an ASN.1:1990 type settles on the following type ones: type exported from another module (`DefType`), boolean (`BooleanT`), integer (`IntegerT`), bit string (`BitStrT`), octet string (`OctStrT`), unspecified[12] (`NullT`), ordered set (called also "sequence") of heterogeneous types (`SeqT`), ordered set of homogeneous types (`SeqOfT`), unordered set (also merely called "set") of heterogeneous types (`SetT`), unordered set of homogeneous types (`SetOfT`), choice (`ChoiceT`), selected (`SelectT`), any (`AnyT`), qualifier in the ISO tree (`ObjIdT`), enumerated (`EnumT`), real number (`RealT`), useful (`UsefulT`) and character string (`CharStrT`).

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

A list of named relative numbers (`NamedNum`) is used to qualify integer and enumerated types. Renaming is done thanks to a value identifier (`Ident`), the integer strictly speaking (`AuxNamedNum`) may either be a literal (`NumNum`), or exported from another module (`DefValNum`).

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

A list of named bits (`NamedBit`) is used to qualify bit strings. Renaming is done thanks to a value identifier (`Ident`), the bit strictly speaking (`AuxNamedBit`) may either be a literal (`NumBit`), or exported from another 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 ...
```

The type of ordered sets of types (sequences) is composed of a list of basic types (`ElementType`). A component (or "field") may be of the form:

[12] Some people say (incorrectly because it contains a value): "empty type".

- `Mandatory` That means that one will be obliged to give the value corresponding to this field when defining the sequence value.
- `Optional` In this case we'll can forget the value of this field when defining the sequence value (note that a possible ambiguity here cannot be detected during parsing).
- `Default` The same as for `Optional`, except that if the value of this field is omitted a default value will be employed.
- `Included` The field can *logically* be expanded into the fields of the indicated type, which must thus be a sequence (that cannot be checked out at parsing-time). It's a flat inclusion rule, i-e. included fields are at the same scoping level as the others. The ASN.1:1990 in this case is COMPONENTS OF.

A named type (`NamedType`) is used to qualify the components upper (except if they are included). The ASN.1:1990 norm allows, when there's no *semantic* ambiguity, to omit the named type identifier — giving this way birth to a strange animal that could be christened "anonymous named type". That's why `Ident` is optional in the Caml Light definition.

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

No comment.

8.5 Subtypes

Fundamentally a subtype is a type. What distinguishes subtypes is a *non-empty* list of constraints (`Constraint`). The attentive reader remarked anyway that in the previous subsection (8.4) the `Constraint` type was not defined... That allows a better structuring of the presentation, devoting this subsection to subtypes.

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

A sub-typing constraint is value list (`SubValSet`) of “parent” type.

```
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 ...
```

A value set of a subtype may be:

- `Single` The subtype hence owns one value `Value` of parent type.
- `Contained` The subtype includes values of type `Type` (which must also be a subtype of the same parent type — It cannot be checked out at parsing-time.)
- `Range` The subtype contains integer or real values, between given bounds (`Bound`). A bound can either be strict (`Strict`) or large (`Large`). There exists two pseudo-values specifying respectively the minimal value of the interval (`Min`) and the maximal one (`Max`). Otherwise an explicit value is introduced by means of `EndVal`.
- `Alphabet` The subtype only contains a part of the characters of parent type (which hence must be of type “character string”, modulo tagging).
- `Size` The subtype imposes a restriction upon the length of the values of the parent type (which thus must have a metric).
- `Inner` The subtype owns the values of the structured parent type, complying with constraints (eventually implicit) about their presence.

```
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 ...
```

A constraint may either be simple (`SingleConst`) or multiple (`MultConst`). In the former case, the constraint applies to the whole subtype (the parent type must be an unordered homogeneous set); in the latter, it applies to the fields, and can either be complete (`Full`) or partial (`Partial`). In both situations, we have to supply with a list of optional named constraints (`NamedConst`). notice that the empty list and the *logically absent* elements (i-e. `None` values) correspond to an abbreviate specification which can be semantically ambiguous. A named constraint is the triple with optional fields composed of a value identifier, a constraint and presence clause (`Member`). Following the grammar (Cf.3.5) and abstract-syntax tree building, we see that it's impossible to have simultaneously these three fields "logically absent" — it's just a coding convenience.

8.6 Values

We present the Caml Light types defining ASN.1:1990 value structures. Ambiguities are numerous and as far as they can be removed at parsing-time, we give an abstract-syntax tree the less ambiguous we are able to. We recall that a "structured" value is a value between braces.

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

We give firstly for sake of simplicity the definition corresponding to the two ASN.1:1990 tokens `PLUS-INFINITY` and `MINUS-INFINITY`. See later the reason for its 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
| ...

A value can be referenced in another module that exports it (`DefVal`), a boolean (`BooleanV`), a signed integer (`IntegerV`), unspecified (`NullV`), chosen (`ChoiceV`), instantiated (`AnyV`), character string (`CharStrV`), etc. There exists the value "empty structured value" (`EmptyV`), used in place of empty sets of any kind (in the mathematical sense: i-e. in place of sequence too) and empty bit strings (Cf. *'BuiltInValue'* in 3.3.2).

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

Following grammar transformations 3.3.3 we note that *IntegerValue*, *EnumeratedValue* and *DefinedValue* have (syntactically) in common the production `lower`. This leads us to create an ambiguous node gathering these different possibilities: `IntEnumVRefV`. Notice also that the node `IntegerV` presented just before express the semantics of the sole non-ambiguous syntactic construction of an integer: the signed integer.

| ...
| RealV **of** Infinity
| ...

Transformations presented in 3.3.1, 3.3.3 and 3.3.4 show that the sole productions syntactically non-ambiguous of *RealValue* are these corresponding to tokens `PLUS-INFINITY` and `MINUS-INFINITY`. That justifies a node `RealV`. If we would have been particular, we could have argued that production "`0`" of *RealValue* is ambiguous, because, once transformed into `number` (Cf. 3.3.2), it is merged with the same one of *IntegerValue* in *Value* (Cf. 3.3.3), and its semantics becomes `IntegerV`. This objection is correct: value 0 is the only integer value which can denote a real number in ASN.1:1990, but because it is unique and it's easy to compare to zero in a semantic analyser, we accept to consider it as always being integer.

| ...
| BitOctStrV **of** string
| ...

The step presented in 3.3.2 shows that *BitStringValue* includes the rule *OctetStringValue* (`basednum`). Besides the step saw in 3.3.3 shows that production "{" {$\texttt{lower}_{id}$ "," ... }* "}" is ambiguous and must be merged in *BetBraces*. Therefore we define a node `BitOctStrV` which gather the two possible semantics of a token `basednum`. The semantics of the other ambiguous construction of *BitStringValue* is partially removed in the following presentation of the remaining Caml Light type `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 *)
;;
```

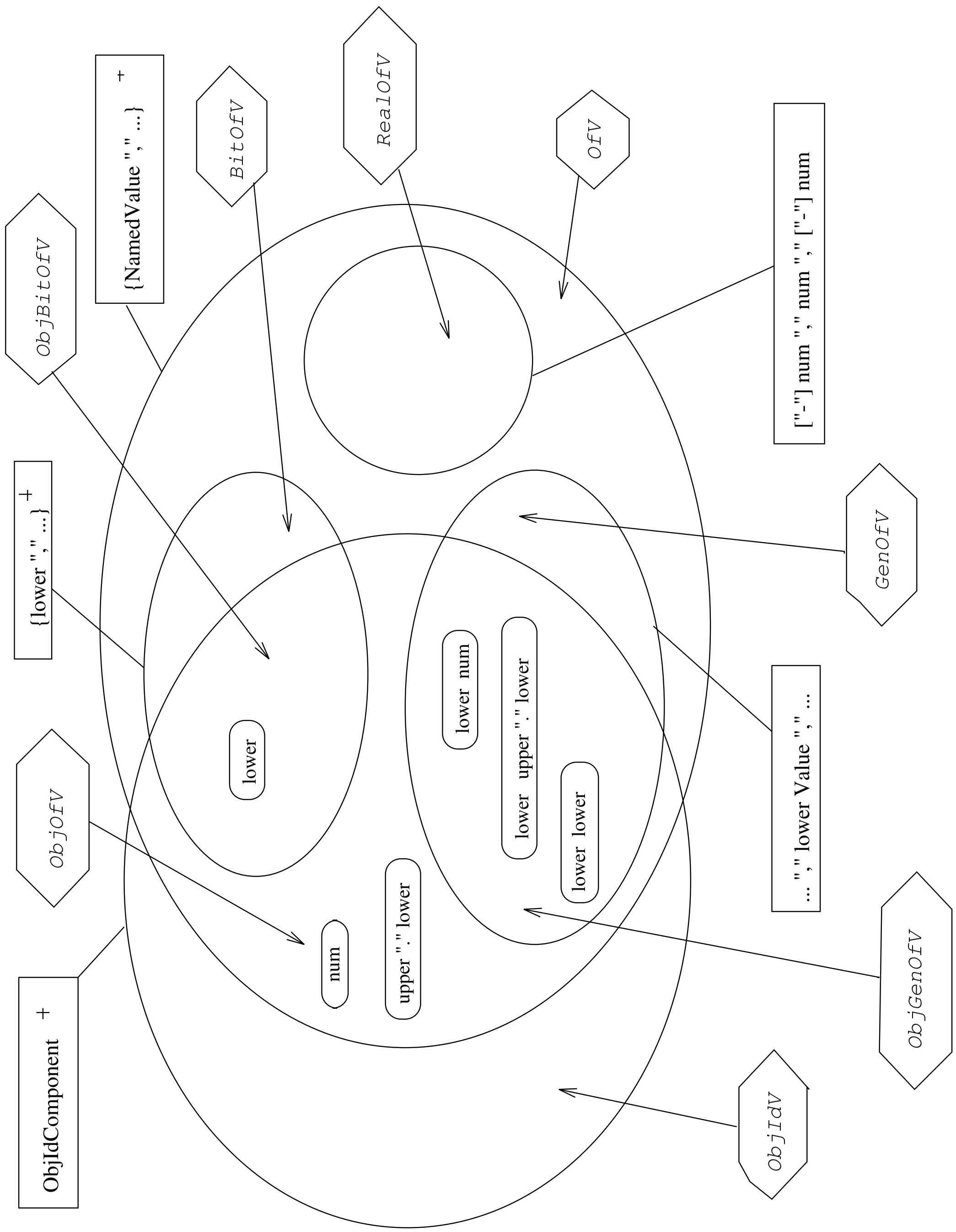

Figure 1: ASN.1:1990 structured values ambiguities

9 ASN.1:1990 macros

ASN.1:1990 includes a construction called "macro" in order to allow user-defined syntax. That is to say that the parser must be able to recognise new value and type notations at run-time, or, with other words, it can extend dynamically the language it accepts. Typically, this makes ASN.1:1990 context-sensitive, i-e. the parsing cannot be achieved without a context (a knowledge about what was defined upper or lower). On the other hand, a macro definition (where the syntax for the new values and types are specified) is basically a grammar extension without any constraint, thus the user can specify ambiguous extensions — remember that it has been proved that no algorithm can check out whether a given grammar is ambiguous or not. Moreover their semantics given in [4] is dark and cumbersome.

The idea for macro handling is to use Caml Light full functionality, that is to say the result of a macro-definition parsing is a pair of specific parsers (functions): one for the new type notations and one for the new value notations. They will be used to parse their corresponding new notations. In other words, we compute functions at run-time that are parsers devoted to new notation recognition. Therefore as far as the result of parsing a macro-definition *is* a parser, our parser should check the *semantics* of the macro-definition... but we have no semantic analyser yet!

We define here a kind of subset of what macros seems have to be, but which has the advantage to be understandable, parsable and also not too limited in order the user doesn't feel bridled. Keep in mind also that this is just a small prototype that could be extended to integrate more features. Many standard macros (like those for the ROSE protocol [3], modulo the restrictions given in 9.8 and 9.5 point 3) are anyway accepted.

For a short introduction and critique of macros, please refer to [9].

9.1 Vocabulary

Actually we use the same vocabulary for macros as for the BNF-like grammar notation we presented in section 1.2, but with the prefix "macro-". For instance, a *macro*-rule is a grammar rule defined in a *macro*-definition.

There are always two special macro-rules `TYPE NOTATION` and `VALUE NOTATION` which are the entry points of the macro-definition: the former defines the new syntax for types, and the latter the new syntax for values. A new type-notation (i-e. a type denoted by means of a syntax defined in a macro) and a new value-notation (i-e. a value denoted by means of a syntax defined in a macro) are respectively called in this document *type instances* and *value instances*.

Beware! In [4] the term "production" is used in the sense of "rule" we defined in 1.2, so be careful when reading the source code of the parser and the error messages...

9.2 Incremental integration

We want to integrate the macro handling to the core[13] parser in an incrementally way. For example we want to keep our core abstract-syntax tree as a strict subtree of the new tree. Another point is that we keep unchanged the lexer. This implies that although the macro-terminal "`"->"`" is legal inside a macro-definition and accepted, the parser will not recognise the corresponding *instance*. Indeed, the lexer actually generated *two* tokens: "`-`" and "`>`", because "`->`" is not an ASN.1:1990 token. The user should have written "`"-" ">"`". In order to relax this constraint, we make the parser accept all kind of symbols which potentially may appear in a macro-definition (like ">", "!", etc.) — the best would be a dynamically extensible lexer, like the parser.

Because of our incremental constraint, we have to leave the lexical convention for local value identifiers inside macro-definitions. Indeed, for historical reasons, it follows the same convention as for type identifiers (Cf. §A.2.8 of [4]), and make therefore impossible the reuse of the core parser for ASN.1:1990 value recognition. Then we impose inside the macro-definitions the same lexical convention as outside. Actually, we identify the token `localvaluereference` with `valuereference`, and *no* longer with `typereference` (Cf. 9.9.2). Note also that the token `macroreference` is a particular case of `typereference`: all its characters must be in upper case. That's why we cannot detect the macro identifiers while lexing, and we let this checking to a possible semantic analyser, identifying `macroreference` with `typereference`.

9.3 Macro-tokens

We decided to remove the macro-tokens[14] `astring` and `"string"`. These two macro-tokens are "defined" respectively in §A.2.7 (and §A.3.10) and §A.3.12 of [4], and their definitions are problematic. For `astring` the difficulty is the same as the one shown in (9.2). For `"string"`, the following example shows the point:

```
MY-MODULE DEFINITIONS ::= BEGIN

PB MACRO ::=
BEGIN
  TYPE  NOTATION ::= string
  VALUE NOTATION ::= value (VALUE BOOLEAN)
END
```

[13] In this document "core" refers to ASN.1:1990 without macros.
[14] That is to say a terminal appearing in the rule 'SymbolDefn' denoting a token in a macro-rule

```
T     ::= TEST this is a string
val T ::= TRUE

END
```

The parser cannot determine when it has to stop consuming tokens (here, the character strings "this", "is", "a", "string") and may therefore "eat" the beginning of a possible following value declaration (here `val`). The worst case is when `T` is replaced by a selection type arbitrarily long. For sake of simplicity, we then propose to remove the macro-token `astring` and to modify the semantics of `"string"`: one will have now to put between inverted commas the denoted string. In our previous sample, we must to write:

```
T     ::= TEST "this is a string"
val T ::= TRUE
```

9.4 One-pass parsing

We want to keep a one-pass parser and it's feasible thanks to high-order parsers (functions) and full functionality of Caml Light. As said before, the result of a macro-definition parsing is a pair of specific parsers: one devoted to type-instance recognition and one to value-instance recognition. They are stored in two different global tables, and when the parser wants to recognise a type definition or a value definition, it then tries first to recognise an *instance* of a macro by calling the available parsers in these tables. If they all fails, then it tries to parse the notation as if it was a core one. Note that our algorithm forces macro-definitions to be *before* instances.

Macro-definitions may use instances of another macro. We keep this possibility, but we forbid mutually recursive definitions because we want to keep a one-pass parsing. Actually we won't check this kind of dependencies: parsing of macro-definitions will simply fail.

9.5 Streams for instance parsing

We want to use the Caml Light stream feature for macro-instance parsing. In order to relax this "constraint", we add the ability of *limited back-tracking*, that is to say if a syntax error occurs inside a right-hand, then the parser tries the following right-hand instead of aborting analysis, as usual (raising `Parse_failure`). If it's the last right-hand of a production, then it reports a failure (not an abortion) to the calling rule (raising `Parse_failure`) that behaves as indicated previously (for more details, see [7]). The important improvement, comparing with the Caml Light stream pattern matching, is that we remove the constraint presented at (5.1).

If we try to sum up now the constraints at the time of writing a macro-definition, we find that:

1. The writer, *as always*, must check by hand that its new notation is not ambiguous (because this checking is in general undecidable). If the writer accepts an ambiguous definition (that can be sometimes very useful as we saw in 5.2), he must remember the order of evaluation of the corresponding stream pattern matching in Caml Light[15].

2. The writer must make sure that there is no left recursive macro-rule, that could make the parser hang up in a loop. This is inherent to our elected method (top-down with limited back-tracking). This point could be automatically detected and solved (using an Arden transformation). For the moment, one must make sure that:

$$\forall \mathrm{A} \in \mathcal{N}, \quad \neg(\mathrm{A} \overset{*}{\Longrightarrow} \mathrm{A}\alpha)$$

3. The writer must make sure that there is no useless production, i-e. a production that generates the empty word (ε) and that is *not* the last of the (macro-)rule. This is due to the stream pattern matching semantics. This could be automatically checked in a further version of the parser (the problem consisting in checking whether ε belongs to a language is decidable), and solved by some transformations upon macro-rules. For the moment, one must make sure:

$$\mathrm{A} \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad \models \quad \forall i \in [1, n],\ \{\varepsilon\} \subseteq \mathcal{P}(\alpha_i) \quad \Longrightarrow i = n$$

   For example, consider the macro BIND for the ROSE protocol [3]. The (macro) empty word `empty` always appears in the first macro-productions, like:

```
Argument ::= empty | "ARGUMENT" Name type(Argument-type)
```

   This way, the parser generated for the instances will always read ε (`empty`) and never try to read the second production. Thus, one must write instead:

```
Argument ::= "ARGUMENT" Name type(Argument-type) | empty
```

9.6 Syntax-error detection

We integrate instance recognition in the core parser adding a pattern at the beginning of the patterns of the functions parsing the ASN.1:1990 types and values. If we put it in the last position (down), the parser would try *first* to analyse a core type (or value) in presence of a macro-instance and probably fail, without having tried to recognise an instance. On the contrary (addition in first position), the parser tries first to analyse an instance, and in case of failure, will naturally try to read a core type (or a value) in the following patterns.

[15] Left-right, top-down

Accordingly, if the writer actually wanted to write an instance but made a syntax error (for example in a core type or value inside a macro-definition), he gets an error notification *as if he had wanted to write a core type (or value)*. This is due of course to the back-tracking method: in general we lose the longest valid prefix property, and thus we lose precision on the error reporting.

9.7 Macro-definition soundness

Exhaustive detection of errors in macro-definitions raise a rather hard difficulty. It should indeed be done a parsing-time whereas it is a semantic diagnostic, and therefore cause problems when we *just* want to build a parser. So, for sake of simplicity, we only detect the following errors:

- Non-definition of a macro-rule.
- Multiple definition of a macro-rule.
- Absence of macro-denotation.

We understand as *denotation* the semantic of an instance, i-e. a type or a value, and the ISO document states that each possible instance must be mapped to only one denotation ([4] §A.3.18). The lack of a denotation is detected when parsing the macro-definition. If we have had a semantic analyser for the core ASN.1:1990 language, it would have been enough to call it to check the macro-definition soundness.

9.8 Macro importation

Because Caml Light has no direct way to export functional values (i-e. making them persistent) and because the result of a macro-definition parsing is a pair of functions (one for type instance parsing and another for value instance parsing), we prefer for sake of simplicity to ignore macro importation. One solution would be to simulate the importation: when a macro is imported, the parser should parse the module containing it in order to get the associated functions (parsers), *but the parser should only look for macros.* The drawback is that we should take care of circular dependencies, which is typically a semantic issue. The advantage would be that it is not too difficult to implement.

9.9 Transformations of the macro grammar

9.9.1 Step 0

We first give the official grammar, after re-structuring it in sections and subsections, and *after the removal of production* `astring` *from rule 'SymbolDefn'* (Cf. 9.3). On the other hand, the note #2 of §A.3.19 [4] tell us that `localvaluereference` in rule 'LocalValueassignement' may be "VALUE". But we decided implicitly from the beginning to parse ASN.1:1990 with *reserved key-words* (following in fact a Technical Corrigenda), and "VALUE" is a key-word.

Therefore, we had to change slightly the rule 'LocalValueassignment' in order to make appear explicitly the key-word "VALUE".

MacroDefinition	$\rightarrow$	`macroreference` MACRO "::=" MacroSubstance
MacroSubstance	$\rightarrow$	BEGIN MacroBody END
	|	`macroreference`
	|	Externalmacroreference
MacroBody	$\rightarrow$	TypeProduction ValueProduction SupportingProductions
Externalmacroreference	$\rightarrow$	`modulereference` "." `macroreference`
TypeProduction	$\rightarrow$	TYPE NOTATION "::=" MacroAlternativeList
ValueProduction	$\rightarrow$	VALUE NOTATION "::=" MacroAlternativeList
SupportingProductions	$\rightarrow$	ProductionList
	|	ε
ProductionList	$\rightarrow$	Production
	|	ProductionList Production
Production	$\rightarrow$	`productionreference` "::=" MacroAlternativeList
MacroAlternativeList	$\rightarrow$	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

9.9.2 Step 1

We take in account the lexical ambiguities which lead us to merge `macroreference`, `productionreference` and `localtypereference`, with `typereference`; and `localvalue-reference` with `valuereference`. When the context allows it, we specify whether the terminal `upper` denotes a macro identifier or a production identifier, with help of a subscript (respectively $\texttt{upper}_{mac}$ and $\texttt{upper}_{prod}$).

MacroDefinition	→	$\texttt{upper}_{mac}$ MACRO "::=" MacroSubstance
MacroSubstance	→	BEGIN MacroBody END
	|	`upper` ["." $\texttt{upper}_{mac}$]
MacroBody	→	TypeProduction ValueProduction [ProductionList]

Global expansion of 'ExternalMacroreference' and then prefix factorisation of 'MacroSubstance'.
Option of 'SupportingProductions' and then global expansion.

TypeProduction	→	TYPE NOTATION "::=" MacroAlternativeList
ValueProduction	→	VALUE NOTATION "::=" MacroAlternativeList

ProductionList	→	Production^+
Production	→	$\texttt{upper}_{prod}$ "::=" MacroAlternativeList
MacroAlternativeList	→	$\{$ MacroAlternative "|" $\ldots\}^+$

Arden of 'ProductionList'.
Arden of 'MacroAlternativeList'.

MacroAlternative	$\rightarrow$	SymbolElement$^+$
SymbolElement	$\rightarrow$	SymbolDefn
	$\mid$	EmbeddedDefinitions
SymbolDefn	$\rightarrow$	`upper`$_{prod}$
	$\mid$	“string”
	$\mid$	“identifier”
	$\mid$	“number”
	$\mid$	“empty”
	$\mid$	“type” [“(” `upper`$_{typ}$ “)”]
	$\mid$	“value” “(” Bind “)”
Bind	$\rightarrow$	[`lower`$_{val}$] MacroType
	$\mid$	VALUE MacroType

Arden of ‘SymbolList’ and global expansion.
Prefix and bifix factorisation of ‘SymbolDefn’ (Creation of ‘bind’.).

EmbeddedDefinitions	$\rightarrow$	“<” EmbeddedDefinition$^+$ “>”
EmbeddedDefinition	$\rightarrow$	`upper`$_{typ}$ “::=” MacroType
	$\mid$	`lower`$_{val}$ MacroType “::=” MacroValue
	$\mid$	VALUE MacroType “::=” MacroValue

Arden of ‘EmbeddedDefinitionList’ and then global expansion.
Global expansion of ‘LocalTypeassignment’.
Global expansion of ‘LocalValueassignment’.

MacroType	$\rightarrow$	Type
MacroValue	$\rightarrow$	Value

Elimination of the production upper$_{typ}$ in the rule *‘MacroType’* because Type $\Longrightarrow$ upper
Elimination of the production lower$_{val}$ in the rule *‘MacroValue’* because Value $\Longrightarrow$ lower

9.9.3 Step 2

MacroDefinition	→	`upper`$_{mac}$ MACRO "::=" MacroSubstance
MacroSubstance	→	BEGIN MacroBody END
	|	`upper` [“.” `upper`$_{mac}$]
MacroBody	→	TypeProduction ValueProduction Production*

Global expansion of ‘ProductionList’.

TypeProduction	→	TYPE NOTATION "::=" { MacroAlternative “|” ...}$^+$
ValueProduction	→	VALUE NOTATION "::=" { MacroAlternative “|” ...}$^+$
Production	→	`upper`$_{prod}$ "::=" { MacroAlternative “|” ...}$^+$

Global expansion of ‘MacroAlternativeList’.

MacroAlternative	→	SymbolElement$^+$
SymbolElement	→	`upper`$_{prod}$
	|	SymbolDefn
	|	“<” EmbeddedDefinition$^+$ “>”
SymbolDefn	→	“string”
	|	“identifier”
	|	“number”
	|	“empty”
	|	“type” [“(” `upper`$_{typ}$ “)”]
	|	“value” “(” Bind “)”
Bind	→	[`lower`$_{val}$] Type
	|	VALUE Type

Partial expansion of the production upper$_{prod}$ of the rule ‘SymbolDefn’ in the rule ‘SymbolElement’.
Global expansion of ‘MacroType’.
Global expansion of ‘EmbeddedDefinitions’.

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

Global expansion of ‘MacroType’ and ‘MacroValue’.

9.9.4 Step 3

MacroDefinition	$\rightarrow$	$\texttt{upper}_{mac}$ MACRO “::=” MacroSubstance
MacroSubstance	$\rightarrow$	BEGIN MacroBody END
	|	$\texttt{upper}$ [“.” $\texttt{upper}_{mac}$]
MacroBody	$\rightarrow$	TypeProduction VALUE NOTATION “::=”
		{ MacroAlternative “|” ... }$^+$ Production*

> Global expansion of ‘ValueProduction’.

TypeProduction	$\rightarrow$	TYPE NOTATION “::=” { MacroAlternative “|” ... }$^+$
Production	$\rightarrow$	$\texttt{upper}_{prod}$ “::=” { MacroAlternative “|” ... }$^+$

> Global expansion of ‘ValueProduction’.

MacroAlternative	$\rightarrow$	SymbolElement$^+$
SymbolElement	$\rightarrow$	$\texttt{upper}_{prod}$
	|	PartElem
PartElem	$\rightarrow$	SymbolDefn
	|	“<” EmbeddedDefinition$^+$ “>”
SymbolDefn	$\rightarrow$	“string”
	|	“identifier”
	|	“number”
	|	“empty”
	|	“type” [“(” $\texttt{upper}_{typ}$ “)”]
	|	“value” “(” Bind “)”
Bind	$\rightarrow$	NamedType
	|	VALUE Type

> Reduction in the rule ‘SymbolElement’ (creation of ‘PartElem’).
> We know that NamedType $\Longrightarrow$ [lower$_{id}$] Type
> Cf. (3.2.3). Thus, rewriting NamedType $\Longrightarrow$ [lower] Type
> we can make a reverse total expansion in the rule ‘Bind’,
> and make appear an occurrence of ‘NamedType’.

EmbeddedDefinition	$\rightarrow$	$\texttt{upper}_{typ}$ “::=” Type
	|	$\texttt{lower}_{val}$ Type “::=” Value
	|	VALUE Type “::=” Value

9.9.5 Step 4

The careful reader noticed the following problem:

$$\mathcal{P}(\text{MacroAlternative}) \cap \mathcal{P}(\text{Production}) = \{\texttt{upper}\}$$

It hence prevents the rule 'MacroBody' of being LL(1). In order to get rid of this difficulty, we now transform this rule as follows:

MacroBody	$\rightarrow$	TypeProduction VALUE NOTATION "::=" MacroSuf
MacroSuf	$\rightarrow$	{ MacroAlternative "|" ... }$^+$ Production*

Reverse total expansion. (Creation of the rule 'MacroSuf'.)

MacroSuf	$\rightarrow$	MacroAlternative ["|" { MacroAlternative "|" ... }$^+$] Production*

MacroSuf	$\rightarrow$	SymbolElement$^+$ ["|" { MacroAlternative "|" ... }$^+$] Production*

Total expansion of 'MacroAlternative'.

MacroSuf	$\rightarrow$	SymbolElement Cont
Cont	$\rightarrow$	SymbolElement* ["|" { MacroAlternative "|" ... }$^+$] Production*

MacroSuf	$\rightarrow$	SymbolElement [Cont]
Cont	$\rightarrow$	SymbolElement$^+$ ["|" { MacroAlternative "|" ... }$^+$] Production*
	|	"|" { MacroAlternative "|" ... }$^+$ Production*
	|	Production$^+$

Option of 'Cont'.

$$
\begin{array}{lcl}
\text{MacroSuf} & \rightarrow & \text{SymbolElement [Cont]} \\
\text{Cont} & \rightarrow & \text{SymbolElement [Cont]} \\
 & | & \text{“|” MacroSuf} \\
 & | & \text{Production}^{+}
\end{array}
$$

We recognise ‘Cont’ and ‘MacroSuf’ (reverse total expansions).

$$
\begin{array}{lcl}
\text{MacroSuf} & \rightarrow & \text{SymbolElement [Cont]} \\
\text{Cont} & \rightarrow & \texttt{upper}\ \text{[Cont]} \\
 & | & \text{PartElem [Cont]} \\
 & | & \text{“|” MacroSuf} \\
 & | & \text{Production Production}^{*}
\end{array}
$$

Total expansion of ‘SymbolElement’ in ‘Cont’.

$$
\begin{array}{lcl}
\text{MacroSuf} & \rightarrow & \text{SymbolElement [Cont]} \\
\text{Cont} & \rightarrow & \text{PartElem [Cont]} \\
 & | & \text{“|” MacroSuf} \\
 & | & \texttt{upper}\ \text{[Cont]} \\
 & | & \texttt{upper}_{prod}\ \text{“::=” \{ MacroAlternative “|” \ldots\}}^{+}\ \text{Production}^{*}
\end{array}
$$

Total expansion of ‘Production’ in ‘Cont’.

$$
\begin{array}{lcl}
\text{MacroSuf} & \rightarrow & \text{SymbolElement [Cont]} \\
\text{Cont} & \rightarrow & \text{PartElem [Cont]} \\
 & | & \text{“|” MacroSuf} \\
 & | & \texttt{upper}\ \text{[ContSuf]} \\
\text{ContSuf} & \rightarrow & \text{Cont} \\
 & | & \text{“::=” MacroSuf}
\end{array}
$$

Prefix factorisation of ‘Cont’. (Creation of the rule ‘ContSuf’.)

9.9.6 Summary

MacroDefinition	$\rightarrow$	`upper`$_{mac}$ MACRO "::=" MacroSubstance
MacroSubstance	$\rightarrow$	BEGIN MacroBody END
	$\mid$	`upper` ["." `upper`$_{mac}$]
MacroBody	$\rightarrow$	TypeProduction VALUE NOTATION "::=" MacroSuf
TypeProduction	$\rightarrow$	TYPE NOTATION "::=" { MacroAlternative "|" ... }$^{+}$
MacroAlternative	$\rightarrow$	SymbolElement^{+}
MacroSuf	$\rightarrow$	SymbolElement [Cont]
Cont	$\rightarrow$	PartElem [Cont]
	$\mid$	"|" MacroSuf
	$\mid$	`upper` [ContSuf]
ContSuf	$\rightarrow$	Cont
	$\mid$	"::=" MacroSuf
SymbolElement	$\rightarrow$	`upper`$_{prod}$
	$\mid$	PartElem
PartElem	$\rightarrow$	SymbolDefn
	$\mid$	"<" EmbeddedDefinition^{+} ">"
SymbolDefn	$\rightarrow$	"string"
	$\mid$	"identifier"
	$\mid$	"number"
	$\mid$	"empty"
	$\mid$	"type" ["(" `upper`$_{typ}$ ")"]
	$\mid$	"value" "(" Bind ")"
Bind	$\rightarrow$	NamedType
	$\mid$	VALUE Type
EmbeddedDefinition	$\rightarrow$	`upper`$_{typ}$ "::=" Type
	$\mid$	`lower`$_{val}$ Type "::=" Value
	$\mid$	VALUE Type "::=" Value

9.10 New complete ASN.1:1990 grammar

We have to graft this new grammar for macros to the core one, [4] didn't telling us explicitly how to do so. Actually, we must integrate the macro definitions among type and value definitions. From the initial form, we get the following transformations:

$$\begin{array}{lcl} \mathit{Assignment} & \rightarrow & \mathtt{upper}_{typ} \text{ “::=” Type} \\ & | & \mathtt{lower}_{val} \text{ Type “::=” Value} \end{array}$$

$$\begin{array}{lcl} \mathit{Assignment} & \rightarrow & \mathtt{upper}_{typ} \text{ “::=” Type} \\ & | & \mathtt{lower}_{val} \text{ Type “::=” Value} \\ & | & \text{MacroDefinition} \end{array}$$

$$\begin{array}{lcl} \mathit{Assignment} & \rightarrow & \mathtt{upper}_{typ} \text{ “::=” Type} \\ & | & \mathtt{lower}_{val} \text{ Type “::=” Value} \\ & | & \mathtt{upper}_{mac} \text{ MACRO “::=” MacroSubstance} \end{array}$$

Global expansion of ‘MacroDefinition’.

$$\begin{array}{lcl} \mathit{Assignment} & \rightarrow & \mathtt{upper} \text{ AssSuf} \\ & | & \mathtt{lower}_{val} \text{ Type “::=” Value} \\ \text{AssSuf} & \rightarrow & \text{MACRO “::=” MacroSubstance} \\ & | & \text{“::=” Type} \end{array}$$

Prefix factorisation of ‘MacroDefinition’.

Finally, the new complete grammar of ASN.1:1990 is:

MODULES

ModuleDefinition	$\rightarrow$	ModuleIdentifier DEFINITIONS [TagDefault TAGS] "::=" BEGIN [ModuleBody] END

ModuleIdentifier	$\rightarrow$	$\texttt{upper}_{mod}$ ["{" ObjIdComponent^{+} "}"]
ObjIdComponent	$\rightarrow$	`number`
	$\mid$	$\texttt{upper}_{mod}$ "." $\texttt{lower}_{val}$
	$\mid$	`lower` ["(" ClassNumber ")"]

TagDefault	$\rightarrow$	EXPLICIT
	$\mid$	IMPLICIT

ModuleBody	$\rightarrow$	[Exports] [Imports] Assignment^{+}
Exports	$\rightarrow$	EXPORTS {Symbol "," ...}* ";"
Imports	$\rightarrow$	IMPORTS SymbolsFromModule* ";"
SymbolsFromModule	$\rightarrow$	{Symbol "," ...}$^{+}$ FROM ModuleIdentifier
Symbol	$\rightarrow$	$\texttt{upper}_{typ}$
	$\mid$	$\texttt{lower}_{val}$

Assignment	$\rightarrow$	`upper` AssSuf
	$\mid$	$\texttt{lower}_{val}$ Type "::=" Value
AssSuf	$\rightarrow$	MACRO "::=" MacroSubstance
	$\mid$	"::=" Type

TYPES

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

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

NamedType	→	`lower` [“<”] 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`$_{val}$

VALUES

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

<u>*BuiltInValue*</u>	$\rightarrow$	TRUE
	|	FALSE
	|	PLUS-INFINITY
	|	MINUS-INFINITY
	|	`basednum`
	|	`string`
	|	"{" [BetBraces] "}"

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

SUBTYPES

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	→	$\texttt{lower}_{id}$ [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	→	$\texttt{upper}_{typ}$ SubtypeSpec* “:” SVSAux
	|	$\texttt{lower}_{val}$ [SubValSetSuf]
SVSAux21	→	Type “:” SVSAux
	|	“..” [“<”] UpperEndValue

MACROS

<u>MacroSubstance</u>	$\rightarrow$	BEGIN MacroBody END
	$\mid$	`upper` [“.” `upper`$_{mac}$]
MacroBody	$\rightarrow$	TypeProduction VALUE NOTATION “::=” MacroSuf
TypeProduction	$\rightarrow$	TYPE NOTATION “::=” { MacroAlternative “$\vert$” ...}$^+$
MacroAlternative	$\rightarrow$	SymbolElement$^+$
MacroSuf	$\rightarrow$	SymbolElement [Cont]
Cont	$\rightarrow$	PartElem [Cont]
	$\mid$	“$\vert$” MacroSuf
	$\mid$	`upper` [ContSuf]
ContSuf	$\rightarrow$	Cont
	$\mid$	“::=” MacroSuf
SymbolElement	$\rightarrow$	`upper`$_{prod}$
	$\mid$	PartElem
PartElem	$\rightarrow$	SymbolDefn
	$\mid$	“<” EmbeddedDefinition$^+$ “>”
SymbolDefn	$\rightarrow$	“string”
	$\mid$	“identifier”
	$\mid$	“number”
	$\mid$	“empty”
	$\mid$	“type” [“(” `upper`$_{typ}$ “)”]
	$\mid$	“value” “(” Bind “)”
Bind	$\rightarrow$	NamedType
	$\mid$	VALUE Type
EmbeddedDefinition	$\rightarrow$	`upper`$_{typ}$ “::=” Type
	$\mid$	`lower`$_{val}$ Type “::=” Value
	$\mid$	VALUE Type “::=” Value

9.11 Checking the LL(1) property of the extended grammar

We're not going to check the LL(1) property of the whole extended grammar (i-e. including macros) from scratch, but do it incrementally, from the results given in (4).

9.11.1 Equation P1

It's obvious that the macro sub-grammar doesn't own any left recursion.

9.11.2 Equation P2

It's easy to check that the intersection of the first functions ($\mathcal{P}$) of each alternative is empty.

9.11.3 Equation P3

Before starting, note that the modification of the rule *'Assignment'* doesn't generate new constraints. For the macro sub-grammar:

Rule	Constraints
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	$\{\text{“(”}\} \cap \mathcal{S}(\text{SymbolDefn}) = \emptyset$

Let:

$$
\begin{array}{ll}
(1) & \{\text{“.”}\} \cap \mathcal{S}(\text{MacroSubstance}) = \emptyset \\
(2) & \{\text{“|”}\} \cap \mathcal{S}(\text{TypeProduction}) = \emptyset \\
(3) & \mathcal{P}(\text{SymbolElement}) \cap \mathcal{S}(\text{MacroAlternative}) = \emptyset \\
(4) & \mathcal{P}(\text{Cont}) \cap \mathcal{S}(\text{MacroSuf}) = \emptyset \\
(5) & \mathcal{P}(\text{Cont}) \cap \mathcal{S}(\text{Cont}) = \emptyset \\
(6) & \mathcal{P}(\text{ContSuf}) \cap \mathcal{S}(\text{Cont}) = \emptyset \\
(7) & \mathcal{P}(\text{EmbeddedDefinition}) \cap \{\text{“>”}\} = \emptyset \\
(8) & \{\text{“(”}\} \cap \mathcal{S}(\text{SymbolDefn}) = \emptyset
\end{array}
$$

Let's compute the first functions ($\mathcal{P}$):

$$
\begin{array}{lcl}
\mathcal{P}(\text{SymbolElement}) & = & \{\ \texttt{upper}, \text{“}<\text{”}, \text{“}\texttt{string}\text{”}, \text{“}\texttt{identifier}\text{”}, \text{“}\texttt{number}\text{”}, \\
& & \text{“}\texttt{empty}\text{”}, \text{“}\texttt{type}\text{”}, \text{“}\texttt{value}\text{”}\ \} \\
\\
\mathcal{P}(\text{Cont}) & = & \{\ \texttt{upper}, \text{“}<\text{”}, \text{“}\texttt{string}\text{”}, \text{“}\texttt{identifier}\text{”}, \text{“}\texttt{number}\text{”}, \\
& & \text{“}\texttt{empty}\text{”}, \text{“}\texttt{type}\text{”}, \text{“}\texttt{value}\text{”}, \text{“}|\text{”}\ \} \\
\\
\mathcal{P}(\text{ContSuf}) & = & \{\ \texttt{upper}, \text{“}<\text{”}, \text{“}\texttt{string}\text{”}, \text{“}\texttt{identifier}\text{”}, \text{“}\texttt{number}\text{”}, \\
& & \text{“}\texttt{empty}\text{”}, \text{“}\texttt{type}\text{”}, \text{“}\texttt{value}\text{”}, \text{“}|\text{”}, \text{“}::=\text{”}\ \} \\
\\
\mathcal{P}(\text{EmbeddedDefinition}) & = & \{\ \texttt{upper}, \texttt{lower}, \text{VALUE}\ \}
\end{array}
$$

And the following functions ($\mathcal{S}$):

$$
\begin{array}{lcl}
\mathcal{S}(\text{MacroSubstance}) & = & \mathcal{S}(\text{AssSuf}) \\
& = & \mathcal{S}(\text{Assignment}) \\
& = & \{\ \text{END}, \texttt{upper}, \texttt{lower}\ \} \\
\\
\mathcal{S}(\text{TypeProduction}) & = & \{\ \text{VALUE}\ \} \\
\\
\mathcal{S}(\text{MacroAlternative}) & = & \{\text{“}|\text{”}\} \cup \mathcal{S}(\text{TypeProduction}) \\
& = & \{\ \text{VALUE}, \text{“}|\text{”}\ \} \\
\\
\mathcal{S}(\text{ContSuf}) & = & \mathcal{S}(\text{Cont}) \\
\\
\mathcal{S}(\text{MacroSuf}) & = & \mathcal{S}(\text{MacroBody}) \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{ContSuf}) \\
& = & \{\text{END}\} \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{ContSuf}) \\
& = & \{\text{END}\} \cup \mathcal{S}(\text{Cont}) \\
\\
\mathcal{S}(\text{Cont}) & = & \mathcal{S}(\text{MacroSuf}) \cup \mathcal{S}(\text{ContSuf}) \\
& = & \mathcal{S}(\text{MacroSuf}) \cup \mathcal{S}(\text{Cont}) \\
& = & \mathcal{S}(\text{MacroSuf}) \\
\\
\mathcal{S}(\text{SymbolDefn}) & = & \mathcal{S}(\text{PartElem}) \\
& = & \mathcal{P}(\text{Cont}) \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{SymbolElement}) \\
& = & \mathcal{P}(\text{Cont}) \cup \mathcal{S}(\text{Cont}) \cup \mathcal{S}(\text{MacroAlternative}) \cup \mathcal{S}(\text{MacroSuf}) \\
& = & \mathcal{S}(\text{MacroSuf}) \cup \{\text{VALUE}, \text{“}|\text{”}, \texttt{upper}, \text{“}<\text{”}, \text{“}\texttt{string}\text{”}, \\
& & \text{“}\texttt{identifier}\text{”}, \text{“}\texttt{number}\text{”}, \text{“}\texttt{empty}\text{”}, \text{“}\texttt{type}\text{”}, \text{“}\texttt{value}\text{”}\ \}
\end{array}
$$

Hence:

$$\mathcal{S}(\text{MacroSuf}) = \mathcal{S}(\text{Cont}) = \mathcal{S}(\text{ContSuf}) = \{\text{END}\}$$

And:

$$\begin{array}{rcl}\mathcal{S}(\text{SymbolDefn}) & = & \{ \text{ END, VALUE, ``|'', } \texttt{upper}\text{, ``<'', ``}\texttt{string}\text{'',} \\ & & \text{``}\texttt{identifier}\text{'', ``}\texttt{number}\text{'', ``}\texttt{empty}\text{'', ``}\texttt{type}\text{'', ``}\texttt{value}\text{'' } \}\end{array}$$

Then the system (1)-(8) is verified.

We have now to make sure that the nonterminal appearing in the macro sub-grammar *and* in the core grammar don't induce sets $\mathcal{S}$ which invalidate the LL(1) property of the core grammar. The nonterminal to examine in this case are 'NamedType', 'Type' and 'Value'.

We had:

$$\mathcal{S}(\text{NamedType}) = \{ \text{ ``,'', ``\}'', OPTIONAL, DEFAULT } \}$$

Now:

$$\begin{array}{rcl}\mathcal{S}(\text{NamedType}) & = & \mathcal{S}(\text{Bind}) \cup \{\text{``,'', ``\}'', OPTIONAL, DEFAULT}\} \\ & = & \{ \text{ ``)'', ``,'', ``\}'', OPTIONAL, DEFAULT } \}\end{array}$$

Let's consider now the places where $\mathcal{S}$(NamedType) is used. It's used to check equations (18) and (19) given in 4.2.3, and, on the other hand, to compute $\mathcal{S}$(AuxType), which itself is used only to compute $\mathcal{S}$(Type). Equations (18) and (19) are still verified and $\mathcal{S}$(Type) is still invariant because it contained yet ")".

The occurrences of 'Type' and 'Value' in the macro sub-grammar introduce in $\mathcal{S}$(Type) and $\mathcal{S}$(Value) the (terminal) symbol ">"; but this one doesn't exist in the core grammar, thus it cannot interfere in the calculi of (4.2.3).

Conclusion The extended grammar is LL(1).

9.12 Extended abstract-syntax tree

Here we are going to explain why it is necessary to extend the abstract-syntax tree in order to process macros and how to do it easily and incrementally. The problem can be divided into two parts not connected: type instances and value instances.

9.12.1 Type instances

The sixth paragraph of section A.1 in [4] says that a type instance may depend on a value instance. Consider indeed the following example:

```
SAMPLE DEFINITIONS ::=
BEGIN

TEST MACRO ::=
BEGIN
 TYPE  NOTATION ::= empty
 VALUE NOTATION ::= value (VALUE BOOLEAN) | value (VALUE INTEGER)
END

T ::= TEST

END
```

What is the type of `T`? Answer: it depends. If a value instance would appear in the module then `T` could be either the boolean type or the integer type. In absence of such an instance, we *must* reject it as being incorrectly defined...[16] At this moment arrives another time the sixth paragraph of section A.1 in [4], telling us that we "should interpret" a type instance as a choice type ("*[...] the use of the new type notation is similar to a CHOICE [...].*"). Therefore in our last sample we are lead to think that `T` would be equivalent (in a sense that is still to be defined) to:

```
T ::= CHOICE { field-0 BOOLEAN,
               field-1 INTEGER }
```

Nevertheless, the possible types for a type instance (i-e. the different fields of the associated choice type) may contain local identifiers inside the macro-definition. For example:

[16]Beware! `T` is not even the NULL type.

```
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
```

In this case one must produce a *type closure*, that is to say a pair made up of a (choice) type and an environment (i-e. a set of local definitions to the macro-definitions, and which would contain here `LT1` and `LT2` definitions). We decide hence that *any type instance is equivalent to such a closure.* In order to take in account this principle, we have to extend the core abstract-syntax tree with a node "type closure", i-e. an additional constructor `TClos` for the Caml Light type `Desc`:

```
type Spec = ...
and ...
and Desc = ...
         | TClos of Desc * Env
and Env == Def list
and ...
;;
```

Cf. Annex.

9.12.2 Value instances

The previous section devoted to type instances lead us naturally to consider value instances as *value closures*, that is to say a pair made up of a value and an environment containing local definitions to the macro-definition. More precisely, these values are implicit choice values, paired with an environment. In order to realise that we must add a node to the core abstract-syntax tree, i-e. a constructor `VClos` to the Caml Light type `Value`:

```
type Spec = ...
and ...
and Value = ...
          | VClos of Value * Env
and ...
;;
```

Cf. Annex.

9.13 Changes to the core parser

We present here the entire list of the modifications to be done on the core parser in order to "interface" it with the parser devoted to macros (presented in 9.14).

We saw that the abstract-syntax tree doesn't keep trace of macro-definitions (only of instances), but we always must to return a Caml Light value of type `Def` for each ASN.1:1990 definition parsed, therefore the solution is to use the (polymorphic) optional type `Option` where we were used to return a value of type `Def`. We also define an auxiliary function `useful` (Cf. 9.15) which extract from a list of optional values the useful ones (`None` correspond to a macro-definition, and `Some` to a core type or a core value). This way the parser `moduleBody` must take in account these optional values:

```
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 ...
```

We also must change the function `assignment` in order to handle the new extended grammar:

```
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 ...
```

Idem for `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 ...
```

The reader will note the first pattern of the function `afterUpper`, which calls the type instance parser (`macroTypeInstance` — Cf. 9.15). The call must be in the first pattern because we always try to read first a macro-instance before a core value or a core type (Cf. 9.4). Its first argument `type_notations` is a hash-table mapping yet known macro identifiers to the parsers of their value instances. Its second argument `id` is interesting: it corresponds to an identifier which may be a macro name and it is necessarily passed as argument to `afterUpper` because *it appears inside the stream pattern* (Cf. 5.4). This shows explicitly the context-sensitive dependence of the language.

For the `xValue` function it's enough to add a first pattern in which we try to read a value instance:

```
and ...
and xValue mode = function
  [⟨ (macroValueInstance !value_notations) v ⟩]
  → v
| ...
and ...
;;
```

The function `macroValueInstance` tries to recognise a value instance, and its argument `value_notations` is the list of the value instance parsers, of which (macro-)definitions were previously find.

9.14 The macro parser

```
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)
```

The function `macroSubstance` reads a macro-definition. It returns a function which takes as argument the name of the macro, read by the parser `assSuf` (Cf. 9.13). We get back a pair `mt` made up of a list of parsers `tp` for type instances, and a list of parsers for value instances. The first element of the Caml Light sequence checks macro-rule completion, and clears the macro-rule table. The second checks whether at least one denotation exists for each macro-instance. The third creates and stocks the parser for type instances. The fourth element do the same for value instances. The fifth clears the top-level reference containing (temporarily) the list of all possible types for a value instance. The sixth (the last, that is: the returned value) is the optional value `None`, reflecting the fact that it was a macro-definition. This `None` will be removed at the level of function `moduleBody` thanks to the auxiliary function `useful` (see 9.13). This is due to the fact that macro-definitions don't let any trace in the abstract-syntax tree (because they are *used* to build this tree).

```
| [⟨ 'Upper (_, up);
    (option (function [⟨ 'Symbol (_, "."); term_macro extMacId ⟩] → extMacId)) macOpt ⟩]
  → (fun _ → None)
```

Here we ignore macro importations (Cf. 9.8).

```
| [⟨ 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
```

The parser `macroBody` returns the pair made up of the lists of parsers respectively for type instances and value instances.

```
and typeProduction = function
  [⟨ 'Keyword (_, "TYPE"); (term_kwd "NOTATION") _;
    (term_sym "::=") _; (list_plus symbolList "|" Abort) rhs ⟩]
  → rhs
```

The parser `typeProduction` recognises and returns a list a 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], [ ]))
```

The parser `prod_call` records, in a macro-production, the call of a macro-rule associated to the identifier `id`. If this latter corresponds to a macro-rule yet declared, then a reference to it is returned; otherwise we create an "empty" parser for a macro-rule (waiting for the forward definition) and we return a reference to it. Cf. (9.15). Notice that the use of these references allows recursive macro-rule definitions.

```
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; ([ ], [ ]))
```

The parser `prod_decl` records a yet known macro-rule.

```
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
| [< 'Lower (_, "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) _ >] → [ ]))))
| [< 'Lower (_, "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
;;
```

9.15 Auxiliary module for macro processing

```
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 "gen";;

#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 pld =
  try
    match find prods pld with
      Call r → Some r
    | Decl r → Some r
  with Not_found → let r = ref (NTerm (fun cnt strm → ([ ], cnt, strm)))
                   in add prods pld (Call r); Some r
;;

let prod_decl pld rhs =
  let p = ref (NTerm (prod rhs))
  in try
       match find prods pld with
         Call r → r := !p;
                  remove prods pld;
                  add prods pld (Decl r)
       | Decl _ → failwith ("Multiple declaration of macro-production '" ^ pld ^ "'")
     with Not_found → add prods pld (Decl p)
;;
```

```
let macroTypeInstance htbl pld =
```

```
  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
;;
```

Annex

ASN.1:1990 lexer source code

First we recall the interface of the Caml Light module `lexer`, partially presented in (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;;
```

We now give the implementation of the module `lexer`.

```
#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" ]
;;

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.1:1990 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.1:1990 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.1:1990 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.1:1990 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
  | [< '(`1`..`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 + !pos) >]
;;
```

Some examples without macros

We give some examples in order to show the output abstract-syntax tree. For sake of clarity, the presentation is done as if we were at the Caml Light top-level loop, during a normal session. *Please read first the README file in the Asno'90 distribution in order to know precisely how to install and run the software.* After entering the system, type:

#include "top";;

We suppose here that we have a ASN.1:1990 source file `ex1.asn` which contents is an extract of the CMIP protocol:

```
-- 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
```

Now we run the parsing and get the abstract-syntax tree:

```
#analyse "ex1.asn";;
- : Spec list Option
  = Some [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) ], [ ])) ]) ]
```

Another example:

```
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
```

Now we run the parsing and get the abstract-syntax tree:

```
#analyse "ex2.asn";;
- : Spec list Option
  = Some
     [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)) ])) ] ])) ])) ]
```

A last one:

```
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
```

Now we run the parsing and get the abstract-syntax tree:

```
#analyse "ex3.asn";;
- : Spec list Option
  = Some [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") ]) ] ])) ]) ]
```

An example with a macro

The example given in §E.3 of [4]:

```
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 Option*
  =
  Some
   [*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) ])) ])) ]) ]
```

References

[1] A. Aho and J. Ullman. *Theory of parsing, Translation and Compiling*, vol. 1 (Parsing) of *Automatic Computation*. Prentice Hall, 1972, ch. 2.6.3, p. 199.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers : Principles, Techniques and Tools.* Addison-Wesley Publishing Company, 1986.

[3] ISO/IEC. *Information processing systems – Text communications – Remote Operations – Part 1: Model, notation and service definition*, first ed., November 1989. Reference ISO/IEC 9072-1.

[4] ISO/IEC. *Information technology – Open Systems Interconnection – Specification of Abstract Syntax Notation One (ASN.1)*, second ed., December 1990. Reference ISO/IEC 8824.

[5] X. Leroy. *The Caml Light system, release 0.7 – Documentation and user's manual.* INRIA-Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France, January 1995.

[6] M. Serrano and P. Weis. 1+1=1 : an Optimizing Caml Compiler. In *ACM-SIGPLAN Workshop on ML and its Applications* (25-26 Juin 1994). Orlando, Florida (USA).

[7] M. Mauny and D. de Rauglaudre. Parsers in ml. In *Proceedings of the ACM International Conference on Lisp and Functional Programming* (San Francisco, USA, 1992).

[8] P. Weis and X. Leroy. *Le langage Caml.* IIA. InterÉditions, France, 1993.

[9] D. Steedman. *Abstract Syntax Notation One (ASN.1) - The Tutorial and Reference.* Technology Appraisals, 1990, ch. 5 Macros, pp. 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