HI-21/8341 **Département TIEM**

DESCRIPTION DU LANGAGE Io DE SPECIFICATION DES ENTREES-SORTIES DE Descartes

JL.DORMOY

SYNTHESE :

Nous présentons ici un langage de spécification des entrées-sorties pour Descartes que nous avons baptisé Io. Io sera également le nom du composant de Descartes qui aura pour tâche, partant de telles spécifications, de générer les programmes qui lisent ou écrivent les données dans des fichiers. Cette tâche n'est en effet pas assumée par Descartes à travers ses trois composants Cogito, Ergo et Sum.

A l'instar du composant Ogre, qui génère les programmes accédant à des bases de données, une spécification en Io comporte trois parties. Tout d'abord le modèle conceptuel des données est décrit. Ce modèle est commun à Descartes, Ogre et Io, mais certains aspects ont été ajoutés pour les besoins de Io, notamment la notion de *portée* et une définition de *l'égalité entre entités*. Ensuite, la syntaxe des données à lire ou à écrire est fournie. Celle-ci est très classiquement décrite par une bnf, aidée d'une grammaire lexicale. Enfin, des liens entre syntaxe et modèle conceptuel sont décrits. On a voulu que la spécification reste aussi pauvre et concise que possible, aussi beaucoup de liens restent-ils implicites, et devront être retrouvés par le système Io. Tous ces éléments sont présentés ici.

Les idées de Io proviennent d'un des composants de Shal, qui devait assurer la traduction entre textes informatiques. Ce composant devait générer des programmes permettant de traduire un texte informatique représenté dans un fichier selon une certaine syntaxe dans une autre syntaxe, mais avec exactement le même "contenu". Io reprend pour l'essentiel ce qui avait été fait à l'époque, sinon que le "traducteur" de Shal est coupé en deux parties : l'une pour *lire* les données, l'autre pour les *écrire*.

En définitive, le système Io comporte trois parties : un premier composant, commun aux entrées et aux sorties, complète les liens fournis entre syntaxe et modèle conceptuel (nous l'appelons IoLink). Les deux autres composants génèrent les programmes, l'un pour les entrées (IoIn), l'autre pour les sorties (IoOut). La spécification complète du système Io n'est pas encore écrite, et

fera l'objet d'un autre document. Cependant, nous décrivons ici le schéma général des connaissances que doit contenir IoLink. Nous pouvons enfin mentionner que les programmes générés par Io le seront dans le langage ErgoAlg de Descartes.

1 Présentation générale de Io

Dans les trois composants principaux de Descartes, Cogito, Ergo et Sum, les entrées-sorties ne sont pas prises en compte. A l'entrée de Descartes, les entrées sont spécifiées implicitement par la connaissance des données (représentées par une partie du modèle conceptuel) dont doit partir le programme. Les sorties, quant à elles, peuvent être tout ou partie du modèle conceptuel "calculé" par le programme Descartes.

Pourtant, lorsque le programme aura été généré, il faudra bien qu'il "lise" les données quelque part. Celles-ci peuvent se trouver en plusieurs "lieux", principalement trois. Elles peuvent être dans une base de données. Dans ce cas, le composant Ogre [Lagrange 92] intervient, encore que Ogre se limite actuellement aux bases de données relationnelles, et à la lecture. Sinon, les données peuvent être présentes dans les structures de données d'un autre programme en activité. Un composant de Descartes, qui n'est pas encore défini, sera nécessaire pour aller "chercher" les données dans ce cas. Enfin, les données peuvent être, c'est le cas le plus classique, dans un fichier. C'est là que Io entre en scène.

Nous présentons donc ici un langage de spécification des entrées-sorties, que nous avons baptisé Io[1], qui permet de spécifier les données à lire et/ou à écrire dans un fichier. Ce langage comporte trois parties. Les données sont toujours "conceptuellement" décrites par un modèle conceptuel, qui est commun à Descartes, Ogre et Io. Certains nouveaux aspects sont cependant ajoutés pour les traitements spécifiques à Io, notamment la notion de *portée*, élargissant la définition de *l'égalité entre entités* limitée dans Ogre aux clés et aux entités faibles. Il va falloir de plus spécifier la syntaxe à laquelle obéissent les données dans le fichier, et les liens entre la syntaxe et le modèle conceptuel des données. La syntaxe est spécifiée, fort classiquement, par une bnf[2]. Les liens sont spécifiés dans un nouveau langage que nous définissons ici.

Le *système* Io devra donc partir d'une telle spécification, et générer un programme qui sache lire les données mémorisées dans un fichier selon la syntaxe spécifiée et faire passer ces données dans la mémoire centrale de l'ordinateur selon une structure de données ad hoc (entrées), ou l'inverse (sorties). Io comporte donc deux parties distinctes : l'une pour les entrées, l'autre pour les sorties. Ces deux parties ont un composant en commun, qui complète les liens fournis entre syntaxe et modèle conceptuel. Nous avons donc trois composants : IoLink, pour compléter les liens entre syntaxe et modèle conceptuel, IoIn pour la génération des programmes lisant les données, et IoOut pour les programmes les écrivant.

L'essentiel des idées exposées ici proviennent de Shal [Dormoy 91]. Nous avions mis en œuvre dans ce système un composant chargé de traduire des textes informatiques ayant un contenu unique, mais représenté sous différentes syntaxes. Io reprend les

---

[1] Io est une célèbre génisse bien connue des cruciverbistes, un satellite de Jupiter où existe le plus grand volcan connu du système solaire, autour duquel d'ailleurs l'ordinateur Hal est devenu fou dans *2001 L'odyssée de l'espace*, et enfin un acronyme pour "Input/Output".

[2] bnf : Backus-Naur Form, ou Backus Normal Form, du nom des créateurs du langage Fortran.

idées alors mises en œuvre, en cassant le processus de traduction en deux étapes : d'abord lecture du texte, avec mémorisation en mémoire centrale, puis écriture à partir de cette mémorisation.

Nous ne décrivons ici que le langage de Io. Le § 2 donne quelques exemples de données que Io doit savoir lire ou écrire. On verra que nous visons large : Io doit savoir lire et écrire la plupart des langages informatiques usuels. Cela sera particulièrement utile pour réaliser les multiples "analyseurs syntaxiques" dont on aimerait disposer pour Descartes lui-même. Le § 3 rappelle les éléments du modèle conceptuel de Descartes utiles ici, et décrit en les motivant les ajouts que nous y avons apportés. Le § 4 rappelle ce qu'est une bnf. Le § 5 présente comment spécifier les liens entre bnf et modèle conceptuel. Le § 6 décrit le composant IoLink du système Io, qui complète les liens spécifiés entre une syntaxe et un modèle conceptuel. Le § 7 conclut rapidement.

Enfin, les composants IoIn et IoOut seront définis dans un autre document.

2 Exemples

2.1 Ambitions de Io

Nous entendons que Io sache lire et écrire une large gamme de "données". En général, lorsque l'on parle de données, on pense à des listes de nombres ou de chaînes de caractères bien tassées que seul un ordinateur saura lire une fois qu'on aura programmé les entrées-sorties. Ces instructions d'entrées-sorties contiennent sous une certaine forme toute la connaissance rendue implicite par la représentation choisie.

```
1250132001003
    Toto    Fulgence    Ulysse  68  1
1491045113012
    Lulu    Achille     Carolus 42  2
1720375014276
    Zonzon  Hector              20  0
```

Exemple 1 : Des données implicites et compactées

On pourra ainsi avoir les données de l'exemple 1, qui représentent des individus reconnus par leurs numéros de Sécurité Sociale, et pour chaque individu ses nom, prénoms, âge et nombre d'enfants.

Une autre représentation de ces données explicite un peu plus les relations existant entre les quantités ou qualités mentionnées. On trouvera ce genre de représentation dans une base de faits, un réseau sémantique, un langage objet, ou (implicitement) dans une base de données. L'exemple 2 reprend l'exemple 1 sous cette nouvelle forme.

```
$1  Nom Toto
```

```
$1  NumeroDeSecuriteSociale 1250132001003
$1  Prenom Fulgence
$1  Prenom Ulysse
$1  Age 68
$1  NombreDEnfants 1

$2  Nom Lulu
$2  NumeroDeSecuriteSociale 1491045113012
$2  Prenom Achille
$2  Prenom Carolus
$2  Age 42
$2  NombreDEnfants 2

$3  Nom Zonzon
$3  NumeroDeSecuriteSociale 1720375014276
$3  Prenom Hector
$3  Age 20
$3  NombreDEnfants 0
```

Exemple 2 : Des données par attribut-valeur

Cependant, il s'avère que ces données ont peu de structure interne. Ce ne sont vraiment que des *valeurs d'attributs d'objets* (d'objets *informatiques*, bien sûr, représentant ici des individus).

Un autre genre de "données" est constitué par ce que nous appelons les textes informatiques, par exemple les programmes. Ainsi, un compilateur (ou un synthétiseur de programme) lira ou écrira un des textes des exemples 3, 4 ou 5.

```
function fact(n : integer) : integer;

begin
    if n=0 then fact:=1
    else fact:=n*fact(n-1)
end;
```

Exemple 3 : Fonction factorielle *en Pascal*

$$\forall\ n \in N \quad n>2 \Rightarrow [x^n + y^n = z^n \Rightarrow x=0 \text{ et } y=0 \text{ et } z=0]$$

Exemple 4 : La conjecture de Fermat

```
grandPere : Set(Pere x Pere) --> Set(GrandPere)
            P --> {GrandPere(x,z) |
                    ∃z∈Homme (Pere(x,y),Pere(y,z))∈P}
```

Exemple 5 : La fonction Descartes grandPere

Dans ce cas, les données méritent bien le qualificatif de texte informatique. Or, nous ambitionnons pour Io de savoir lire ou écrire n'importe lequel de ces textes. La lecture ou l'écriture des "programmes" sera particulièrement utile pour Descartes lui-même : il faut que Descartes lise les spécifications écrites dans ses langages, et qu'il écrive les programmes générés selon le langage-cible visé. A long terme, plutôt que de réécrire des analyseurs spécifiques pour chaque langage à traiter, nous utiliserons plutôt Io et Descartes lui-même pour synthétiser des composants de Descartes. Il s'agit d'un cas limité de bootstrapping.

2.2 Un langage de spécification couvrant les exemples

Qu'y a-t-il de commun entre ces exemples ? Certains paraissent très simples (écrire le programme manuellement prendrait moins d'une heure). D'autres réclameraient beaucoup plus de temps : il faudrait faire une partie non négligeable d'un *compilateur*.

L'expérience en informatique, et en particulier la théorie des compilateurs, fournit heureusement les moyens d'atteindre nos ambitions.

Premièrement, il faut construire (ou plutôt générer, et donc spécifier) un *analyseur syntaxique*. *Stricto sensu*, un analyseur syntaxique est un programme qui *vérifie* qu'un texte informatique *respecte* une syntaxe définie par ailleurs. Il ne "lit" donc pas vraiment le texte, au sens où il ne "retient" pas ce qu'il lit. Cependant, un tel programme peut servir de base, en lui rajoutant des instructions pour mémoriser de façon ad hoc le contenu du texte informatique. C'est sur cette mémorisation que d'autres programmes travailleront, pour compiler, traduire, etc.

La théorie des langages pour lesquels il est facile et systématique de construire un analyseur syntaxique est assez bien développée (même si elle est en constante évolution). Cette théorie utilise le plus souvent la notion de *grammaire context-free*, définie par une *bnf*, qui spécifie la syntaxe du langage auquel on s'intéresse. A une bnf est le plus souvent associée une *grammaire lexicale*, qui définit les *lexèmes*, ou "mots" présents dans le texte. L'avantage des grammaires context-free est qu'elles sont assez générales pour modéliser la plupart des textes informatiques utilisés, et en même temps suffisamment spécifiques pour permettre leur traitement de façon automatisée. Ainsi, il existe maintenant de nombreux générateurs d'analyseurs syntaxiques, dont le plus connu est *yacc* (*yet another compiler's compiler*). Toutefois, ces générateurs n'acceptent en général qu'une sous-classe des grammaires context-free (grammaires SLR(1) pour yacc), ce qui n'est pas trop gênant dans la pratique.

La question de la description de la syntaxe a donc été en quelque sorte résolue pour nous par les nombreux travaux en informatique[3].

Pour autant, le contenu du texte informatique n'est pas décrit par sa syntaxe. Un contenu donné peut d'ailleurs se "couler" dans de nombreuses syntaxes. Par exemple, n'importe lequel des textes des exemples 3, 4 et 5 (un programme Pascal, un énoncé

[3] Tout cela était loin d'être évident. Rappelons par exemple que réaliser un compilateur Algol était en 1960 du niveau d'une thèse d'état.

mathématique et une fonction Cogito) peut être représenté dans la syntaxe utilisée dans l'exemple 2 (un ensemble de triplets objet-attribut-valeur). Tous ceux qui ont travaillé sur la synthèse de programme en utilisant Genesia II ou Shal le savent bien.

La description du contenu a aussi été résolue par le travail de Bruno Ginoux [Ginoux 91] et de Jean-Philippe Lagrange [Lagrange 92]. Ceux-ci, s'inspirant des nombreux modèles de données existant, ont défini un modèle conceptuel entité-associations, largement étendu dans Ogre, et qui sert de base à Descartes et Ogre. Nous reprenons donc intégralement les acquis de ces travaux.

2.3 Aspects spécifiques à Io

Finalement, que nous reste-t-il à faire ? Deux choses.

Premièrement, il va falloir (encore) étendre le modèle conceptuel. Deux notions vont être introduites : une définition de *l'égalité entre entités*, et la notion de *portée* qui est essentielle à cette définition. Une justification rapide : dans Descartes et dans Ogre, on suppose que l'on "dispose" déjà de données bien définies (dans le premier cas, elles "sont" dans le modèle conceptuel, dans le second cas dans la base de données relationnelle). On a aussi implicitement en donnée l'égalité entre entités. Or, dans notre cas, il peut fort bien s'avérer que deux objets syntaxiquement identiques correspondent en fait à des objets conceptuellement distincts (homonymes). L'inverse est aussi possible (synonymes). Lorsqu'on lit le texte informatique, il faut donc pouvoir distinguer les entités qui y sont représentées. Nous avons préféré fournir au niveau conceptuel les notions qui permettent cela, par une nouvelle définition de l'égalité des entités, c'est-à-dire indépendamment de la syntaxe. La notion de portée est essentielle à cette définition.

Deuxièmement, il va falloir définir des liens entre la spécification de la syntaxe et le modèle conceptuel. Dans le modèle conceptuel, on mentionne des *entités* et des *associations*. Dans une bnf, des *termes* (se répartissant en *non-terminaux* et *terminaux*) et des *règles*. On va définir des liens entre entités et termes, et des liens entre associations (orientées) et chemins entre termes via les règles de grammaire.

Une grande partie de ces liens pourra être retrouvée, en partie heuristiquement. Nous verrons en effet qu'il n'y a à notre connaissance que deux types de syntaxes informatiques généralement utilisées : des syntaxes que nous baptisons "emboîtantes", et des syntaxes "en vrac". Les syntaxes des exemples 1, 3, 4 et 5 sont des syntaxes emboîtantes : on comprend le rôle d'un identificateur à partir de sa place par rapport aux autres identificateurs dans le texte. De plus, un objet comme une expression englobe syntaxiquement un objet comme une variable. Les liens entre objets représentés sont donc implicitement représentés par leur places et emboîtements respectifs. L'exemple 2 correspond à une syntaxe en vrac : les liens entre objets sont *explicitement* décrits, via les attributs. L'aspect contextuel de la syntaxe (une suite de triplets, et c'est tout) n'a qu'un rôle mineur.

3 Le modèle conceptuel

3.1 Le modèle conceptuel de Descartes

Le modèle conceptuel de Descartes comprend des entités et des associations. Les entités sont regroupées en ensembles. Les associations modélisent des relations entre entités, et forment elles-mêmes des ensembles. Un premier groupe de contraintes peut être défini sur les associations, à savoir sur les cardinalités des liens qu'elles créent entre entités.

Divers opérateurs permettent de combiner entités et associations, comme les opérateurs Set (ensemble de), List (liste de) et le produit cartésien x. On peut aussi définir un ensemble d'entités comme réunion (éventuellement disjointe) d'autres ensembles d'entités. Jean-Philippe Lagrange utilise aussi dans Ogre des définitions d'ensembles d'entités utilisant tout le langage des prédicats du premier ordre (adapté au cadre du modèle conceptuel).

On peut aussi définir des contraintes. Dans Ogre, Jean-Philippe Lagrange considère des contraintes simples comme les contraintes d'inclusion (on sait qu'un ensemble d'entités est inclus dans un autre), mais aussi des contraintes plus complexes, auxquelles il réserve le nom “noble” bien que répandu de contraintes d'intégrité, et qui utilisent toute la puissance des prédicats du premier ordre.

Enfin, on utilise dans Descartes des fonctions issues des associations. Ainsi, si A est une association entre trois ensembles d'entités E, F et G de cardinalités quelconques, on peut définir par exemple la fonction

$$\begin{aligned} h : \quad & E \dashrightarrow Set(F \times G) \\ & x \dashrightarrow \{(y,z) \in F \times G \mid A(x,y,z)\} \end{aligned}$$

Dans le cas plus simple où l'association est binaire, par exemple une association B sur E x F, on peut considérer les deux fonctions inverses l'une de l'autre

$$\begin{aligned} f_{B,E\dashrightarrow F} : \quad & E \dashrightarrow Set(F) \\ & x \dashrightarrow \{y \in F \mid B(x,y)\} \end{aligned}$$

$$\begin{aligned} f_{B,F\dashrightarrow E} : \quad & F \dashrightarrow Set(E) \\ & y \dashrightarrow \{x \in E \mid B(x,y)\} \end{aligned}$$

Si les cardinalités sont correctes, ces fonctions arrivent dans F ou E au lieu de Set(F) ou Set(E). Elles sont appelées *fonctions issues du modèle conceptuel*.

3.2 Restrictions pour Io

Nous adoptons les restrictions suivantes pour Io :

- On autorise les ensembles d'entités "élémentaires" et les ensembles obtenus par utilisation des opérateurs Set, List et réunion disjointe. Les produits cartésiens

doivent être "cassés" en associations binaires. Les ensembles définis par des formules ensemblistes et/ou prédicatives quelconques ne sont pas admis.

- Certains ensembles d'entités seront considérés comme "terminaux", ce qui signifie qu'ils ne sont qu'un renommage d'un des ensembles d'entités suivants : entiers, réels ou chaînes de caractère[4]. Les autres sont appelés dans notre jargon un peu spécial "non terminaux"[5].

- Un des ensembles d'entités est appelé "ensemble de départ". On considère que ses entités résument toute l'information contenue dans le texte à traiter.

- Les associations sont binaires (quoique cette contrainte pourrait être levée si nécessaire). Le plus souvent, les associations ne seront vues que comme étant "orientées dans un sens", c'est-à-dire qu'on ne considérera qu'une des deux fonctions issues du modèle qui en sont issues. La fonction utilisateur préférée sera souvent nommée attribut conceptuel[6].

- Les contraintes diverses (inclusions, contraintes d'intégrité) ne sont pas prises en compte.

Il faut noter que ces restrictions ne limitent pas la puissance du langage Io par rapport à Descartes, car il est toujours possible d'écrire des fonctions Cogito passant d'un modèle conceptuel Io limité à un modèle Descartes général.

3.3 Exemple et motivation de ce qui suit

3.3.1 Clés

Nous donnons ci-dessous le modèle conceptuel commun aux exemples 1 et 2. La syntaxe que nous adoptons ici est un peu particulière. On trouve entre crochets ou entre guillemets les ensembles d'entités. Les ensembles d'entités "terminaux" sont entre guillemets.

```
<Homme> <--   @Nom          "Nom"
              <--  @Prenom          &Set("Prenom")
              <--  @Age             "Age"
              <--  @NombreDEnfants  "NombreDEnfants"
              <--  @NumeroSS        "NumeroSS"
```

Modèle conceptuel des exemples 1 et 2

---

[4] Les chaînes de caractères sont en Descartes l'ensemble d'entités CC = List(Caractere) où Caractere est l'ensemble fini connu à l'avance des caractères disponibles.

[5] On verra les raisons de ces appellations lorsque nous décrirons les bnf (ou il y a aussi des non-terminaux et des terminaux).

[6] Qu'il ne faudra pas confondre avec les attributs d'un modèle conceptuel Cogito ou Ogre, qui sont des associations un peu particulières, liant un "vrai" ensemble d'entités avec un ensemble "prédéfini", comme les entiers, les réels ou les chaînes de caractères.

Les attributs conceptuels (fonctions dérivées d'associations) sont précédées d'un caractère @. Set s'écrit ici &Set et List &List. Lorsque le type d'un attribut conceptuel est un ensemble d'entités sans utilisation de &Set ou de &List, c'est que cet attribut est "monovalué". Sinon, on utilise &Set ou &List. Un attribut monovalué peut ne pas avoir de valeur du tout pour une entité donnée. De même, lorsqu'il est multivalué, l'ensemble de ses valeurs peut être vide.

Enfin, le fait que des noms d'ensembles d'entités soient identiques à des noms d'attributs conceptuels n'est pas ici gênant.

Lorsque l'on va lire le texte informatique pour "charger" le modèle conceptuel Cogito (du moins une représentation dans une structure de données du programme qui lit), il va falloir reconnaître les entités et les instances d'associations. Paradoxalement, le plus difficile sera de reconnaître les entités.

Prenons l'exemple 1. Dès que l'on commence, on lit un numéro de sécurité sociale. Nous savons (mais le système ne le sait pas encore) qu'un tel numéro correspond à un unique individu. Dans le vocabulaire des bases de données, le numéro de SS est une *clé*. Lorsque l'on continue la lecture et que l'on rencontre un nouveau numéro de SS, on a affaire à un nouvel individu si celui-ci est différent de tous les numéros déjà rencontrés. Sinon, on a affaire au même individu que celui lu précédemment et ayant le même numéro de SS.

De manière générale, une clé d'un ensemble d'entités sera un sous-ensemble de ses attributs conceptuels. Dans le cas de Homme, le singleton {NumeroSS} est une clé. Mais il peut y avoir d'autres clés. Par exemple, si l'on se trouve dans un monde restreint où il n'y a pas deux personnes distinctes ayant les mêmes nom et prénoms, alors {Nom,Prenom} constitue aussi une clé.

Une clé permet donc de savoir exactement quand deux entités sont *égales*. L'égalité des entités n'est pas *donnée*, mais *définie par leurs propriétés*.

Lorsqu'on "lira" des entités, on pourra donc savoir dès qu'une de ses clés est "remplie" de quelle entité il s'agit, i.e. s'il s'agit de la même entité qu'une entité du même type déjà rencontrée ou pas.

Ogre utilise cette notion de clé.

3.3.2 Portée

Malheureusement, la notion de clé ne suffit pas.

Considérons l'exemple 3.

```
function fact(n : integer) : integer;

begin
    if n=0 then fact:=1
    else fact:=n*fact(n-1)
```

```
end;
```

Exemple 3 : Fonction factorielle *en Pascal*

Le paramètre `n` de la fonction Pascal `fact` est "défini" par le fait qu'il est une variable de cette fonction. Lorsque dans le corps du programme, nous rencontrons l'instruction

```
if n=0 then …
```

il faut savoir s'il s'agit bien de la même variable `n`, ce qui est le cas ici.

Or, il s'avère qu'en Pascal on peut utiliser une variable définie dans une procédure *emboîtante*. Plus précisément, si on a k procédures ou fonctions emboîtées :

```
procedure p1(…) : …;
        …
        procedure p2(…) : …;
                …
                procedure pk(…) : …;
                begin
                        if v=0 then …
                end;
        begin …
        end;
begin
end;
```

la variable v mentionnée dans l'instruction if v=0 then … *est* la variable v déclarée dans la procédure emboîtante *la plus emboîtée* contenant l'instruction.

Considérons un modèle conceptuel simplifié et partiel d'un programme Pascal.

```
<SousProg>      =     <Programme> ∪ <Procedure> ∪ <Fonction>
<SousProg>      <--   @SPEmboite &Set(<SousProg>)
                <--   @Variables &Set(<Variable>)
                <--   @Corps <Instruction>
<Instruction>   =     <Affectation> ∪ <IfThenElse> ∪ <Bloc> ∪ …
<Affectation>   <--   @PGauche <Expression>
                <--   @PDroite <Expression>
<IfThenElse>    <--   @If <Expression>
                <--   @Then <Instruction>
                <--   @Else <Instruction>
<Bloc>          <--   @ContientBloc &Set(<Instruction>)
        …
<Expression>    =     <Variable> ∪ <Constante> ∪ <ExpComplexe>
<Variable> <--  @NomVariable "NomVariable"
                      @TypeVariable <Type>
```

```
<Constante>      <--   @NomConstante "NomConstante"
<ExpComplexe>    =     <ExpCompare> ∪ …
<ExpCompare>     <--
                       @OpCompare {"=","<>","<","<=",">",">="}
                             @CompareG <Expression>
                             @CompareD <Expression>
        …
```

Modèle conceptuel simplifié et partiel d'un programme Pascal

Il n'est pas possible de définir une clé pour l'ensemble d'entités <Variable>. En effet, deux variables portant même nom et ayant même type peuvent correspondre à deux variables distinctes, et une variable n'a pas d'autre attribut que NomVariable et TypeVariable.

Par contre, il y a plusieurs "chemins" allant de l'ensemble d'entités <SousProg> vers <Variable>. Deux de ces chemins sont

```
[Variables]
[Corps,<IfThenElse>,If,<ExpCompare>,CompareG,<Variable>]
```

Ces chemins mentionnent des attributs conceptuels et des parties d'une réunion. Il y en a d'autres, qui relient aussi une procédure à une variable *via* une de ses instructions. Il y en a même une infinité, car certains de ces chemins comportent des circuits (un bloc d'instructions contient des instructions). Appelons C leur ensemble.

Enfin, il y a une boucle de <SousProg> vers lui-même :

```
[SPEmboite]
```

Définissons la *portée* de l'ensemble d'entités <Variable> comme étant le quadruplet

```
(<SousProg>,[Variables],C,[SPEmboite])
```

La portée permet de retrouver la procédure où une variable est déclarée. Plus précisément, considérons la traduction dans le modèle conceptuel de l'instruction if v=0 then …. Nous remontons le chemin de C [Corps,<IfThenElse>,If,<ExpCompare>,CompareG,<Variable>]. Nous trouvons une procédure. Si cette procédure a pour variable déclarée une variable de même nom que v, i.e. une des valeurs de l'attribut composé Variables.NomVariable égale à "v", alors cette procédure est celle où v est déclarée. Sinon, il faut parcourir "à l'envers" la boucle [SPEmboite] pour trouver une nouvelle procédure (emboîtante), et recommencer le test. On remonte ainsi dans les procédures emboîtantes jusqu'à retrouver la procédure où v est déclarée.

Ainsi, une variable (i.e. un élément de l'ensemble d'entités <Variable>) est définie

- par sa "clé" {NomVariable}

- par sa portée

La règle (utilisant la portée) pour savoir si deux variables sont égales est définie formellement au § 3.4.

3.3.3 Généralité de la portée

La notion de portée accompagnée de la règle définissant l'égalité des entités s'avère générale et rencontrée fréquemment. Ainsi, si l'on prend une expression mathématique du genre de celle de l'exemple 4, par exemple

$$\forall x \quad [\exists y\, P(x,y)] \quad \Rightarrow \quad [\forall y \quad Q(x,y) \Rightarrow R(x,y)]$$

cette expression mentionne une variable x mais *deux* variables y. Une variable est ici définie par son nom et sa présence après un quantificateur dans l'expression la plus emboîtée contenant son occurrence. Pour simplifier, la portée dans une expression prédicative est définie par les quantificateurs[7].

Si l'on veut s'envoler quelque peu, on peut dire que la notion de portée est ce qui permet de résoudre, dans les textes informatiques, le problème de l'anaphore.

3.4 Dimension, clés et portée : définitions

3.4.1 Clé seule

Nous donnons maintenant une définition formelle des notions introduites au § 3.3.

A tout ensemble d'entités sont associées des clés, éventuellement liées à des portées. Tous les cas sont possibles; on peut ainsi avoir une clé sans portée et deux clés avec chacune sa portée pour le même ensemble d'entités. Regardons d'abord le cas d'une clé à laquelle aucune portée n'est associée.

Une clé d'un ensemble d'entités E est un ensemble $\{a_1,\ldots,a_n\}$ d'attributs conceptuels de E. Sa propriété caractéristique, lorsqu'aucune portée n'est mentionnée, est

$$\forall (x,y) \in E^2 \quad [\forall i \in [1,n] \quad a_i(x) = a_i(y)] \quad \Rightarrow \quad x = y$$

En Français : si deux entités ont les mêmes valeurs pour les attributs d'une clé, alors il s'agit d'une seule et même entité.

Attention ! Si $a_i(x)$ et $a_i(y)$ n'ont pas de valeur (attributs "monovalués"), on considère que l'on ne peut pas conclure. La condition précédente impose donc *l'existence* de valeurs pour les attributs de la clé.

Enfin, des clés différentes doivent fournir la même définition de l'égalité.

---

[7] On dit en Anglais le "scope" de la variable quantifiée.

Mais cette définition n'est plus valable lorsqu'une portée est associée.

3.4.2 Clé et portée

Lorsque l'on a un modèle conceptuel, on peut définir le graphe orienté dont les sommets sont les ensembles d'entités et où on a un arc de E vers F ssi F est type des valeurs d'un attribut de E, ou fait partie d'une réunion définissant E. Notons GC ce graphe, et appelons-le *graphe conceptuel*.

Les chemins dans GC définissent des attributs composés. Ainsi, dans l'exemple 3, et étant donné le modèle conceptuel d'un programme Pascal du 3.3.2, si l'on suit le chemin

[Corps,<IfThenElse>,If,<ExpCompare>,CompareG,<Variable>]

à partir de l'entité `fact` de <SousProg>, on "tombe" sur la variable `n`.

La portée d'un ensemble d'entités E est la donnée d'un quadruplet

(S,C,$\mathcal{C}$,B)

où :

- S est un ensemble d'entités,
- C est un chemin sans circuit de S vers E,
- $\mathcal{C}$ est un ensemble de chemins, ceux avec circuits compris, de S vers E,
- B est un circuit élémentaire de S vers S.

S est appelé le *pivot* de la portée, C le *chemin de déclaration*, les chemins de $\mathcal{C}$ les *chemins d'occurrence*, et B sa *boucle d'emboîtement*.

En général, on prend pour $\mathcal{C}$ tous les chemins de S vers E qui ne repassent pas par S (la boucle B n'est donc incluse dans aucun de ces chemins). Vus les circuits possibles, il y a a priori une infinité de chemins dans $\mathcal{C}$.

Donnons-nous maintenant une clé A = $\{a_1,\ldots,a_n\}$ de E. Pour toute entité e de E définissons $p_A(e)$ = s l'élément s de S tel que :

- Si l'on suit à partir de S la boucle B k fois, puis un chemin de $\mathcal{C}$, on trouve e
- Si l'on suit le chemin C à partir de s, on trouve une entité e' de E ayant les mêmes valeurs que e pour tous les attributs de la clé A
- k est minimum

On suppose que s existe et est bien défini (est unique).

Définition : Deux entités e_1 et e_2 de E sont *égales* ssi $p_A(e_1) = p_A(e_2)$.

Cela implique que les valeurs des attributs de leurs clés sont égales. Comme précédemment, les clés doivent être compatibles, c'est-à-dire fournir une même définition de l'égalité.

3.4.3 Portée affaiblie

La définition de la portée peut être affaiblie pour englober certains cas plus simples. Ainsi, dans une règle Genesia II ou Shal, comme

```
regle GrandPere
si    Pere(X) = (Y)
      Pere(Y) = (Z)
alors
      GrandPere(X) = (Z)
```

les variables X, Y et Z sont définies pour cette règle GrandPere. Cela signifie que deux occurrences de (X) à deux endroits de cette règle constituent bien des références à une seule et même variable, mais que deux occurrences dans deux règles distinctes se réfèrent à deux variables distinctes.

Dans ce cas, on va définir la portée de l'ensemble d'entités <Variable> par un quadruplet (<Regle>,[],C,[]), où le chemin de déclaration de <Regle> vers <Variable>, et la boucle d'emboîtement de <Regle> vers <Regle> sont vides. C est l'ensemble des chemins joignant <Regle> à <Variable> dans le modèle conceptuel.

De manière générale, l'égalité entre deux entités e_1 et e_2 d'un ensemble d'entités E sera définie par une clé A associée à une portée affaiblie (S,[],C,[]) de la manière suivante:

- e_1 et e_2 ont mêmes valeurs pour les attributs de la clé A,
- si l'on appelle $p(e_1)$ et $p(e_2)$ les éléments de S auxquels on aboutit en remontant les chemins de C, on a $p(e_1) = p(e_2)$.

En fait, une portée affaiblie revient à rajouter à la clé C les composés des inverses des attributs le long des chemins de C.

Nous ne mentionnerons plus dans la suite les portées affaiblies. Ce qui sera dit pourra facilement être étendu à ce cas.

3.4.4 Définition par point fixe de l'égalité

Les définitions précédentes ne sont pas tout à fait des définitions. Par exemple, si l'on regarde de près la seconde partie de la définition de la portée :

- Si l'on suit le chemin C à partir de s, on trouve une entité e' de E ayant les mêmes valeurs que e pour tous les attributs de la clé A

on voit que l'on compare des valeurs des attributs de e et e'. Or, ces valeurs *sont elles-mêmes des entités*, et leur égalité étant aussi définie par clés et portées, elles peuvent très bien mentionner l'égalité entre éléments de E que nous sommes justement en train de définir. De même, le cas plus simple où l'égalité des entités est définie par une clé sans portée mentionne l'égalité entre les valeurs pour les attributs de la clé.

Nous avons donc une circularité de la définition de l'égalité. Dans ces cas-là, pour s'en sortir, on déclare que la définition est une équation de point fixe. Maintenant, une telle équation peut avoir plusieurs solutions[8], et il faut choisir entre elles.

L'égalité que nous voulons définir est une relation sur la somme directe de tous les ensembles d'entités du modèle conceptuel, et de telles relations sont ordonnées par l'inclusion.

Nous définissons l'égalité entre entités comme la solution *maximale* des équations de point fixe ci-dessus. Il faudrait montrer que cette solution maximale existe, nous ne le faisons pas.

3.3.5 Mise en œuvre de la vérification de l'égalité

En pratique, la définition de l'égalité comme solution maximale des équations de point fixe signifie :

> *"Deux entités sont égales à moins qu'on puisse prouver qu'elles sont différentes"*

qui est une sorte de *règle de défaut*. Si l'on y regarde de près, on va pouvoir presque se ramener à l'égalité sur les clés, ce qui va avoir des conséquences importantes pour la mise en œuvre de la vérification de l'égalité.

Supposons que nous voulions définir une fonction $Eg_E(e_1,e_2)$ pour chaque ensemble d'entités E qui vérifie l'égalité de e_1 et e_2 en renvoyant "vrai" ssi les deux entités sont égales. Si E est un ensemble d'entités terminales (renommage des chaînes de caractères, entiers ou réels), la définition de cette fonction est évidente. Sinon, ces procédures vont s'appeler mutuellement, et donc être éventuellement récursives. Ce qu'il faut, c'est être sûr que cette récursion se termine dans tous les cas.

Pour que les deux entités e_1 et e_2 de E soient différentes, soit e_1 et e_2 ont une valeur différente pour un des attributs de la clé, soit $p_A(e_1)$? $p_A(e_2)$. Il faut donc regarder la

---

[8] Il faudrait aussi démontrer l'existence de ne serait-ce qu'une solution. Nous ne le faisons pas, car il faudrait pour cela imposer des contraintes sur les clés et les portées, ce que nous avons laissé dans l'ombre.

définition de l'égalité pour les valeurs des attributs de A et pour $p_A(e_1)$ et $p_A(e_2)$, et pour cela appeler les Eg_V et/ou Eg_S.

Si les appels des Eg_V et de Eg_S n'impliquent pas celui de Eg_E, il n'y a pas de problème. Si par contre Eg_E est appelée, deux cas sont possibles : elle est appelée avec des arguments différents de (e_1,e_2), ou avec ces arguments. Dans le premier cas l'appel va se dérouler normalement. Mais dans le second, l'appel doit renvoyer immédiatement la valeur "vrai". En effet, *il sera impossible de prouver que e_1 et e_2 sont différents, à moins que le premier appel d'un des Eg_V ou de Eg_S ne renvoie "faux"*.

Il faut donc que les fonctions Eg maintiennent l'ensemble des couples d'entités pour lesquels une fonction d'égalité a été appelée[9]. Si une fonction d'égalité est appelée avec un couple élément de cet ensemble, elle doit immédiatement retourner "vrai".

En définitive, tous les appels vont bien s'arrêter, car, à la limite, l'égalité de tous les couples d'entités de même type aura été testée, et il n'y a qu'un nombre *fini* d'entités.

Quelque chose ressemblant à cet algorithme devra être généré par le composant IoIn. En fait, la situation y est un peu différente, car on voudrait que les égalités soient testées *alors que le "texte" n'est pas encore complètement lu*. Dans ce cas, certains attributs peuvent ne pas encore avoir été remplis, qui permettraient à une des fonctions Eg de décider de la réponse, et il leur faudra donc attendre la valeur correspondante. Il serait bon de mettre en place un système de *démons* qui fasse redémarrer le calcul dès que la valeur est trouvée (ou, si la valeur est un ensemble ou une liste, dès que l'on est sûr que l'ensemble ou la liste est complètement rempli). Tout cela sera spécifié dans IoIn.

3.4.6 Dimension

En plus des définitions précédentes, et pour faciliter la tâche des générateurs de programmes IoIn et IoOut, on introduit une nouvelle notion, la dimension[10].

Nous avons vu que les attributs "monovalués" n'ont pas nécessairement de valeur. Les attributs "multivalués" en ont forcément une, puisque l'ensemble vide ∅ constitue une telle valeur.

Nous définissons la *dimension* d'un ensemble d'entités E comme une clé $A = \{a_1,\ldots,a_n\}$ dont *tous les attributs ont au moins une valeur*.

Cette définition ne concerne donc que les attributs "monovalués" de la clé.

On impose aussi de ne définir *qu'une dimension* par ensemble d'entités.

---

9 Plus précisément, cet ensemble doit constituer un paramètre supplémentaire des Eg.

10 Nous n'avons pas voulu jouer sur la musique de nos vocables jusqu'à baptiser la dimension la *tonalité*.

Dans l'esprit, la dimension est ce qui fait "qu'une entité est bien cette entité là". Par exemple, une règle Genesia II ou Shal porte un nom, mais les utilisateurs de Genesia II et de Shal savent bien que ce nom importe peu. En fait une règle est caractérisée :

- par son ensemble de prémisses
- par son ensemble de conséquents

Ces deux attributs formeront la dimension d'une règle dans un modèle conceptuel[11].

4 Syntaxe

La syntaxe selon laquelle lire ou écrire les données est représentée essentiellement par une grammaire sous forme de bnf. Il faut en fait ajouter à cela une grammaire lexicale, qui modélise indépendamment de la syntaxe globale les lexèmes, c'est-à-dire les "mots" qui constituent le texte au-delà de la simple suite de caractères. Ces mots sont les identificateurs, les nombres, les mots-clés, les signes de ponctuation, etc.

Nous commençons par la description d'une bnf, avant d'aborder les aspects lexicaux. Il est donc demandé au lecteur de supposer dans la première partie que l'on dispose d'un moyen pour regrouper les caractères du texte en mots.

4.1 Bnf

4.1.1 Un exemple

Pour appuyer les explications, la bnf de l'exemple 1 (qui est rappelé) est donnée ci-dessous.

```
1250132001003
    Toto    Fulgence    Ulysse  68  1
1491045113012
    Lulu    Achille     Carolus 42  2
1720375014276
    Zonzon  Hector              20  0

<suiteindividu>   ::=   <individu> <suiteindividu>,
                            "";
<individu>        ::=   <numeross> <nom>
                                <suiteprenom> <age> <nbredenfants>;
<suiteprenom>     ::=   <prenom> <suiteprenom>,
                            "";
<numeross>              ::=   "code";
<nom>                   ::=   "identificateur";
<prenom>          ::=   "identificateur";
```

[11] En fait, si l'on voulait être tout à fait précis, il faudrait aussi tenir compte de la position de la règle dans la base de règles.

```
<age>                ::=   "nombre";
<nbredenfants>   ::=   "nombre";
```

L'exemple 1 et sa bnf

4.1.2 Définition

Une bnf est une suite de *règles de grammaire*. Les termes employés sont de deux sortes : *non-terminaux* (qui sont en quelque sorte des "variables") et *terminaux* (en quelque sorte des "constantes").

Les terminaux, aussi appelés *tokens*, représentent les *mots-clés* du langage et les *catégories* des autres mots. Ces mots-clés sont des chaînes de caractères fixes qui délimitent les expressions. Les éléments de ponctuation font partie des mots-clés. On utilise aussi comme token la chaîne de caractères vide "".

Les non-terminaux représentent des morceaux du texte ayant une certaine structure syntaxique. Cette structure est justement fournie par les règles.

A chaque non-terminal est associée une règle unique. A gauche de la règle, on trouve ce non-terminal, soit ntg. A droite, on trouve une ou plusieurs séquences de termes (non-terminaux et tokens). Chaque séquence représente une alternative pour que l'expression représentée par ntg soit syntaxiquement correcte. L'expression lue doit ainsi correspondre à une des séquences de la partie droite. Cela signifie que les tokens doivent être mentionnés en des places correctes, et les non-terminaux matcher les morceaux d'expressions résiduels selon les règles de grammaire qui les définissent.

Quelques points de terminologie. Lorsqu'on décompose un non-terminal ntg grâce à une des alternatives de sa partie droite, on dit que l'on a *effacé* le non-terminal par la règle, ou qu'il *s'efface* par la règle. On parle aussi quelquefois de *plusieurs* règles pour un même non-terminal, une par alternative de la partie droite.

Rien n'empêche (au contraire !) qu'un non-terminal soit utilisé de façon récursive, c'est-à-dire qu'il y ait une suite d'applications de règles partant de la règle associée à un non-terminal nt qui conduise à une expression contenant nt. En particulier, un même non-terminal peut apparaître à gauche et à droite de la même règle, comme <suiteindividu> et <suiteprenom> dans notre exemple.

Enfin, un des non-terminaux doit être distingué comme non-terminal de départ. Il modélise en quelque sorte tout le texte, c'est-à-dire qu'il doit correspondre à toute l'expression allant du début à la fin du fichier.

4.1.3 Analyse de l'exemple

Dans notre exemple, nous avons 8 non-terminaux, et donc 8 règles. On a également 3 tokens ("code", "identificateur" et "nombre"), plus le quatrième token "virtuel", la chaîne de caractères vide "". Les non-terminaux sont entre crochets <…>, les tokens entre guillemets "…". Les alternatives à droite d'une règle sont séparées par une virgule, et la règle se termine par un point-virgule.

Seules les première et troisième règles comportent une alternative à leur droite.

Regardons maintenant pas à pas pourquoi et comment le texte correspond bien à sa grammaire. Le non-terminal <suiteindividu> est le non-terminal de départ. Celui-ci s'efface (une première fois) par la première règle (première alternative) :

<suiteindividu> ::= <individu> <suiteindividu>

En effet, le texte peut être décomposé en deux parties, la première

```
1250132001003
    Toto    Fulgence    Ulysse  68  1
```

qui va correspondre au non-terminal <individu> à droite de la règle, le reste du texte correspondant au <suiteindividu> qui est à droite de la règle. On voit ici un usage de la récursivité sur les non-terminaux : elle permet de modéliser une suite d'expressions.

Pour voir que le non-terminal <individu> correspond bien au morceau de texte précédent, il faut utiliser la règle qui le définit :

<individu> ::= <numeross> <nom>
<suiteprenom> <age> <nbredenfants>;

<numeross> va correspondre à `1250132001003`, <nom> à `Toto`, etc. Comment ? En utilisant les règles pour ces non-terminaux, qui ont à droite un unique token, sauf <suiteprenom>. Or, l'analyseur lexical (que nous n'avons pas encore défini) a déjà découpé le texte en mots et renvoyé les tokens correspondant à ces mots. Ainsi, il a "dit" que la chaîne `1250132001003` était un token "code", `Toto` un token "identificateur", etc.

<suiteprenom> est cependant à part car il correspond à `Fulgence Ulysse`. On l'analyse par la règle

<suiteprenom> ::= <prenom> <suiteprenom>

<prenom> va correspondre à `Fulgence` et une nouvelle occurrence de <suiteprenom> à `Ulysse`. Cette nouvelle occurrence s'efface par la même règle, d'où une nouvelle occurrence de <prenom> correspondant à `Ulysse` et une troisième de <suiteprenom> à … la chaîne vide. Cette dernière occurence va s'effacer en utilisant la règle

<suiteprenom> ::= ""

et l'analyse du premier individu est terminée.

Le reste du texte va s'analyser de la même manière, énumérant une suite d'individus.

4.1.4 Arbre syntaxique

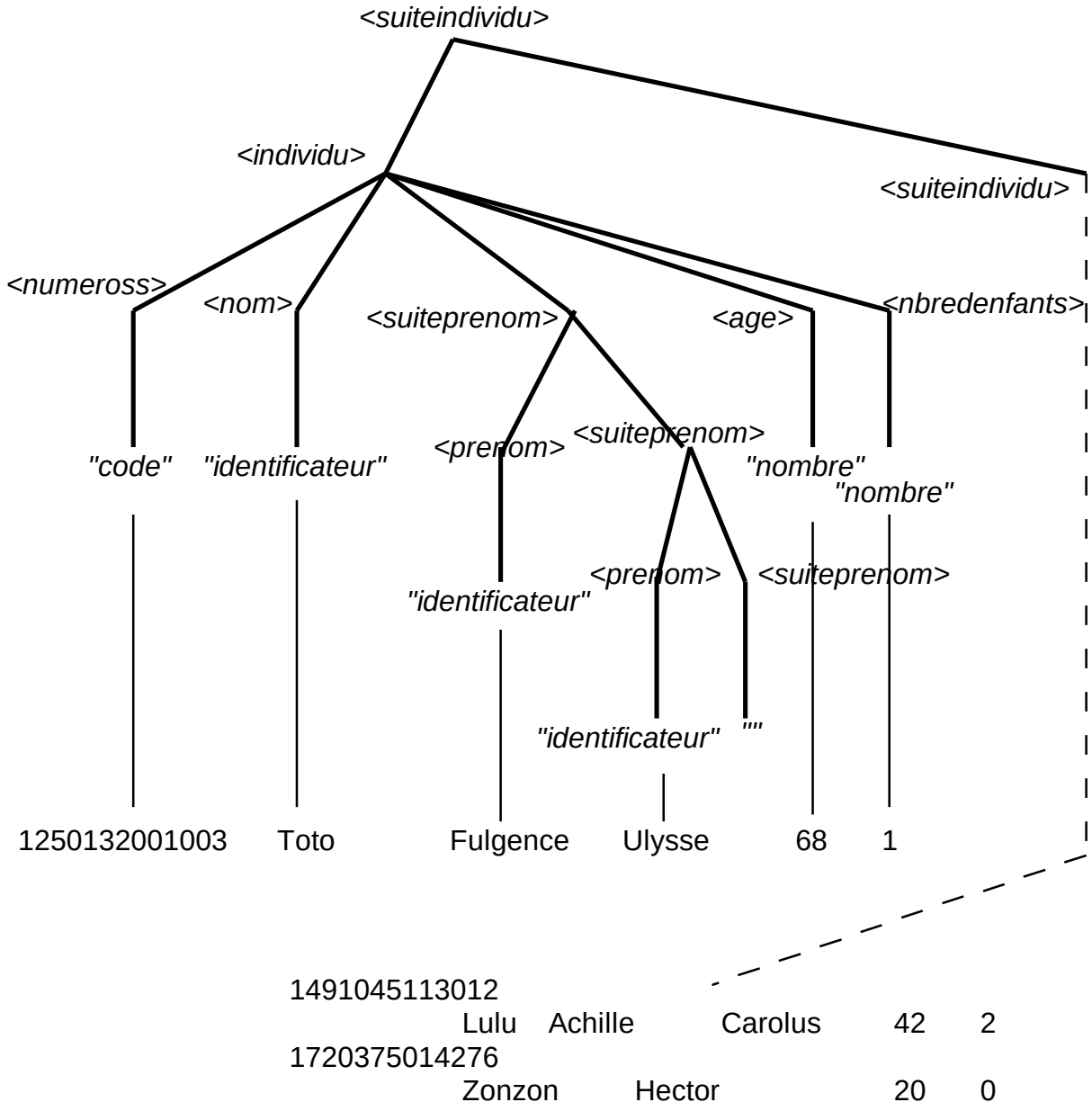

Figure 1 : L'arbre syntaxique de l'exemple 1

Lorsqu'un texte suit une grammaire bnf donnée, les règles le structurent en un arbre appelé *arbre syntaxique*. Chaque nœud de l'arbre est libellé par un non-terminal ou un token, et correspond à une partie du texte. La partie correspondant à un sous-arbre est incluse dans celle correspondant à n'importe quel sur-arbre.

La figure 1 représente une partie de l'arbre syntaxique de l'exemple 1.

4.1.5 Grammaires bnf et analyse syntaxique

Dans le § 4.1.3, nous avons montré pourquoi notre exemple obéissait bien à sa grammaire : il est possible d'appliquer d'une certaine manière les règles (car il y a des alternatives) de manière à décomposer le texte jusqu'à ses "mots" élémentaires.

Cependant, un programme effectuant une analyse syntaxique, c'est-à-dire effectuant cette vérification, ne procède en général pas de cette manière[12]. Diverses sous-classes des grammaires bnf ont été définies dans la littérature, qui permettent de mettre en œuvre, si la grammaire à traiter appartient à une de ces sous-classes, des algorithmes beaucoup plus rapides. Citons les grammaires LL(k), LR(k) et SLR(k). Nous avions successivement mis en œuvre dans Shal des générateurs d'analyseurs pour les grammaires LL(2) et SLR(0). Le compromis habituellement admis est SLR(1), qui est la classe traitée par *yacc*.

Le choix du type d'analyseur syntaxique est repoussé à la spécification du composant IoIn. Il se pourrait, si nous en avons le temps, que nous tentions d'innover radicalement par rapport à la littérature.

Par ailleurs, l'analyse syntaxique stricto sensu ne suffit pas : il faut aussi "traiter ce qu'on a lu". Dans notre cas, il s'agit de mettre en relation les éléments syntaxiques reconnus par l'analyseur et les éléments du modèle conceptuel. La base pour cela est fournie dans le chapitre suivant.

Enfin, d'autres problèmes peuvent se poser pour certaines grammaires. Elles peuvent par exemple être *ambiguës*. Nous avons vu qu'un texte est syntaxiquement correct s'il y a une manière d'utiliser les règles pour les faire correspondre à ses morceaux. Mais il peut exister plusieurs manières, car les règles présentent généralement plusieurs branches d'alternative. Ainsi, à un même texte peuvent correspondre plusieurs arbres syntaxiques. Lorsque l'on fait ensuite le lien avec le modèle conceptuel, on pourrait trouver plusieurs contenus différents à un même texte.

Nous rejetons tous ces cas d'ambiguïté. Si la grammaire elle-même est non-ambiguë (c'est-à-dire, si pour tout texte acceptable, il n'y a qu'une suite unique d'applications des règles qui l'acceptent), toute ambiguïté aux autres niveaux est banie. Or, il existe des algorithmes qui testent si une grammaire est ambiguë [Aho, Sethi & Ullman 88]. Les composants Io pourront utiliser un de ces algorithmes.

4.2 Grammaires lexicales

Venons-en maintenant à la reconnaissance des mots. Les mots sont définis par ce qu'on appelle une grammaire lexicale. Une telle grammaire est un ensemble *d'expressions régulières* qui modélisent des "patrons" de chaînes de caractères. Les expressions régulières peuvent aussi être vues comme l'ensemble des chaînes de caractère obéissant au patron. Les chaînes à 1 (caractère seul) ou 0 (chaîne vide) caractère sont admises. La grammaire lexicale définit les tokens utilisés dans la grammaire bnf associée.

---

[12] Quoiqu'une bnf puisse être écrite en Prolog, et constituer ainsi *immédiatement* un analyseur, qui va "fonctionner" comme notre explication du § 4.1.3.

4.2.1 Exemple

Pour fixer les idées, la grammaire lexicale de l'exemple 1 est :

```
{delim}          [ \t\n];
{ws}             {delim}+    ;
{maj}            [A-Z]
{min}            [a-z]  ;
{digit}          [0-9]
{id}             (({maj} {min}*) [-])* ({maj} {min}*)    ;
{slash}          [/]     ;
{star}           [\*]    ;
{charnoneof}     {char} \ {eof}    ;
{comm}           {slash} {star} {charnoneof}* {star} {slash}    ;
{null}           {ws} | {comm}    ;

"code"                  [12]{digit}^12
"identificateur"  {id}
"nombre"          {digit} | {digit}^2 | {digit}^3
```

La grammaire lexicale de l'exemple 1

4.2.2 Définition

Chaque token est défini par des règles qui mentionnent en partie droite une expression régulière.

Une expression régulière élémentaire est un caractère. Une expression complexe peut être formée en utilisant une des opérations :

- ensemble de caractères, soit par énumération, soit par intervalle (en considérant un ordre alphabétique donné sur les caractères)
- concaténation d'expressions régulières
- alternative entre expressions
- différence d'expressions
- utilisation d'un des opérateurs ^n, * ou +.

Quelques notations :

- Certains caractères non directement accessibles au clavier ou ayant un sens particulier comme mot-clé ou ponctuation dans notre syntaxe des grammaires lexicales sont représentés en utilisant en général le caractère \. Ainsi, \t est le caractère de tabulation, \n celui de fin de ligne et * tout simplement *. \\ est \. Exception : eof est le caractère de fin de fichier. {char} est l'ensemble de tous les caractères.

- Un identificateur entre accolades (comme {digit}) est une notation pour une expression régulière. Ces notations ne sont pas indispensables, mais allègent l'écriture. Une notation peut être utilisée comme une expression régulière.

- Les tokens définis sont entourés par des guillemets (comme "code").

Nous définissons le sens de chaque expression par l'exemple.

- [\t\n] est l'ensemble des trois caractères blanc, tabulation et fin de ligne.

- [a-z] est l'ensemble des caractères compris entre a et z (toutes les lettres minuscules).

- {min}* est l'ensemble des chaînes de caractères de longueur quelconque dont tous les caractères sont des minuscules.

- {maj} {min}* est l'ensemble des chaînes résultat de la concaténation d'une chaîne de {maj} et d'une chaîne de {min}*. Ici, il s'agit des chaînes d'au moins un caractère, dont les caractères sont des minuscules sauf le premier caractère qui est une majuscule.

- {digit}^12 est l'ensemble des chaînes de 12 caractères appartenant à {digit}, c'est-à-dire les chaînes de 12 chiffres.

- {digit} | {digit}^2 | {digit}^3 est l'ensemble des chaînes appartenant à une des trois branches de l'alternative. Ici, ce sont les chaînes à 1, 2 ou 3 chiffres.

- {char} \ {eof} est l'ensemble de tous les caractères sauf le caractère de fin de fichier.

- {delim}+ est l'ensemble des chaînes de caractères ne comprenant que des caractères de {delim} et de longueur quelconque supérieure à 1.

Le lecteur pourra vérifier que la définition des tokens correspond bien à ce que l'on peut désirer. Un "code" (numéro de SS) est une suite de 13 chiffres dont le premier est un 1 ou un 2. Un "nombre" est une suite de 1, 2 ou 3 chiffres (ce qui est assez bien adapté à un âge). Un identificateur (nom, prénom) est une suite de lettres ou de "-" commençant par une majuscule et dont chaque "-" est suivi d'une majuscule, les autres lettres étant des minuscules.

Enfin, certaines notations semblent ne servir à rien, comme {null}. {null} est spécial : il modélise l'ensemble des chaînes de caractères qui délimitent les mots entre eux. Ici, ce sont les blancs, les fins de ligne, les tabulations, et les commentaires. On peut donc ajouter, selon notre grammaire, des commentaires dans le fichier d'individus, qui comme en C commencent par /* et finissent par */.

4.2.2 Analyseurs lexicaux

A partir d'une grammaire lexicale, un programme appelé analyseur lexical peut être généré. Celui-ci renvoie à chaque appel le token et la chaîne de caractères associée suivant juste ce qui a été lu au dernier appel.

De tels programmes sont en fait des interpréteurs d'automates d'états finis, car la reconnaissance d'expressions régulières peut toujours se faire par un automate d'états finis. C'est même ce qui caractérise les expressions régulières.

Des fonctionalités supplémentaires peuvent être ajoutées, comme de renvoyer un entier ou un réel si la chaîne de caractères analysée doit être comprise comme tel.

Nous avons mis en œuvre dans Shal un générateur d'analyseurs lexicaux. La méthode choisie est celle de [Aho, Sethi & Ullman 88], pp 128-146.

Remarque : On peut avoir dans une grammaire lexicale des chaînes de caractères correspondant à deux tokens à la fois. Un cas particulier est celui où une expression régulière subsume une autre expression, c'est-à-dire où elle caractérise à un sous-ensemble de chaînes de caractères. Par exemple, si nous avons les deux tokens

```
"Nom"          [N][o][m]
"identificateur"    (({maj} {letter})* [-])* ({maj} {letter})*
```

Il est clair que toute occurrence de la chaîne "Nom" (encadrée dans le texte de séparateurs admis comme tels) "matche" à la fois les tokens "Nom" et "identificateur".

Une telle situation est *permise*. Dans ce cas, l'analyseur lexical est construit de manière à renvoyer le token "Nom"[13], et non "identificateur". *L'analyseur syntaxique renvoie donc toujours le token le plus "spécifique"*.

Une difficulté survient lorsque deux expressions régulières peuvent matcher une même chaîne de caractères sans que l'une subsume l'autre. Dans ce cas, le système refuse de synthétiser un analyseur. Il pourrait aussi générer un analyseur qui renvoie le token placé en premier dans la grammaire.

5 Liens entre syntaxe et modèle conceptuel

Comme le montrent nos exemples, une grammaire bnf et un modèle conceptuel "se ressemblent". Cependant, ils ne sont pas identiques, et il serait impossible de les confondre. En effet, une bnf explicite la syntaxe, dont toutes les "scories" qui sont indifférentes au niveau conceptuel. La preuve en est qu'un même contenu conceptuel peut être représenté sous des syntaxes très différentes.

Nous décrivons ici d'abord un exemple, pour soutenir l'intuition du lecteur. Puis nous donnons la définition générale des liens. Enfin, nous commentons la manière de spécifier de tels liens par un second exemple.

---

[13] S'il en était autrement, les analyseurs lexicaux seraient d'une utilité très faible, car ils ne pourraient reconnaître aucun mot-clé.

Nous montrons au paragraphe suivant (§ 6) comment ces liens sont exprimés sur un graphe, le graphe syntaxique, et comment ils peuvent être, au moins partiellement, automatiquement reconstitués.

5.1 Exemple

Reprenons l'exemple 1, son modèle conceptuel et sa bnf.

```
1250132001003
    Toto    Fulgence    Ulysse  68  1
1491045113012
    Lulu    Achille     Carolus 42  2
1720375014276
    Zonzon  Hector              20  0
```

<Homme> <-- **@Nom "Nom"**
<-- **@Prenom &Set("Prenom")**
<-- **@Age "Age"**
<-- **@NombreDEnfants "NombreDEnfants"**
<-- **@NumeroSS "NumeroSS"**

<suiteindividu> ::= <individu> <suiteindividu>,
"";
<individu> ::= <numeross> <nom>
<suiteprenom> <age> <nbredenfants>;
<suiteprenom> ::= <prenom> <suiteprenom>,
"";
<numeross> ::= "code";
<nom> ::= "identificateur";
<prenom> ::= "identificateur";
<age> ::= "nombre";
<nbredenfants> ::= "nombre";

Modèle conceptuel et syntaxe de l'exemple

Nous écrivons désormais pour les distinguer les éléments du modèle conceptuel en gras et les éléments syntaxiques en italique.

Quelles correspondances "voit-on" ? Un homme doit correspondre à un individu dans le texte, c'est-à-dire à tout le morceau de texte le concernant[14]. Ecrivons :

<Homme> <--> *<individu>*

[14] *Attention !* Nous allons dans la suite systématiquement utiliser l'expression *morceau de texte* pour être plus intuitif. Cependant, cette expression doit être comprise comme *nœud de l'arbre syntaxique*. Cela signifie que deux occurrences *distinctes* d'une *même* chaîne de caractères ne constituent pas un *même* "morceau de texte", car elles ne peuvent pas représenter le *même* nœud de l'arbre syntaxique.

Les attributs de **<Homme>** (**NumeroSS**, etc) correspondent de manière triviale à des non-terminaux (<*numeross*>, etc). Cependant, les valeurs de ces attributs sont des entités "terminales", c'est-à-dire des renommages des entiers, réels ou chaînes de caractères. Nous devons donc plutôt les associer aux tokens "code", "identificateur" et "nombre". Mais si nous écrivons

<Homme>.Nom <--> *"identificateur"*

un problème se pose : de quel *"identificateur"* s'agit-il ?

Pour modéliser correctement les liens, nous utilisons trois opérateurs syntaxiques : /, \ et ? (nous verrons l'usage de \ et ? ensuite). L'opérateur / s'utilise comme dans le lien

<Homme>.Nom <--> *"identificateur"* / <*nom*>

et signifie “la valeur de l'attribut **Nom** d'un élément de l'ensemble d'entités **<Homme>** (le nom d'un homme, pour être plus court) est la chaîne de caractères associée au token *"identificateur"* lu après effacement d'une règle de partie gauche <*nom*>”. L'opérateur / peut être vu comme caractérisant un *chemin* dans l'arbre syntaxique construit par l'analyse : le *"identificateur"* concerné est celui qui est "sous" le nœud correspondant à <*nom*>.

Cette règle doit être précisée : on prend les chaînes de caractères appartenant au morceau de texte apparié au non-terminal auquel est associé **<Homme>**, c'est-à-dire au morceau de texte apparié à <*individu*>. Ou, si l'on veut, le token *"identificateur"* concerné doit être dans l'arbre syntaxique sous le nœud associé à <*nom*>, qui est lui-même sous le nœud associé à <*individu*>. L'écriture *correcte* serait

<Homme>.Nom <--> *"identificateur"* / <*nom*> / <*individu*>

On peut cependant se contenter de l'écriture raccourcie (cf. § 6).

Ainsi, bien que le fichier comporte plusieurs individus (éléments de **<Homme>**), et donc plusieurs noms, le bon nom sera affecté comme valeur de l'attribut **Nom**.

Les opérateurs /, \ et ? servent donc à *ôter les ambiguïtés* sur les liens en termes de chemins dans l'arbre syntaxique.

Enfin, pour bien préciser les choses, nous utilisons trois fonctions, notées &VL (pour valeur lexicale), &VE (pour valeur entière) et &VR (pour valeur réelle), qui disent de prendre la chaîne de caractères, l'entier ou le réel associé au token qui est son argument.

Finalement, nous écrivons :

<Homme> <--> <*individu*>
"Nom" <--> &VL(*"identificateur"* / <*nom*>)
"Prenom" <--> &VL(*"identificateur"* / <*prenom*>)

"Age" <--> &VE(*"nombre"* / <*age*>)
"NumeroSS" <--> &VL(*"code"*)
"NombreDEnfants" <--> &VE(*"nombre"* / <*nbredenfants*>)
<Homme>.Nom <--> &VL(*"identificateur"* / <*nom*>)
<Homme>.Prenom <--> &VL(*"identificateur"* / <*prenom*>)
<Homme>.Age <--> &VE(*"nombre"* / <*age*>)
<Homme>.NombreDEnfants <-->
&VE(*"nombre"* / <*nbredenfants*>)
<Homme>.NumeroSS <--> &VL(*"code"*)

Remarques :

- On n'a pas utilisé d'opérateur de chemin / pour **<Homme>.NumeroSS**, car il n'y a pas d'ambiguïté avec *"code"*, qui n'apparaît qu'une fois dans la grammaire. On aurait toutefois pu préciser (et il aurait mieux valu pour plus de clarté) en écrivant &VL(*"code"* / <*numeross*>).

- On a froidement écrit un lien semblable aux autres pour l'attribut **Prenom**. Et on a eu raison ! En effet, la grammaire précise bien qu'il y a plusieurs <*prenom*> sous <*individu*> (via la récursion sur <*suiteprenom*>). Ils seront *tous* versés dans l'ensemble valeur de l'attribut **Prenom** de **<Homme>**.

- Remarque à la remarque précédente : l'ordre dans lequel les prénoms sont écrits est "perdu" lorsque l'on passe au niveau conceptuel, puisque nous y avons utilisé l'opérateur **&Set**. Or, on peut vouloir distinguer premier, second, etc prénoms. On peut alors plutôt utiliser l'opérateur conceptuel **&List**, et les prénoms seront alors rangés dans la liste *dans l'ordre où ils sont lus*.

- Nous verrons au § 6 que certains liens peuvent être laissés implicites, et qu'ils seront retrouvés par le système qui y est décrit. Ainsi, il n'est pas obligatoire de définir des liens pour *tous* les ensembles d'entités, *tous* les attributs conceptuels, *toutes* les portées. Dans l'exemple ci-dessus, les liens sur les ensembles d'entités auraient suffi, le système aurait reconstitué les liens sur les attributs conceptuels.

5.2 Définition des liens

5.2.1 Constituants et forme d'un lien

Du côté conceptuel, chaque ensemble d'entités du modèle conceptuel peut être lié à un non-terminal ou un token de la bnf, de même que la valeur de chaque attribut conceptuel. Les ensembles d'entités terminaux doivent être liés à des tokens. Enfin, la portée d'un ensemble d'entités (l'exemple précédent ne comportait pas de portée) peut également être liée à un non-terminal.

Du côté syntaxique, trois fonctions et trois opérateurs peuvent être utilisés. Les trois fonctions sont &VL, &VE et &VR qui prennent la valeur lexicale d'un token, c'est-à-

dire respectivement la chaîne de caractères, l'entier ou le réel associé à un token. Les trois opérateurs sont /, \ et ? .

Une expression de lien comporte l'utilisation de l'opérateur de lien <-->.

On trouve à gauche de <--> une expression d'une des cinq sortes :

(1) **<E>**
(2) **<E>**.**Attc**
(3) **<F>**in**<E>**
(4) **<E>**=**&Set**(**<F>**) ou **<E>**=**&List**(**<F>**)
(5) Portée(**<F>**)

<E> est un ensemble d'entités et **Attc** un de ses attributs conceptuels. **<F>** dans l'expression (3) est un ensemble d'entités mentionné dans l'expression définissant **<E>** comme réunion : **<E>** = **<F>** È **…**. Portée(**<F>**) se réfère à la portée associée à une clé bien déterminée de **<F>**. S'il peut y avoir ambiguïté, cette clé doit être explicitement mentionnée.

On appelle ces expressions *expressions conceptuelles*.

On dit qu'une expression conceptuelle a pour *domaine* un ensemble d'entités **<G>** si

- Cas (1) : **<G>** = **<E>**
- Cas (2) : **<G>**, **&List(<G>)** ou **&Set(<G>)** selon le cas est le domaine des valeurs de **Attc**
- Cas (3) et (4) : **<G>** = **<F>**
- Cas (5) : **<G>** = **<F>**

Dans les cas (2), (3) et (4), on appelle **<E>** la *source* de l'expression conceptuelle. Dans le cas (1), on considérera que la source est l'ensemble d'entités de départ. Dans le cas (5), la source est l'ensemble d'entités pivot de la portée de **<F>** considérée.

A sa droite, l'expression la plus générale admise est

(i) $<t_1> / \ldots / <t_n>$? $<t'_1> \setminus \ldots \setminus <t'_m>$? $<t''_1> / \ldots / <t''_p>$

Les $<t_i>$, les $<t'_j>$ et les $<t''_k>$ sont des termes (non-terminaux ou tokens). Dans cette expression, $<t_n>$, $<t'_1>$ et $<t''_1>$ doivent être les *mêmes* termes.

Appelons $expr_1$, $expr_2$, $expr_3$ les trois sous-expressions séparées par les deux ? . On les appelle respectivement *ascendance*, *descendance* et *référence*. $expr_2$ et $expr_3$ sont en fait optionnels (c'est-à-dire que l'on peut avoir m=0 ou p=0, ou les deux). Par contre, on doit avoir n=1. On peut donc avoir une des trois expressions

(ii) $<t_1> / \ldots / <t_n>$? $<t'_1> \setminus \ldots \setminus <t'_m>$
(iii) $<t_1> / \ldots / <t_n>$? $<t''_1> / \ldots / <t''_p>$

(iv) $<t_1>$ / … / $<t_n>$

Lorsque l'opérateur ? est mentionné, on dit qu'on a une expression *en triangle*.

On appelle $<t_1>$ le *terme-domaine* de l'expression syntaxique. On appelle $<t'_m>$ (cas (i) et (ii)) ou $<t_n>$ (cas (iii) et (iv)) son *terme-source*.

De plus, si $expr_2$ est présente, alors m=2 et si $expr_3$ est présente alors p=2. Seuls le terme-domaine et le terme-source peuvent être des tokens. Si le terme-domaine est un token, alors l'expression doit être englobée dans une des trois fonctions &VL, &VE ou &VR.

Ces expressions sont appelées *expressions syntaxiques*.

D'autres contraintes pèsent sur les expressions syntaxiques et leurs liens avec les expressions conceptuelles :

- Une expression conceptuelle de la forme **<E>** ne peut être liée qu'à une expression syntaxique de la forme (iv).

- Soient **<F>** le domaine et **<E>** la source d'une expression conceptuelle **expr** (hors cas (1)), et *expr*, $expr_F$ et $expr_E$ les expressions syntaxiques auxquelles sont liés **expr**, **<F>** et **<E>**. Le terme-domaine de *expr* doit être le terme-domaine de $expr_F$, et le terme-source de *expr* doit être le terme-domaine de $expr_E$.

- Si une expression conceptuelle a pour domaine (resp. source) un ensemble d'entités terminales, alors l'expression syntaxique à laquelle elle est liée doit avoir comme terme-domaine (resp. terme-source) un token.

L'apparition de la portée peut paraître ici assez curieuse. En effet, nous avons défini la portée d'un ensemble d'entités E comme un quadruplet (S,C,C,B), où S est un ensemble d'entités, C un chemin sans circuit de S vers E, C un ensemble de chemins de S vers E, et B un circuit élémentaire de S vers S. Les liens sur la portée dont nous autorisons ici la spécification se limitent à définir le chemin de déclaration C. Les autres éléments de la portée seront reconstitués par IoLink (comme d'ailleurs en général la portée toute entière).

Les contraintes s'expliquent intuitivement par les raisons suivantes. Le domaine d'une expression conceptuelle correspond au terme-domaine de l'expression syntaxique de droite, et la source (quand elle existe) au terme-source. Le reste de l'expression syntaxique n'est là que pour préciser un chemin dans l'arbre syntaxique. Ce chemin, composé de l'ascendance et de la descendance, doit joindre ce à quoi correspondent la source et le domaine de l'expression conceptuelle, c'est-à-dire le terme-source au terme-domaine. La référence précise le contexte dans lequel le lien est valable.

5.2.2 Sémantique des liens

Nous expliquons maintenant le sens de ces expressions. Nous commençons par la partie gauche (conceptuelle) en supposant simplement que son lien avec la partie droite assure au moment de la lecture ou de l'écriture un lien avec un morceau de texte (c'est-à-dire, répétons-le, avec un nœud de l'arbre syntaxique). Nous décrivons ensuite la partie droite, c'est-à-dire quand et comment ce lien avec un morceau de texte est assuré.

Pour aider à la compréhension, nos explications mentionnent souvent la lecture. Tout reste valable pour l'écriture.

- Si l'on a à gauche une expression de la forme **<E>**, alors le lien caractérise les *morceaux de texte* associés aux entités de **<E>**. Plus précisément, lorsqu'un morceau de texte *m* a été trouvé qui matche l'expression de droite, alors une nouvelle entité **e** de **<E>** est susceptible d'être "créée". Elle est "lue" dès que le morceau de texte *m* est lu.

  Toutefois, cette "création" est soumise à notre définition de l'égalité. Il est en effet possible que l'on dispose déjà d'entités (lues ou autres) de **<E>**. Avant de créer la "nouvelle" entité **e**, il va falloir savoir si elle "existe déjà". Pour cela, il faudra connaître ses éléments constituants (clé et portée), qui sont déterminés durant la lecture du texte par les autres liens. Précisons qu'au moment où le morceau de texte *m* est lu, ces éléments constituants ne sont pas, eux, nécessairement lus. La "création" sera donc "en attente".

- Considérons maintenant une expression à gauche de la forme **<E>.Attc** de domaine **<F>**. Considérons un morceau de texte *m* matchant l'expression syntaxique à laquelle elle est liée, et soient *md* et *ms* les morceaux de texte matchant respectivement son terme-domaine et son terme source. Le morceau de texte *ms* doit correspondre à une entité **e** de **<E>** et *md* à une entité **f** de **<F>**. **f** est alors valeur de l'attribut **Attc** de l'entité **e**. Comme précédemment, la détermination exacte des entités **e** et **f** peut être "en attente", et donc l'affectation aussi.

- Le cas **<E>**in**<F>** revient au précédent en considérant l'inclusion comme un attribut conceptuel un peu à part.

- Le cas **<E>**=**&Set**(**<F>**) ou **<E>**=**&List**(**<F>**) décrit les éléments de l'ensemble ou de la liste de **<E>**.

- Si nous avons à gauche une expression de la forme Portée(**<F>**), les choses sont assez similaires, **f** est l'entité de **<F>** et **e** l'entité pivot de la portée, c'est-à-dire p_A(**f**).

Remarque : Pour les expressions de la forme **<E>.Attc**, notre définition dépend de la *multiplicité* du domaine de **Attc**. Lorsque le domaine de **Attc** est **<F>** et que **<F>** est lié par un autre lien, alors notre définition est inchangée. Mais lorsque le domaine est de la forme **&Set**(**<F>**) ou **&List**(**<F>**), et que **&Set**(**<F>**) ou

&List(<F>) n'est pas explicitement lié, on considère alors que les entités lues sont celles de **<F>**. On peut évidemment dans ce cas en lire plusieurs.

Décrivons maintenant le sens des expressions syntaxiques, et ce que signifie le fait qu'un morceau de texte matche cette expression ou une de ses sous-expressions.

- Lorsque l'expression se limite à un seul terme (token ou non-terminal), alors tout nœud de l'arbre syntaxique libellé par ce terme matche l'expression.

- Si l'expression est de la forme <*t*> / <*nt*> (<*t*> est un token ou un non-terminal), alors un nœud de l'arbre syntaxique libellé par <*t*> et descendant dans cet arbre d'un nœud libellé par <*nt*> matche l'expression. Autrement dit, le morceau de texte correspondant à <*t*> est *contenu* dans un morceau de texte associé à <*nt*>.

- Lorsqu'une suite d'opérateurs / est utilisée, par exemple dans $<t_1>$ / … / $<t_n>$, chaque terme doit être associé à un morceau de texte contenu dans le suivant (ou, en d'autres mots, les nœuds correspondants dans l'arbre syntaxique sont descendants des suivants).

- Regardons maintenant l'opérateur \. Il a le "même" sens que l'opérateur /, sinon qu'il prend les arcs de l'arbre syntaxique dans le sens opposé. Ainsi, une expression de la forme <*nt*> \ <*t*> correspond aux morceaux de texte associés à <*nt*> qui *contiennent* un morceau de texte associé à <*t*>. Autrement dit, dans l'arbre syntaxique, le nœud correspondant libellé par <*nt*> est un *ascendant* d'un nœud libellé par <*t*>.

- Comme pour l'opérateur /, une chaîne d'opérateurs \, comme dans $<t'_1>$ \ … \ $<t'_m>$ correspond à une suite de nœuds de plus en plus "bas" dans l'arbre syntaxique.

- L'expression $<t_1>$ / … / $<t_n>$? $<t'_1>$ \ … \ $<t'_m>$ est matchée par les morceaux de texte correspondant à $<t_1>$, contenus dans une suite croissante de textes correspondant à $<t_2>,\ldots,<t_n>$, celui correspondant à $<t_n> = <t'_1>$ contenant des morceaux de texte contenant le suivant et correspondant à $<t'_2>,\ldots<t'_m>$. On a donc dans l'arbre syntaxique un chemin remontant par des nœuds libellés par $<t_1>$, …, $<t_n>$, puis redescendant par des nœuds libellés par $<t'_1>,\ldots,<t'_p>$.

- Enfin, quand une référence $<t''_1>$ / … / $<t''_p>$ est mentionnée, le morceau de texte correspondant à $<t_n> = <t''_1>$ doit être contenu dans une chaîne de morceaux de texte inclus dans le suivant et libellés par $<t''_2>,\ldots,<t''_p>$. Dans le cas le plus général, on a dans l'arbre syntaxique une "étoile" à trois branches (les trois chemins), dont les sommets forment donc un triangle.

Le lecteur pourra se reporter à l'exemple du § 5.2.1 pour constater que les liens qui y sont définis "informellement" correspondent bien aux définitions présentées ici.

5.3 Second exemple

Nous traitons ici un autre exemple, l'exemple 2, qui utilise les opérateurs \ et ? . Nous verrons aussi comment et pourquoi le modèle conceptuel doit quelquefois être complété pour préciser ses liens avec une syntaxe.

5.3.1 Liens entre le modèle conceptuel et la syntaxe de l'exemple 2

Nous avons vu que, sous des syntaxes différentes, les textes de l'exemple 1 et de l'exemple 2 contenaient les mêmes informations. Ils ont un modèle conceptuel commun.

Conformément à ce que nous avions annoncé, on peut définir les liens entre ce modèle conceptuel et la syntaxe de l'exemple 2. Nous utilisons pour la première fois les opérateurs \ et ? . Ces liens sont présentés ici sous leur forme "complète". Nous verrons au § 6 que nous aurions pu laisser beaucoup de choses implicites.

```
$1  Nom Toto
$1  NumeroDeSecuriteSociale 1250132001003
$1  Prenom Fulgence
$1  Prenom Ulysse
$1  Age 68
$1  NombreDEnfants 1

$2  Nom Lulu
$2  NumeroDeSecuriteSociale 1491045113012
$2  Prenom Achille
$2  Prenom Carolus
$2  Age 42
$2  NombreDEnfants 2

$3  Nom Zonzon
$3  NumeroDeSecuriteSociale 1720375014276
$3  Prenom Hector
$3  Age 20
$3  NombreDEnfants 0
```

<bf> ::= <fait> <bf> ,
""";
<fait> ::= <fnom>, <fprenom>, <fnss>, <fage>, <fnenf>;
<fnom> ::= <id1> "Nom" <nom> ;
<fprenom> ::= <id1> "Prenom" <prenom> ;
<fnss> ::= <id1> "NumeroDeSecuriteSociale" <nss> ;
<fage> ::= <id1> "Age" <age> ;
<fnenf> ::= <id1> "NombreDEnfants" <nenf> ;
<id1> ::= "identificateur" ;
<nom> ::= "identificateur" ;
<prenom> ::= "identificateur" ;
<nss> ::= "code" ;

<age> *::=* *"nombre" ;*
<nenf> *::=* *"nombre" ;*

<Homme> <--> *<id1>*
"Nom" <--> &VL(*"identificateur"* / *<nom>*)
"Prenom" <--> &VL(*"identificateur"* / *<prenom>*)
"Age" <--> &VE(*"nombre"* / *<age>*)
"NumeroSS" <--> &VL(*"code"* / *"nss"*)
"NombreDEnfants" <--> &VE(*"nombre"* / *<nenf>* / *<fnenf>* ?
<fnenf> \ *<id1>*)
<Homme>.Nom <--> &VL(*"identificateur"* / *<nom>* /
<fnom> ? *<fnom>* \ *<id1>*)
<Homme>.Prenom <--> &VL(*"identificateur"* / *<prenom>* /
<fprenom> ? *<fprenom>* \ *<id1>*)
<Homme>.Age <--> &VE(*"nombre"* / *<fage>* ? *<fage>* \ *<id1>*)
<Homme>.NombreDEnfants <--> &VE(*"nombre"* /
<fnbredenfants> ? *<fnbredenfants>* \ *<id1>*)
<Homme>.NumeroSS <--> &VL(*"code"* / *<fnss>* ?
<fnss> \ *<id1>*)

L'exemple 2, sa syntaxe et les liens avec le modèle conceptuel

5.3.2 Adaptation du modèle conceptuel et des liens

Notre spécification des liens ci-dessus n'est cependant pas tout à fait complète. En effet, lorsqu'on lit deux faits épars se référant à un même individu (car on peut ici "mélanger" les informations relatives à Toto et Lulu, par exemple), il faut savoir qu'il s'agit bien du même individu. Or, celui-ci est caractérisé par une chaîne de la forme $xxx qui apparaît à gauche des faits.

Mais attention ! Dans les faits

```
$1  NumeroDeSecuriteSociale 1250132001003
$1  Prenom Fulgence
```

on a bien la même chaîne de caractères à gauche ($1) *mais pas le même morceau de texte*. En effet, notre expression *morceau de texte* est synonyme de *nœud de l'arbre syntaxique*, or ici chaque $1 correspond à un nœud *distinct*[15].

La spécification ci-dessus n'est donc pas complète, car elle ne permet pas la lecture des entités de **<Homme>**.

Nous complétons donc le modèle conceptuel en ajoutant un attribut à l'ensemble d'entités **<Homme>**, appelons-le **Id1**, dont l'ensemble d'entités valeur est **"Id1"**, **"Id1"** étant un ensemble d'entités terminales, renommage des chaînes de caractères.

---

[15] On ne pourrait pas baser l'égalité entre entités sur l'égalité (et non l'identité) des chaînes de caractères constitutives de morceaux de texte auxquelles elles seraient associées, car l'égalité conceptuelle fait en général intervenir la portée (ce qui n'est pas le cas dans l'exemple).

Nous lions maintenant cet attribut conceptuel à la syntaxe de l'exemple 2 par les liens

"Id1" <--> &VL(*"identificateur"* / <*id1*>)
<Homme>.Id1 <--> &VL(*"identificateur"* / <*id1*>)

(le deuxième lien peut se déduire du premier, cf. § 6).

De plus, nous déclarons que l'ensemble d'attributs conceptuels {**Id1**} constitue une *clé* (sans portée) de **<Homme>**. Ainsi, les divers nœuds de l'arbre syntaxique libellés par le même $xxx correspondront-ils bien maintenant à une même entité.

Cet exemple est caractéristique d'une situation fréquente, où il faut introduire dans le modèle conceptuel certains éléments purement syntaxiques. Cependant, le *cœur* du modèle reste inchangé et commun à toutes les syntaxes. Par ailleurs, on voit que le générateur d'entrées-sorties aura à choisir la clé ou le couple clé-portée par lequel caractériser le plus facilement et rapidement les entités "lues".

5.3.3 Remarques sur la bnf de l'exemple 2

On constate que la bnf de l'exemple 2 contient les tokens "Nom", "Prenom", etc. Ces relations sont donc considérées comme des mots-clés.

Pourtant, une base de faits comme celle de l'exemple a une syntaxe bien plus simple et plus générale, à savoir :

```
<bf>  ::=   <fait> <bf>;
<fait>      ::=   <objet> <objet> <objet>;
<objet>     ::=   "identificateur";
```

Mais la définition des liens entre le modèle conceptuel et cette syntaxe serait impossible dans notre langage.

Une solution intermédiaire a été envisagée pour résoudre ce problème, sans avoir été mise en œuvre. Elle consisterait à partir des *contraintes sémantiques* [Dormoy 90] sur une base de faits pour générer automatiquement la syntaxe de l'exemple 2. De plus, ces contraintes sémantiques pourraient elles-mêmes être définies de manière standard à partir du modèle conceptuel, et donc retrouvées automatiquement. En définitive, on aurait un moyen standard pour représenter les instances d'un modèle conceptuel dans une base de faits, et spécifier les entrées-sorties pour une telle base de faits se résumerait à dire "lis-moi (ou écris-moi) la base de faits du modèle conceptuel xxx".

Une telle solution, adaptée à Shal et Genesia II, pourrait rendre de grands services dans la construction de Descartes.

6 Reconstitution et complétion des liens

6.1 Motivation et idée générale

Nous avons mentionné que l'on pouvait se satisfaire de décrire partiellement les liens entre modèle conceptuel et syntaxe, et qu'un système, IoLink, se chargeait de les compléter (et aussi éventuellement de trouver une incohérence).

Notre volonté est en effet de permettre à l'utilisateur de spécifier ses entrées-sorties avec aussi peu de "mots" que possible. Mais dans le même temps, il faut qu'il sache quelle "dose minimale" d'information il doit fournir.
En général, il faut lier tous les ensembles d'entités "présentes dans le fichier". Par contre, les attributs conceptuels et les portées vont pouvoir dans la plupart des cas être reconstitués. Pourquoi ?

Cette reconstitution se fait en partie de façon heuristique. Mais que l'utilisateur (futur) de Io se rassure, nos heuristiques sont solidement fondées, comme on va le voir.

Tout repose sur notre constatation (expérimentale) que *tous* les langages de données et de programmation ont des syntaxes réparties dans *deux* catégories : ce que nous appelons les *syntaxes emboîtantes*, et les *syntaxes en vrac* (ou un mélange des deux, mais rarement)[16].

Les syntaxes emboîtantes sont caractérisées par le fait qu'un lien conceptuel entre deux informations contenues dans le texte se fait par *inclusion des morceaux de texte* associés aux informations. Ainsi, si on a deux ensembles d'entités **<E>** et **<F>** liés par un attribut conceptuel **Attc**, une entité **e** de **<E>** va correspondre à un morceau de texte *contenant* celui associé à sa valeur **f** par **Attc**. Dans l'expression des liens entre modèle conceptuel et syntaxe, seul l'opérateur / va (en général) être utilisé.

Les syntaxes des exemples 1, 3, 4 et 5 du § 2 sont emboîtantes. Par exemple, le morceau de texte correspondant à un sous-programme Pascal (c'est-à-dire tout le texte du sous-programme, pas simplement sa déclaration ou son nom) contient le texte de ses variables, de ses types, de ses instructions, les morceaux correspondant aux instructions contiennent ceux correspondant aux expressions qu'elles mentionnent, etc.

Par contre, la syntaxe de l'exemple 2 du § 2 est en vrac. Les attributs d'une entité de **<Homme>** sont dispersés dans des morceaux de texte qui ne sont pas contenus dans ceux correspondant à cette entité. Ici, les opérateurs \ et ? sont indispensables. Il est à noter que c'est ce genre de syntaxe en vrac qui nécessite l'ajout d'attributs dépendant de la syntaxe qui permettent de repérer sans ambiguïté les entités dont parle le texte (cf. § 5.3.2).

---

[16] Il y a d'ailleurs sans doute une raison profonde à cela, qui tient au fait que les syntaxes sont descriptibles par une grammaire context-free. Nous n'avons cependant pas à ce jour appuyé notre constatation empirique sur des arguments théoriques.

6.2 Graphe syntaxique

6.2.1 Définition

Pour ce faire, IoLink construit à partir de la grammaire bnf un graphe orienté que nous appellerons *graphe syntaxique* et noterons GS. Les nœuds de ce graphe sont les termes (non-terminaux et terminaux) utilisés dans la grammaire. On définit un arc d'un nœud-terme t_1 vers t_2 ssi t_2 apparaît à droite dans la règle définissant t_1[17].

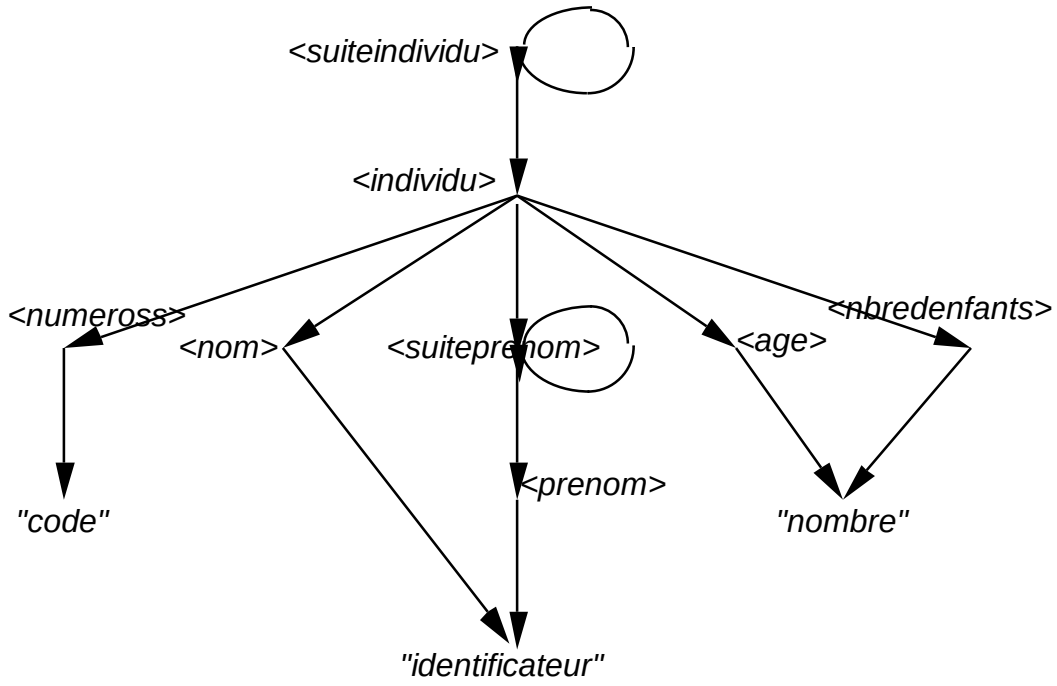

Figure 2 : Le graphe syntaxique de l'exemple 1

Un peu de terminologie. Un chemin de GS est appelé *emboîtement*. Le premier nœud par lequel un emboîtement passe est appelé sa *source*, le dernier son *domaine*. Un emboîtement peut contenir un ou plusieurs circuits. Si c'est le cas, il est dit *récursif*.

Un circuit élémentaire de GS (i.e. ne passant pas deux fois par le même nœud) est appelé *récursion*.

On dit qu'une récursion *passe* par un emboîtement ssi la récursion a la forme $[x_1,\ldots,x_k,x_{k+1},\ldots x_n,x_1]$, où tous les arcs $[x_i,x_{i+1}]$, 1=i=k-1 appartiennent à l'emboîtement, mais où aucun des $[x_i,x_{i+1}]$, k=i=n-1 ne lui appartiennent.

---

[17] Pour être parfaitement précis, il faudrait en fait définir un *hypergraphe syntaxique*, dont une arête (orientée) relierait le terme apparaissant à gauche d'une règle à tous les termes d'une des branches d'alternative à droite de la règle. La structure de cet hypergraphe conserverait ainsi les "et" et les "ou" des règles. Cependant, cette notion n'est que rarement utile, en l'occurrence pour l'analyse des multiplicités minimales des expressions conceptuelles (cf. § 6.3).

On appelle *triangle d'emboîtements* un triplet de 3 emboîtements (ε_1,ε_2,ε_3) où le domaine de ε_3 et les sources de ε_1 et ε_2 sont égaux. On admet les cas dégénérés où ε_2 ou ε_3 ou les deux sont le chemin vide. Quand les deux sont vides, on identifie le triangle avec le chemin ε_1 restant.

On notera dans la suite T l'ensemble des triangles d'emboîtements et R l'ensemble des récursions. On inclut le chemin vide dans R.

Le graphe syntaxique de l'exemple 1 est représenté dans la figure 2.

6.2.2 Graphe et arbre syntaxiques

Le graphe syntaxique d'une bnf a un lien avec un arbre syntaxique la vérifiant. Le graphe est en effet le résultat sur un tel arbre de l'opération consistant à *identifier* d'une certaine manière les nœuds libellés par le même terme de la grammaire.

Plus précisément, soit GS le graphe syntaxique et AS un arbre syntaxique. On définit une fonction f de AS dans GS (i.e. des nœuds vers les nœuds, et des arcs vers les arcs) de la manière suivante :

- A tout nœud n de AS libellé par le terme t on associe le nœud t de GS;
- A tout arc a de AS rentre nœuds libellés respectivement par n et n' et correspondant à l'application de la règle n ::= α n' β, on associe l'arc [n,n'] de GS entre les nœuds n et n' correspondant à cette même règle.

Il est clair que le graphe quotient de AS par la relation d'équivalence x R y ssi f(x) = f(y)[18] (aussi bien sur les nœuds que sur les arcs) est isomorphe à un sous-graphe de GS (l'image de f).

Cette relation entre graphe et arbre syntaxiques va nous permettre d'exprimer les liens, a priori définis par référence à l'arbre syntaxique, sur le graphe syntaxique. En effet, si une partie de l'arbre syntaxique AS (en général une étoile de 3 chemins) matche une expression syntaxique *expr*, alors son image par f dans le graphe syntaxique va correspondre à trois emboîtements éventuellement complétés de quelques circuits. Par exemple (exemple 1), le lien entre le morceau de texte correspondant à l'entité de **<Homme>** de nom `Lulu` et son prénom `Carolus` est représenté par le chemin de l'arbre syntaxique de la figure 3a. Ce chemin est transformé dans GS via f en le chemin, plus une récursion sur <*suiteprenom*>, de la figure 3b. On peut d'ailleurs mentionner la référence, c'est-à-dire le chemin partant du non-terminal de départ <*suiteindividu*> et aboutissant à <*individu*>, plus un passage par la récursion sur <*suiteindividu*>.

[18] Le lecteur vérifiera que cette relation respecte la structure de graphe.

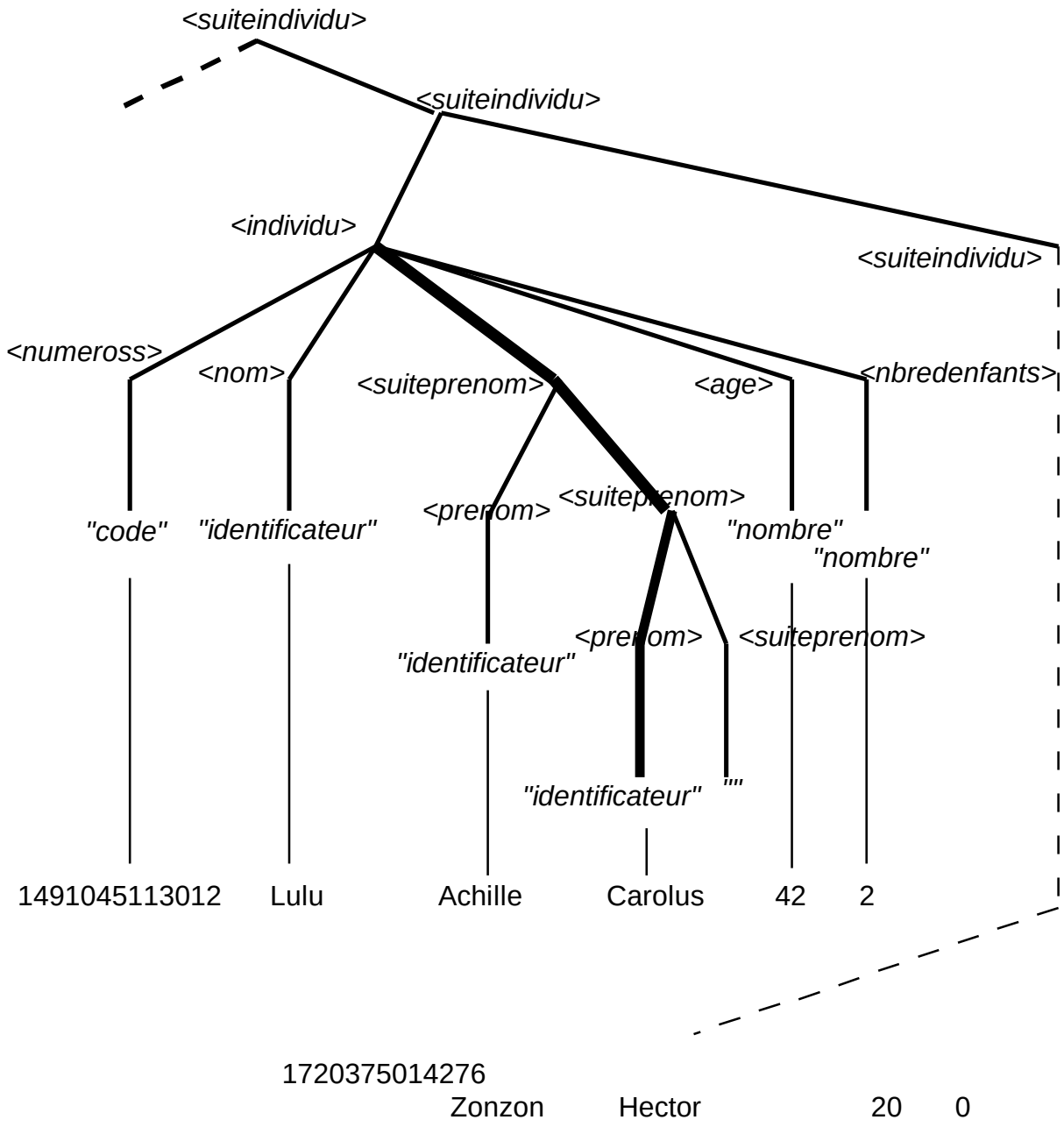

*Figure 3a : Le chemin entre l'<**Homme**> `Lulu` et son prénom `Carolus` dans l'arbre syntaxique*

En résumé, grâce à f, un chemin dans l'arbre syntaxique devient un chemin dans le graphe syntaxique passant éventuellement par certains circuits, où il "fait un certain nombre de tours".

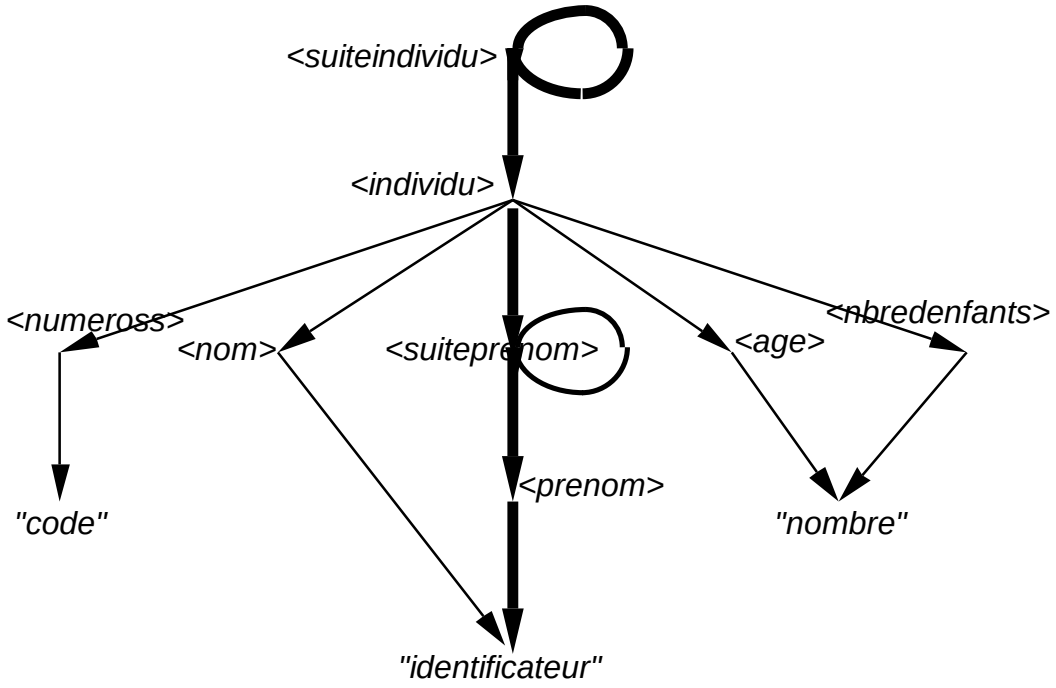

*Figure 3b : Le chemin entre l'<**Homme**> `Lulu` et son prénom `Carolus` dans le graphe syntaxique*

6.3 Expression des liens sur le graphe syntaxique

IoLink exprime les liens entre modèle conceptuel et syntaxe en termes de liens entre le modèle conceptuel et les triangles d'emboîtements et récursions du graphe.

A toute expression conceptuelle **expr**, IoLink associe une partie C(**expr**) de T x P(R), c'est-à-dire des triangles d'emboîtements joints à un ensemble de récursions :

C : **Expression Conceptuelle** --> P(T x P(R))
expr --> C(**expr**)

Les triangles d'emboîtements dans C(**expr**) sont en relation directe avec les expressions syntaxiques à droite des liens, et les récursions avec une notion que nous allons introduire (§ 6.3.2), la *multiplicité* d'une expression conceptuelle.

6.3.1 Ensemble d'emboîtements correspondant à un lien

Nous définissons d'abord une fonction auxilliaire, T(**expr**), qui associe à toute expression conceptuelle **expr** un ensemble de triangles d'emboîtements, et telle que T(**expr**) sera égal à la première projection de C(**expr**).

Supposons que le lien suivant soit donné.

expr <--> $<t_1>$ / … / $<t_n>$? $<t'_1>$ \ … \ $<t'_m>$? $<t''_1>$ / … / $<t''_p>$

Nous passons ici sous silence l'éventuelle mention des fonctions &VL, &VR, &VE.

On dit qu'un emboîtement *matche* l'expression syntaxique $<t_1>$ / … / $<t_n>$ (resp. $<t'_1>$ \ … \ $<t'_m>$) ssi il passe successivement par les nœuds $<t_n>$, …, $<t_1>$ (resp. $<t'_1>$, …, $<t'_m>$) et s'il est minimal avec cette propriété. Cette propriété signifie qu'il peut exister des récursions dans ce chemin (car un même terme peut se répéter dans l'expression syntaxique), mais qu'il doit y en avoir le moins possible. En particulier, s'il n'y a pas répétition de terme, alors l'emboîtement est non récursif.

Ecrivons l'expression de droite sous la forme $expr_1$? $expr_2$? $expr_3$.

T(**expr**) est défini comme étant l'ensemble des triangles d'emboîtement (ε_1,ε_2,$\varepsilon.\varepsilon_3$) tels que chaque ε_i matche chaque expression syntaxique $expr_i$ et où ε est n'importe quel emboîtement non récursif joignant le non-terminal de départ à $<t''_p>$ si $expr_3$ n'est pas vide ou à $<t_n>$ sinon. Si l'expression $expr_2$ (resp. $expr_3$) est vide, alors ε_2 (resp. ε_3) est vide. $\varepsilon.\varepsilon_3$ est la concaténation des chemins ε et ε_3.

Remarque : Cette définition impose une contrainte encore non énoncée sous cette forme sur la définition des liens : les emboîtements constituant les triangles associés à **expr** doivent *exister*.

6.3.2 Multiplicité d'une expression conceptuelle

Pour pouvoir définir C(**expr**), c'est-à-dire les récursions associées à T(**expr**), il faut tout d'abord définir la multiplicité d'une expression conceptuelle. Nous donnons ici une définition complète, dont une partie seulement sert à la définition de C(**expr**). L'autre partie sert à la reconstitution des liens par IoLink.

Les récursions adjointes dépendent de la *multiplicité* de l'expression conceptuelle **expr**. Les expressions du type **<E>=&Set(<F>)** ou **<E>=&List(<F>)** et Portée(**<F>**) sont dites *multiples*, les autres pas.

La multiplicité a en fait une définition plus générale, dont nous nous servirons ensuite. Lorsqu'une expression conceptuelle lie deux ensembles d'entités **<E>** et **<F>** (la source et le domaine), la multiplicité caractérise le nombre d'entités **f** de **<F>** qui doivent être mises en relation avec une même entité **e** de **<E>**. Cette caractérisation se fait dans les mêmes termes que les contraintes imposées à une branche d'une association dans le modèle conceptuel. On a une cardinalité minimale, qui est un entier naturel (le plus souent 0 ou 1), et une cardinalité maximale, qui est un entier non nul ou n. On ne considère ici que quatre cas de multiplicité : (1,1), (0,1), (1,n), (0,n). La multiplicité associée à une expression conceptuelle est :

- **<E>** (0,n)
- **<E>.Attc** max=1 si le domaine de **Attc** n'a pas la forme **&Set(<F>)** ou **&List(<F>)**, ou s'il a cette forme si l'expression **&Set(<F>)** ou **&List(<F>)** est explicitement liée,

max=n si le domaine a la forme **&Set(<F>)** ou **&List(<F>)**, et si l'expression **&Set(<F>)** ou **&List(<F>)** n'est pas explicitement liée,
min=0, sauf si **Attc** fait partie de la dimension de **<E>**, auquel cas min=1

- **<F>**in**<E>** (0,1)
- **<E>**=**&Set(<F>)** (0,n)
- Portée(**<F>**) (0,n)

L'interprétation de la multiplicité est la suivante. Soient **expr** une expression conceptuelle, **<E>** sa source et **<F>** son domaine. Si min=1, alors il doit y avoir dans un texte mentionnant une entité **e** de **<E>** au moins une entité correspondante **f** de **<F>**. Cela signifie qu'il doit y avoir dans le graphe syntaxique GS un triangle d'emboîtements conduisant du terme $expr_E$ associé à **<E>** vers le terme associé à **<F>**, ceci quelles que soient les règles utilisées pour analyser $expr_E$. Si max=n, il doit y avoir pour toute occurrence **e** de **<E>** potentiellemment plusieurs entités correspondantes **f** de **<F>**. Cela signifie qu'il doit exister dans GS une récursion liant les termes liés respectivement à **<E>** et **<F>**.

Quelques commentaires. L'expression conceptuelle **<E>** est vue comme liant l'ensemble d'entités de départ à **<E>**, et il est donc normal d'avoir une multiplicité (0,n) (i.e., on peut ne trouver dans le texte aucune occurrence d'entités de **<E>**, ou au contraire plusieurs). Quant à portée de **<E>**, elle *doit* passer par une boucle de récursion (i.e. la boucle d'emboîtement passant par le pivot), d'où une multiplicité maximale n.

6.3.3 Récursions adjointes aux emboîtements

Nous pouvons maintenant définir C. Pour chaque élément de T(**expr**), il va falloir déterminer quelle(s) récursion(s) lui adjoindre pour former un couple de C(**expr**). Soit donc un chemin (ε_1,ε_2,$\varepsilon.\varepsilon_3$) de T(**expr**).

Dans tous les cas, sont adjointes :

- les récursions passant par ε et n'ayant aucun nœud commun avec les ε_i, sauf éventuellement la source de ε_3 ou la source de ε_1 si ε_3 est vide;

- les récursions passant par ε_2 et n'ayant aucun nœud commun avec les autres, sauf éventuellement la source de ε_1.

Lorsque l'expression n'est pas multiple, les seules récursions adjointes sont celles-là.

Lorsque l'expression est multiple, on ajoute à ces récursions :

- les récursions passant par ε_1 et n'ayant aucun nœud commun avec les autres chemins autre que la source de ε_1.

6.3.4 Exemple

Dans l'exemple 1, l'expression conceptuelle **<Homme>.Prenom** est liée à l'expression syntaxique &VL(*"identificateur"* / <*prenom*> / <*individu*>). T(**<Homme>.Prenom**) contient l'unique triangle d'emboîtements où

ε_1 = [<*individu*>,<*suiteprenom*>,<*prenom*>,*"identificateur"*]
ε_2 = []
ε_3 = []
ε = [<*suiteindividu*>,<*individu*>]

Les récursions associées sont

[<*suiteindividu*>,<*suiteindividu*>]
[<*suiteprenom*>,<*suiteprenom*>]

la première à cause des conditions générales, la seconde parce que l'expression **<Homme>.Prenom** est considérée comme multiple (pas de lien sur **&Set**(**"Prenom"**)).

6.4 Découverte des liens sur le graphe syntaxique

Lorsque les liens sont partiellement spécifiés, IoLink tente de les compléter. IoLink commence par construire les ensembles d'expressions syntaxiques susceptibles d'être liées à une expression conceptuelle non liée, puis tente d'éliminer ces expressions (sauf une) par divers moyens de propagation de contraintes. Ce faisant, il construit les C(**expr**) qui permettront ensuite à IoIn ou IoOut de construire les programmes d'entrées-sorties. De plus, le processus produit en quelque sorte par effet de bord une vérification de cohérence des liens fournis ou trouvés.

Ainsi, IoLink peut fournir trois types de réponse : les liens sont trouvés et satisfaisants, il reste une ambiguïté (plusieurs possibilités de liens), ou un lien est impossible, ce qui signifie que la spécification est contradictoire. Dans le second cas, l'utilisateur devra mieux spécifier ses liens, dans le troisième, il devra corriger une erreur dans la spécification.

Nous décrivons maintenant plus en détail les actions conduites par IoLink. Certaines sont accompagnées d'exemples, celles qui ne le sont pas nécessitant des cas plus complexes que les exemples mentionnés dans cette note.

- IoLink établit d'abord les expressions conceptuelles pour lesquelles un lien doit être trouvé. Il utilise ici les clés et les portées : pour chaque ensemble d'entités susceptible d'apparaître dans le texte, au moins un couple (clé,portée) - avec éventuellement une portée vide - doit pouvoir être lié. Les attributs mentionnés dans une dimension doivent aussi pouvoir être liés. Enfin, il faut lier les expressions de la forme **<F>**in**<E>** où **<E>** est susceptible d'apparaître dans le texte.

- L'expression syntaxique à droite du lien est complétée pour faire apparaître un terme-source et un terme-domaine compatibles avec les autres liens. Si aucune expression syntaxique n'est fournie dans la spécification, IoLink propose celles qui sont compatibles avec ce qu'il sait déjà, et poursuit ses recherches pour chaque expression ainsi proposée. Le but est de toutes les éliminer, sauf une.

  Exemple : Si on a fourni le lien **<Homme>.Prenom** <--> &VL(*"identificateur"* / <*prenom*>), celui-ci est complété en **<Homme>.Prenom** <--> &VL(*"identificateur"* / <*prenom*> / <*individu*>).

- C(**expr**) est calculé pour chaque expression syntaxique candidate (on calcule donc en fait C'(**expr**,$expr_i$) pour chaque expression syntaxique $expr_i$ candidate).

- Les emboîtements et les récursions de C'(**expr**,$expr_i$) sont examinés, et éventuellement éliminés, pour rester compatibles avec la multiplicité de **expr** et les autres liens déjà déterminés de façon sûre.

- Principe général d'élimination : On préfère systématiquement les liens "droits", i.e. avec ε_2 et ε_3 vides. Donc, tant qu'un tel lien subsiste (n'est pas éliminé), les autres ne sont pas considérés. Ils ne le seront que si tous les liens "droits" ont été éliminés (ou s'il n'y en a jamais eu).

Nous donnons maintenant quelques principes d'élimination :

- Une expression multiple doit être liée à une récursion "bien placée", c'est-à-dire dans ε_1, ou sinon dans ε, avec ε_2 non vide.

  Exemple : Si aucun lien n'a été fourni, ni pour **"Prenom**", ni pour **<Homme>.Prenom**, alors IoLink va commencer par lier **"Prenom"** à *"identificateur"*. Les liens "droits" sont préférés. Parmi ceux-ci, seul l'emboîtement passant par <*suiteprenom*> comporte une récursion assurant la multiplicité de l'expression **<Homme>.Prenom**. Il est donc le seul conservé à ce stade. S'il n'est pas éliminé par la suite, c'est lui qui sera considéré comme le "bon" lien. Cela permet de compléter le lien sur **"Prenom"** en &VL(*"identificateur"* / <*suiteprenom*>) (qui est équivalent à &VL(*"identificateur"* / <*prenom*>))[19].

- C'est dans cette phase d'élimination que la multiplicité minimale de l'expression à lier est utile. Considérons *l'hypergraphe syntaxique* défini dans la note du § 6.2. Pour qu'une multiplicité minimale égale à 1 soit respectée, il faut que *tous* les chemins (dans l'hypergraphe) partant du terme-source associé à l'expression conceptuelle passent par le terme-domaine. Cela permet d'éliminer des expressions syntaxiques candidates.

---

[19] Parmi les fonctions lexicales, &VL est choisie par défaut.

Il peut arriver malgré tout qu'après la phase d'élimination plusieurs possibilités de liens à des expressions syntaxiques subsistent. Dans certains cas, IoLink utilise des heuristiques pour choisir (en éliminant les autres possibilités).

Ces heuristiques concernent le cas où il ne reste que des expressions vraiment en triangle (i.e. mentionnant les opérateurs ?). Pour voir comment ces heuristiques procèdent, il nous faut définir une relation entre triangles d'emboîtements. Un triangle (ε_1,ε_2,ε_3) est dit *plus haut* qu'un triangle (ε'_1,ε'_2,ε'_3) ssi il existe un emboîtement non récursif ε tel que $\varepsilon'_3 = \varepsilon_3.\varepsilon$. Une expression syntaxique $expr_1$ possiblement liée à **expr** sera dite plus haute qu'une autre expression syntaxique $expr_2$ possiblement liée à la même expression **expr** ssi tout emboîtement de C'(**expr**,$expr_1$)est plus haut qu'au moins un emboîtement de C'(**expr**,$expr_2$).

Les heuristiques sont :

- Pour une expression de la forme **<E>.Attc**, on préfère les expressions les plus basses

- Pour une expression de la forme Portée(**<F>**), on préfère les expressions les plus hautes.

Nous ne justifierons pas ici ces heuristiques, qui d'ailleurs ne se manifestent que dans des cas assez complexes (représentation dans une base de faits d'un programme). Elles ont toujours conclu jusqu'à présent de manière correcte. De toute façon, elles signalent leur application.

Evidemment, si ces heuristiques ne s'appliquent pas, ou si elles ne parviennent pas à ne laisser qu'une possibilité, IoLink conclura à une ambiguïté des liens.

7 Conclusion

Nous ne prétendons pas avoir avec Io complètement résolu le problème des entrées-sorties. Cependant, nous croyons l'avoir résolu dans la plupart des cas pratiques. Si cela s'avérait nécessaire, certaines extensions, comme la prise en compte d'expressions syntaxiques mentionnant plus de deux opérateurs ?, pourraient être réalisées. On pourrait aussi sans doute améliorer IoLink.

Mais les choses semblent déjà assez complexes comme cela, et surtout, permettent d'ores et déjà de traiter une gamme de textes informatiques assez large (cf. nos exemples).

L'avenir nous dira, lorsque le système Io aura été complètement spécifié et mis en œuvre, si ce langage, comme nous le croyons, tient bien ses promesses.

Références

[Aho, Sethi & Ullman 88] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers : Principles, Techniques and Tools*. Addison-Wesley, reprinted and corrected in 1988.

[Dormoy 90] Jean-Luc Dormoy. *Représentation et utilisation des connaissances : contraintes sémantiques sur une base de faits Shal*. Note EDF HI21/7185.

[Dormoy 91] Jean-Luc Dormoy. *Compilation de connaissances par des connaissances : le système Shal*. Note EDF HI21/7450 et Revue de l'Intelligence Artificielle, 1991.

[Dormoy 93] Jean-Luc Dormoy. *Définition du langage ErgoAlg*. Note EDF HI21/8340.

[Ginoux 91] Bruno Ginoux. *Génération automatique d'algorithmes par système expert à partir de spécifications de haut niveau : le système Cogito*. Thèse de l'Université Paris-Dauphine, octobre 1991.

[Lagrange 92] Jean-Philippe Lagrange. *OGRE : un système expert pour la génération de requêtes relationnelles*. Thèse de l'Université Paris-Dauphine, décembre 1992.