PRESENTATION DE STARLET/GL

IF

Créé en 1989
Révisé en 2001
Année scolaire 2008 – 2009

Cours

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

FORMATION

Auteur de la Ressource Pédagogique
BENEY Jean

membre de UNIVERSITÉ DE LYON

INSA LYON

Département Informatique de l'INSA de Lyon

PRESENTATION DE STARLET/GL

Institut National des Sciences Appliquées de Lyon
Laboratoire d'Ingénierie des Systèmes d'Information
Bâtiment 502
F-69621 VILLEURBANNE Cedex
Tél. 78.94.83.05

Juillet 1989

Révisé en Mars 2001

Sommaire

1 Introduction

Ce document reprend les présentations du système STARLET/GL parues en plusieurs parties dans le bulletin BULLET (Bulletin des utilisateurs et développeurs de LET et STARLET, n° 2 à 6).
STARLET/GL est la première version du Système de Traitement Automatique des Règles du Langage d'Ecriture de Traducteurs (en abrégé STARLET ou *LET) développé à la suite de LET pour des applications de Génie Logiciel. Après une présentation générale de ses caractéristiques, nous introduirons progressivement la méthode de programmation en *LET.

1.1 But

Dans la lignée de LET, ce nouveau langage a été conçu pour faciliter la réalisation de compilateurs et d'interprètes. C'est-à-dire qu'il doit permettre plus que l'écriture d'accepteurs ou d'analyseurs syntaxiques, par la prise en compte conjointement des aspects syntaxiques et sémantiques.

Plus généralement, *LET permet le développement de tout programme considéré comme une traduction de données structurées en des résultats déduits de ces données. Il permet ainsi de mettre en œuvre une méthode de programmation grammaticale par raffinements successifs.

1.2 Idées conductrices

*LET veut permettre des développements rapides et fiables de programmes efficaces.

Rapidité de développement :
*LET est basé sur l'exploitation automatique d'un formalisme grammatical adapté. Ainsi, la spécification grammaticale de la traduction à réaliser suffit à définir le programme qui la réalise. L'utilisateur peut alors considérer son texte comme une grammaire de traduction ou comme un programme.
Fiabilité :
La fiabilité d'un compilateur ne peut reposer sur des essais, même innombrables (cf. ADA), mais doit être assurée au maximum par la vérification automatique de la description de ce compilateur. Le vérificateur *LET contrôle donc la cohérence grammaticale et algorithmique des spécifications qui lui sont soumises, assurant ainsi, une certaine fiabilité a priori du traducteur fabriqué.
Efficacité :
Bien qu'étant un langage de programmation logique, *LET se veut efficace. C'est pourquoi il est doté d'emblée d'un compilateur-optimiseur ; l'interprète *LET n'est qu'un outil de prototypage facilitant la démonstration des programmes. Cet interprète bénéficie d'ailleurs de certaines optimisations effectuées avant exécution, ainsi que du contrôle de cohérence.

1.3 Bases théoriques

Pour atteindre les buts fixés, *LET repose sur les Grammaires Affixes de Traductions proches des grammaires de Van Wijngaarden à 2 niveaux, (W-Grammaires) des grammaires attribuées et de Prolog

Les **W-Grammaires** apportent la puissance expressive requise : un même formalisme permet de décrire les règles de syntaxes, les restrictions contextuelles et/ou sémantiques et même la «sémantique dynamique» [WIJ 76], [CLE 77]. Si le traitement automatique de ces grammaires est possible, [SIM 81] il ne peut, dans le cas général, être que très lourd, ne serait-ce que par la difficulté de déterminer quelles sont les règles applicables à chaque instant de l'analyse. Les grammaires affixes sont également des grammaires à 2 niveaux qui possèdent la puissance descriptive [KOS 71] et la régularité des W-Grammaires. Mais, du fait de contraintes judicieuses, elles permettent d'obtenir efficacement l'analyseur le compilateur et l'interprète d'un langage à partir de sa définition formelle.

Les **Grammaires Attribuées** [DER 88] apportent l'efficacité de traitement car elles permettent d'appliquer les méthodes classiques d'analyse non contextuelle sur lesquelles sont greffés les calculs sémantiques dans lesquels on sait a priori quels attributs sont calculés à chaque étape de l'analyse. Mais cette greffe interdit que la sémantique guide l'analyse. Ainsi, les calculs complexes doivent obligatoirement être exprimés dans un autre formalisme ce qui empêche de prévoir des reprises automatiques. Avec les grammaires affixes, les calculs sémantiques sont étroitement liés à l'analyse et peuvent guider l'analyse ce qui permet d'exprimer les calculs complexes par des notions dites prédicatives.

Prolog apporte la puissance de traitement : la possibilité de traiter des grammaires ambiguës permet d'assurer que l'on trouvera toujours la bonne analyse d'un texte, sans que le programmeur ait à se préoccuper des reprises. Mais ceci se fait au prix d'une certaine lenteur, due aux nombreuses tentatives infructueuses d'unification.
La métagrammaire des grammaires affixes permet de prévoir a priori des échecs certains, ce qui par un tri statique, accélère l'exécution. De plus, sans limiter la puissance, les règles de cohérence grammaticale, vérifiables automatiquement, aident à la mise au point des traducteurs.

Ayant ainsi choisi la famille des grammaires affixes, nous voulons prendre en compte les différents langages qui interviennent dans la traduction source, objet mais également structures de données considérées comme langages intermédiaires. D'où l'idée des **Grammaires Affixes de Traduction** qui ne définissent plus un seul langage (source) mais plusieurs langages conjointement.

1.4 Caractéristiques générales

Un texte *LET peut-être considéré soit comme une grammaire à deux niveaux (Grammaire Affixe de Traduction) décrivant conjointement les langages en cause, soit comme un programme réalisant une traduction.

La grammaire est constituée d'abord de règles décrivant des notions. Ces règles pouvant être paramétrées par des séquences représentant des structures arborescentes ; en ce sens, elles peuvent être considérées comme des clauses définissant des prédicats. Ces règles s'appuient sur des notions terminales (prédéfinies ou non) qui décrivent les actions élémentaires sur les langages manipulés : reconnaissance dans les textes sources, génération dans les textes objets, création et exploitation des langages intermédiaires.

L'exécution d'un programme *LET revient à prouver le but principal : la racine de la grammaire, pour des valeurs particulières des textes sources. Cette preuve se fait par chaînage avant et fournit comme résultat les textes objets. L'activation de règles s'accompagne de confrontation des paramètres et arguments ce qui se traduit par la création ou l'éclatement de structures arborescentes.

Un deuxième groupe de règles définissent les affixes : ce sont les noms utilisés pour les variables et constantes des règles de notions. Algorithmiquement, ce sont des déclarations de types construits par union et ramification.

Pour simplifier certaines notations, très lourdes, des types sont prédéfinis ainsi que les opérations usuelles associées (entier, réel, chaîne de caractères).

Nos conventions typographiques sont les suivantes. Les exemples de programmes apparaissent dans une police de caractère différente :

```
Ceci est un morceau de programme *LET
```

Les règles de grammaires utilisées pour la définition du langage *LET sont mises en évidence par un trait vertical et une police de caractère spécifique (elles sont regroupées en Annexes p.53) :

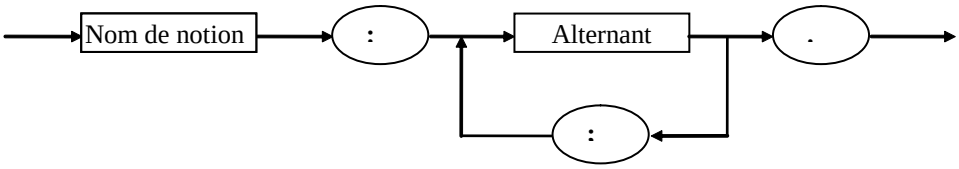
Ceci est une règle de grammaire : c'est le début , c'est déjà fini .

1.5 Formalisme de description

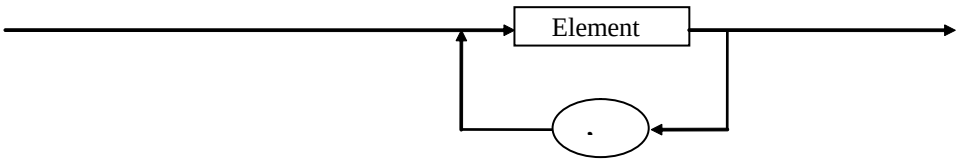
Nous allons décrire le langage *LET en utilisant son propre formalisme, d'abord de façon très simple (syntaxe non contextuelle uniquement) puis, au fur et à mesure de l'introduction des concepts, nous préciserons la description.

Si l'on se limite à la description non contextuelle d'un seul langage source, la notation est très proche de celle de Van Wijngaarden à 1 niveau.

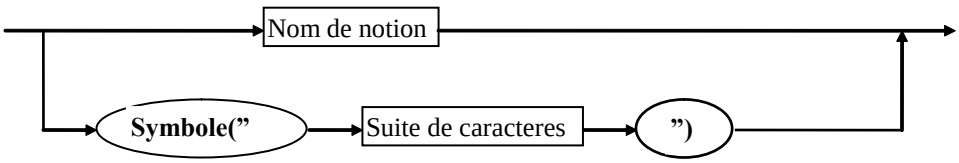
Règle :



Alternant :



Règle :



Ce qui s'exprime ainsi en *LET, en utilisant des récursivités pour les répétitions :

règle : *nom de notion, symbole (":"), suite d'alternants.*
suite d'alternants : *alternant, symbole (";"), suite d'alternants ;*
alternant, symbole (".").
alternant : *élément, symbole (","), alternant ; élément.*
élément : *nom de notion ;*
symbole("symbole(""), suite de caractères, symbole(""))".

Remarque : en pratique, dans cette dernière règle, il faudra mettre un anti-slash devant les guillemets placés entre guillemets.

1.6 Un premier exemple

Soit à traduire des expressions arithmétiques en notation polonaise postfixée. La traduction met en jeu deux langages source (notation infixée) et objet (notation postfixée) et marques d'erreurs éventuelle). Les vocabulaires de base de ces deux langages seront représentés respectivement par la notion symbole pour le source et la notion objet pour le code objet. La traduction est alors définie par la «grammaire» :

```
expression : somme, symbole(«.») ; objet («?») .
somme : produit, reste de somme .
reste de somme : symbole(«+»), produit, objet («+») ,
                        reste de somme ; VRAI.
produit : primaire, reste de produit .
reste de produit : symbole («*»), primaire, objet («*») ,
                        reste de produit ; VRAI.
primaire :      symbole («a») , objet («a») ;
                symbole («(«), somme, symbole («)») ;
                        objet (« ?»).
```

Algorithmiquement, «symbole» et «objet» sont des notions prédéfinies :

- «symbole» reconnaît une chaîne de caractère (éventuellement précédée d'espaces) dans le fichier source (stdin).
- «objet» écrit dans le fichier objet (stdout) une chaîne de caractère.

Ces 2 actions sont susceptibles d'être défaites si le choix d'un alternant se révèle erroné : cela permet de traiter des grammaires non LL(1).

Remarques :

- si on ôte les utilisations de «objet», on obtient une grammaire décrivant les expressions infixes. Mis à part les traitements d'erreurs, ce texte représente également un programme de reconnaissance de gauche à droite de ces expressions.
- si on ôte les utilisations de «symbole», on obtient une grammaire décrivant les expressions postfixées. Mais cette grammaire est récursive à gauche et ne décrit donc pas un programme de reconnaissance de gauche à droite de ces expressions.

2 Règles générales d'écriture

Un texte *LET se présente comme une grammaire à deux niveaux, plus précisément, une Grammaire Affixe de Traduction. Il est donc composé de deux ensembles de règles : les règles d'affixes et les règles de notions.

2.1 Règles d'affixes (métarègles)

Ces règles (non contextuelles) définissent des affixes dits variables en terme d'affixes variables ou constants.

2.1.1 Notation des règles d'affixes

règle d'affixe :	*affixe variable , symbole (« :»), développement d'affixe ;* *affixe variable, symbole («=«) , règle d'affixe.*
développement d'affixe :	*affixe prédéfini ;* *suite d'alternants d'affixe .*
suite d'alternants d'affixe :	*alternant d'affixe , symbole (« ;») , suite d'alternants d'affixe ;* *alternant d'affixe.*
alternant d'affixe :	*affixe , alternant d'affixe ; affixe.*
affixe :	*affixe variable ; affixe constant .*
affixe variable :	*suite de lettres.*
affixe constant :	*suite de lettres .*
affixe prédéfini :	*symbole («ENTIER») ;* *symbole («REEL) ;* *symbole («CARAC») ;* *symbole(«CHAINE»).*

Remarque :

- Les noms des affixes variables et constants sont tous composés de lettres minuscules ou majuscules. Ils sont distingués par le fait qu'un affixe variable doit être défini par une règle d'affixe alors qu'un affixe constant ne doit pas l'être (cf. 2.2).
- Le « :» sépare l'entête du corps de la règle tandis que les «=« séparent les noms d'affixes pour lesquels on veut avoir un même type (affixes synonymes).

Les écritures a = b : ....
a : b. b : ...
b : a. a : ... sont strictement équivalentes

2.1.2 Exemples de règles

a) **nom = texte : vide ; car texte.**

Cette règle définie l'affixe «texte» de manière récursive : c'est une suite quelconque de caractères (cf. d) se terminant par l'affixe constant vide. L'affixe nom est un synonyme de texte.

b) **absy : constante ; absy opérateur absy.**

On définit de manière récursive les absy représentant des expressions arithmétiques entières (arbre abstrait).

c) **opérateur : plus ; moins ; mult.**

On définit en extension l'ensemble des opérateurs arithmétiques.

d) **x= constante : ENTIER.**
car : CARAC.

Un x est du type prédéfini entier ; constante est un type synonyme de x (entier) et un car est du type prédéfini caractère.

Les affixes «texte», «expr», «opérateur» «constante» et «car» sont variables car définis. «vide», «plus», «moins» et «mult» sont constants. Enfin, «CARAC» et «ENTIER» sont des affixes variables prédéfinis.

2.1.3 Interprétation grammaticale

L'ensemble des règles d'affixes constitue la métagrammaire. Ce sont des règles non contextuelles associant à chaque affixe variable un langage dont les symboles sont les affixes constants. La possibilité de définition récursive de ces règles induit une infinité potentielle de phrases pour un langage d'affixe. Ces phases seront les modalités acceptables pour les règles de notions où un affixe variable apparaîtra. Une phrase d'affixes est une suite d'affixes constants.

2.1.4 Interprétation algorithmique

On peut également considérer que ce sont des définitions de types en ce sens qu'elles permettent d'associer à chaque affixe variable un ensemble de valeurs (phrases d'affixes structurées par leurs arbres syntaxiques) construit par union et concaténation. La définition de la métagrammaire apparaît alors comme une structuration logique de données.

2.2 Types et variables

2.2.1 Types et arbres

La définition de types au moyen d'une métagrammaire non contextuelle permet d'associer à chaque valeur de ce type (phrase d'affixes) un arbre d'affixes ou plusieurs si cette grammaire est ambiguë. En pratique, la phrase sera construite progressivement avec un arbre d'affixes qui dépendra de l'ordre de construction. C'est cet arbre qui représentera la phrase.

Lorsqu'une règle d'affixe comporte plusieurs alternants, on dit que les valeurs de ce type peuvent avoir différentes structures.

Remarque :
Lorsque les règles ont une structure simple, la phrase ne sera pas représentée par l'arbre complet (voir 3.3)

2.2.2 Variables

La description des notions (cf. 2.3) va utiliser des noms d'affixes variables, en position de paramètres formels ou d'arguments.

Grammaticalement, ces variables servent en position de paramètres formels, à déterminer les modalités acceptables pour une règle : c'est le langage associé à cet affixe ; en position d'arguments elles servent à transmettre un contexte à la règle utilisée, c'est-à-dire, à fixer des contraintes entre les diverses règles par le remplacement uniforme (cf. 2.3.3).

Algorithmiquement, ce sont des variables au sens de la programmation logique : ce sont des inconnues dont le programme cherche la valeur.

Les valeurs possibles pour une variable sont définies par la règle d'affixe correspondante. Ainsi, deux variables ayant le même nom ont toujours le même type. Si l'on désire utiliser plusieurs variables du même type dans une situation donnée, on peut suffixer les noms d'affixes par un entier pour désigner d'autres variables ou utiliser la possibilité de déclarer des affixes synonymes.

nom de variable : affixe variable, suffixe.
suffixe : entier décimal ; VRAI.

Remarque :
Dans certaines situations, il faudra faire précéder le nom de la variable par un blanc souligné pour indiquer que l'on n'a pas besoin de la calculer d'où la règle :

variable : nom de variable ;
symbole («_») , nom de variable.

Il est également possible d'utiliser des variables globales, connues dans tout ou partie d'un programme et qui ont un rôle plus algorithmique (cf. 2.5.4 et 5)

2.2.3 Types prédéfinis

Ces définitions de types sont donc bien adaptées à la spécification d'arbres et de listes, mais conviennent mal à la spécification d'ensembles dotés d'opérations plus complexes. C'est pourquoi, il existe 4 types prédéfinis : l'entier noté ENTIER, le nombre flottant noté REEL, le caractère noté CARAC et la chaîne de caractères notée CHAINE.

Ces noms de types ne sont pas des affixes banalisés car ils ne peuvent être utilisés que seuls dans des règles d'affixes et non pour former des noms de variables. De plus, les valeurs de ces types ne sont pas des arbres d'affixes, mais respectivement des entiers décimaux, des réels, des caractères et des chaînes de caractères (voir ci-dessous).

Les mêmes noms, avec seule l'initiale en majuscule, sont des affixes prédéfinis banalisés (noté Entier, Reel, le caractère noté Carac et la chaîne de caractères notée Chaine.

2.2.4 Constantes

Les valeurs élémentaires des types arborescents sont des arbres d'affixes dont les feuilles contiennent des affixes constants. Ces valeurs constantes sont représentées en mémoire au gré du compilateur *LET. Il est cependant possible de leur donner une représentation externe par l'intermédiaire d'une déclaration de constantes (cf. 2.5.3).

Les valeurs possibles pour les types prédéfinis sont respectivement les entiers décimaux, les caractères entre apostrophes, les suites de caractères entre guillemets.

D'où les règles :

constante : entier décimal ;
symbole («'») , caractère, symbole («'») ;
symbole («\»»), suite de caractères, symbole («\»»).
suite de caractères : caractère, suite de caractères ; VRAI.

Les caractères autorisés sont les caractères ASCII imprimables ou spéciaux (notés comme en C) :

\ n \r \t \' \» \entier octal.

2.2.5 Expressions d'affixes

Les expressions qui apparaissent en paramètres et arguments de notions sont composées de parties constantes et de parties variables.
Ces parties variables seront évaluées, au fur et à mesure de l'exécution, par des confrontations avec des valeurs connues.
Une expression d'affixes représente, dans le cas général, une structure d'arbre.

expression d'affixes : atome affixe, expression d'affixes ;
atome affixe.
atome affixe : variable ; affixe constant ; constante.

2.2.6 Exemples

Avec les règles d'affixes du paragraphe 2.1.2 :

texte1	est une variable de type texte
plus	est une constante de type opérateur
«abc»	est une constante de type CHAINE
'a' 'b' 'c' vide	est une expression d'affixes (phrase) de type texte
'a' texte1	est une expression d'affixes de type texte
constante0 moins absy1	est une expression d'affixes de type absy

2.3 En-tête des règles de notions (hyper règles)

Une règle de notion donne une définition d'une notion (non terminale, paramètre ou non) en membre gauche en termes d'autres notions (terminales ou non, avec ou sans arguments) concaténées dans un membre droit.

2.3.1 Notation

règle de notion :	en-tête de notion, symbole (« :»), développement de notion.
en-tête de notion :	marque, notion et paramètres formels, déclaration de variables locales.
marque :	marque de test ; marque d'action.
marque de test :	symbole (« ?»).
marque d'action :	symbole (« !») ; VRAI.
notion et paramètres formels :	nom de notion, paramètres formels, suite notion et paramètres formels ; nom de notion.
suite notion et paramètres formels :	nom de notion, paramètres formels, suite notion et paramètres formels ; nom de notion ; VRAI.
nom de notion :	suite de lettres chiffres tirets apostrophes et espaces.
paramètres formels :	symbole («(«), liste de paramètres, symbole («)»).
liste de paramètres :	paramètre, symbole («,»), liste de paramètres ; paramètre.
paramètre :	paramètre en entrée ; paramètre en sortie.
paramètre en entrée :	symbole («>«), expression d'affixes.
paramètre en sortie :	expression d'affixes, symbole («>«).
déclaration de variables locales :	symbole («/»), liste de variables ; VRAI.
liste de variables :	variable, symbole («,»), liste de variables ; variable.

Remarque :

- Une notion est repérée par un nom qui peut être éclaté en plusieurs morceaux placée devant et entre les paramètres formels ou les arguments. Tous ces morceaux de noms sont significatifs ainsi que leur emplacement parmi les arguments.
- Une même notion peut-être définie par plusieurs règles. Toutefois, si ces règles ont les mêmes paramètres formels, elles peuvent être regroupées en une règle à plusieurs alternants (cf. 2.4).

2.3.2 Exemples

Avec les règles d'affixes du paragraphe 2.1.2

somme (absy>)

indique une règle fabriquant un «absy» quelconque

génération (>absy opérateur absy1)

indique une règle traitant un «absy» composé de 2 «absy» encadrant un «opérateur»

opérateur (>plus)

indique une règle traitant la valeur «plus»

2.3.3 Interprétation grammaticale

Une hypernotion est formée d'un nom de notion et de paramètres formels (ou d'arguments).

L'en-tête d'une règle spécifie les conditions d'utilisation de cette règle. Les paramètres formels servent à définir les contraintes contextuelles qui devront être vérifiées lors de l'utilisation d'une notion en membre droit (cf. 2.4.).

Les variables qui apparaissent en membre gauche indiquent des contraintes de type : les valeurs possibles sont les phrases du langage de l'affixe de la variable. Chaque constante qui apparaît en tant que paramètre formel indique une contrainte de valeur : seule cette valeur est autorisée. De plus, lorsqu'une variable apparaît plusieurs fois en membre gauche, elle impose un couplage entre les valeurs des arguments correspondant.

Remarque :
Le sens des paramètres ainsi que les marques d'actions ou de tests ne jouent grammaticalement aucun rôle.

2.3.4 Interprétation algorithmique

Les règles sont soit des marqués par ?) soit des actions (marquées où non par !) suivant que leur développement peut échouer ou non.

Les paramètres sont dotés d'un sens : entrée ou sortie indiqué par «>« respectivement devant ou derrière.

Une constante en entrée indique une contrainte de valeur préalable à l'utilisation de la règle. C'est le cas du 3ème exemple du paragraphe 2.3.2 où la constante **plus** indique que la règle ne s'applique que pour cette valeur. Une constante en sortie fixe un résultat.

Une variable en entrée est destinée à recevoir une valeur à l'entrée de la règle. Elle fixe une contrainte de type car elle ne peut recevoir qu'une valeur du type de son affixe.

Une variable en sortie transmet un résultat calculé lors de l'essai du développement de la règle. Elle fixe également une contrainte de type pour la variable argument qui lui correspond. Ainsi, dans le 1er exemple du paragraphe 2.3.2, la notion **somme** est sensée calculer un résultat renvoyé dans la variable **absy**.

Une expression en entrée fixe une contrainte de structure sur la valeur reçue en plus des contraintes exprimées par chacun des affixes de cette expression. Le second exemple du paragraphe 2.3.2 indique que la valeur reçue par cette règle de **génération** n'accepte que des valeurs de la forme **absy opérateur absy1**.

Une expression en sortie fixe la structure du résultat transmis.

2.4 Développement de notion

2.4.1 Notation

développement de notion :	alternant de notion, symbole (« ;»), développement de notion ; alternant de notion, symbole («.») ; symbole («VRAI»), symbole («.»).
alternant de notion :	utilisation de notion, symbole («,»), alternant de notion ; utilisation de notion.
utilisation de notion :	nom de notion et arguments.
nom de notion et arguments :	nom de notion, arguments, suite de la notion et arguments ; nom de notion.
suite de la notion et arguments :	nom de notion, arguments, suite de la notion et arguments ; nom de notion ; VRAI.
arguments :	symbole («(«), liste d'arguments, symbole («)»).
liste d'arguments :	argument, symbole («,»), liste d'arguments ; argument.
argument :	expression d'affixes.

Remarque :
• Une même notion (même nom et même nombre de paramètres) peut être définie par plusieurs règles. Si ces règles ont exactement les mêmes paramètres, elles peuvent être regroupées en une seule règle à plusieurs alternants.
• Il n'existe pas de notation unique pour les symboles terminaux car le nombre des langages définis conjointement est quelconque. Les symboles de base des langages sont représentés par des notions dites terminales qui algorithmiquement sont prédéfinies ou écrites en C et représentent soit la reconnaissance d'un symbole dans un texte analysé, soit la génération d'un symbole dans un texte engendré.

Pour les langages standard source et objet, ces notions sont respectivement «**symbole**» et «**objet**».

2.4.2 Exemples

Présentons une solution au problème de l'analyse d'expressions arithmétiques avec fabrication d'un absy (arbre abstrait) puis exploitation de cet absy pour engendrer la notation postfixée.

```
AFFIXES :     absy :: x ; absy op absy
              op :: plus ; mult.
              x :: ENTIER.
NOTIONS :
1)    expression / absy :
      somme (absy), symbole («.»),
            génération (absy) ;
      objet («erreur»).
2)    somme (absy> ) / absy1 :
      produit (absy1),  reste de somme (absy1, absy).
3)    ? reste de somme ( >absy1, absy >) / absy2 :
      symbole («+») , produit (absy2),
            reste de somme (absy1 plus absy2, absy).
4)    reste de somme ( >absy, absy>) : VRAI.
5)    produit (absy> ) / absy1 :
      primaire (absy1), reste de produit (absy1, absy).
6)    ? reste de produit ( >absy1, absy> ) / absy2 :
      symbole («*»), primaire (absy2),
            reste de produit (absy1 mult absy2, absy).
7)    reste de produit ( >absy, absy> ) : VRAI.
8)    ? primaire (x> ) : entier (x).
9)    ?primaire (absy>) : symbole («(«), somme (absy), symbole («)»).
10)   primaire (0> ) : objet («erreur»).
11)   génération ( >absy op absy1) :
      génération (absy), génération (op), génération (absy1).
12)   génération ( >x ) : objet (x).
13)   génération (> plus) : objet («+»).
14)   génération (> mult) : objet («*»).
```

Les règles semblables comme «somme» et «produit» peuvent être regroupées en une seule règle paramétrée. Ceci permettrait de généraliser aisément les expressions à un grand nombre d'opérateurs de priorités différentes.

```
1)    expression / absy :
      expr (plus, absy), symbole («.»),
      génération (absy) ; objet («erreur»).
2)    expr ( >op, absy> ) :   opérande (op, absy1),
                              reste de (op, absy1, absy).
3)    ?reste de ( >op, >absy1, absy>) / absy2 :
      opération (op), opérande (op, absy2),
            reste de (op, absy1 op absy2, absy).
4)    reste de ( >op , >absy, absy>) : VRAI
5)    ? opération ( >plus) : symbole («+»).
6)    ? opération ( >mult ) : symbole («*»).
7)    opérande ( >plus, absy> ) : expr (mult, absy).
8)    ? opérande ( >mult, x> ) : entier (x).
9)    ? opérande ( >mult, absy> ) :
      symbole («(«), expr (plus, absy), symbole («)»).
10)   opérande ( >mult , 0>) : écrire («erreur»).
```

2.4.3 Interprétation grammaticale

L'ensemble des règles de notions constitue l'hypergrammaire. Ce sont des règles contextuelles, modèles de règles non contextuelles obtenues par remplacement uniforme comme pour les grammaires de Van Wijngaarden à deux niveaux : on peut les utiliser comme des règles de réécriture après avoir remplacé chaque variable qui y figure par une phrase d'affixe appartenant au langage de l'affixe correspondant ; toutes les occurrences de la même variable étant remplacée par la même phrase.
Ce faisant, on peut obtenir dans le développement une hypernotion qui ne peut-être réécrite car elle ne peut être construite par remplacement uniforme d'un en-tête de règle : c'est une **impasse**.

Ainsi paramétrées, les règles de notions sont donc à la fois générales par le nombre de règles non contextuelles qu'elles remplacent et sélectives par le couplage imposé par le remplacement uniforme qui rend certaines règles inapplicables.

Une notion particulière est la racine de la grammaire. Par réécritures successives de cette notion, on obtient finalement une suite de notions terminales correspondant aux symboles terminaux des grammaires classiques. En séparant dans cette suite les notions s'appliquant aux langages source et objet, on obtient une phrase de chacun de ces langages. L'ensemble des phrases sources forme le langage source (respectivement objet). La traduction est définie par l'ensemble des couples phrase source- phase objet que l'on peut obtenir par ce procédé.

Remarque :
Nous étendrons par la suite cette définition à un nombre quelconque de langages définis conjointement.

2.4.4 Interprétation algorithmique

(Détails au paragraphe 4)

a) Une règle de notion définit une méthode d'essai d'une notion : il s'agit d'essayer dans l'ordre les notions qui apparaissent comme éléments du membre droit avec les contraintes contextuelles transmises par les arguments.
Par la suite, la notion utilisée en membre droit est dite notion appelée alors que la notion en membre gauche est dite notion appelante.
Les arguments comportent des variables et des constantes qui jouent un rôle symétrique de celles qui apparaissent en paramètres.
Un argument en entrée transmet une valeur à la règle appelée : valeur fixe pour une constante, calculée préalablement pour une variable. Une constante en sortie fixe une contrainte de valeur pour le résultat de la règle appelée. Une variable en sortie est destinée à recevoir un résultat et fixe donc une contrainte de type sur ce résultat.
Une expression en entrée fixe la structure de la valeur transmise. Une expression en sortie indique une contrainte sur la structure du résultat attendu.

b) L'essai d'un élément consiste d'abord en la sélection d'une règle applicable par confrontation des arguments avec les paramètres formels de la règle.
La confrontation est une unification typée et dirigée par le sens des paramètres. En entrée, les valeurs des arguments sont transmises aux paramètres formels ce qui s'accompagne d'un test de valeur pour les paramètres constants ou d'un test de type suivi d'une instanciation pour les paramètres variables.
En sortie, les valeurs des paramètres sont transmises de la même manière aux arguments : c'est la confrontation en sortie. Cependant, afin d'éviter des calculs inutiles, les contraintes de valeurs et de types peuvent être transmises à la règle appelante pour être prises en compte au plus tôt.
Lorsqu'une variable apparaît plusieurs fois dans les paramètres, les valeurs des arguments correspondant doivent être identiques (remplacement uniforme). La vérification de cette contrainte s'accompagne soit de tests d'égalité soit d'instanciations de variables inconnues.

c) Lorsqu'une règle appelée est sélectionnée, son développement est essayé de la même manière en utilisant les valeurs transmises par la confrontation en entrée. Si cet essai réussi, la confrontation en sortie fournira des résultats utilisables pour l'essai des éléments suivants de la règle appelante.
Un élément de notion est donc accepté si la confrontation a réussi avec les paramètres d'une règle et si l'essai de cette règle a réussi.

d) Finalement le membre droit d'une règle est accepté lorsque tous les éléments sont acceptés avec les mêmes valeurs des variables. Si une règle échoue, on essaie les autres alternants de la même règle ou bien on cherche à sélectionner une autre règle pour le même élément (cf. 6).
Remarque :

- Une règle est dite test si son essai peut échouer, elle est dite action dans les cas contraire.
- Le fait d'accepter une règle pour un élément n'est pas définitif : si les résultats fournis par cette règle ne permettent pas de terminer positivement l'essai du développement, on essaiera les autres règles possibles pour cet élément (indéterminisme local).

e) L'exécution du texte consiste en l'essai de la règle racine ; cet essai est décomposé récursivement par le procédé ci-dessus et fournira un texte objet dépendant du texte source.

2.5 Autres règles d'écriture

2.5.1 Structure générale d'un programme

Un programme *LET simple commence par la définition de la racine et se compose de paragraphes définissant les affixes et les notions. Ces paragraphes peuvent être alternés à volonté. Les règles définissant une même notion doivent être consécutives. Un programme plus important peut être divisé en modules (cf. 6).

programme :	symbole («RACINE :») , développement de notion, paragraphes.
paragraphes :	paragraphe , paragraphes ; VRAI.
paragraphe :	symbole («AFFIXES :»), règles d'affixes ; symbole («NOTIONS :»), règles de notions ; déclaration de variables ; définitions de constantes ; importation de modules.
règles d'affixes :	règle d'affixe, règles d'affixes ; règle d'affixe.
règles de notions :	règle de notion, règles de notions ; règle de notion.

2.5.2 Définitions de constantes

On peut donner des noms à des valeurs constantes dans un paragraphe :

définitions de constantes	symbole («CONSTANTES :»), liste de définitions de constantes.
liste de définitions de constantes :	définition de constante, symbole («,») , liste de définitions de constantes ; définition de constante.
définition de constante :	affixe constant , symbole («=«), constante .

Les valeurs attribuées à ces noms seront utilisées pendant les calculs de variables des types prédéfinis. Ces définitions ne servent alors qu'à rendre les programmes plus lisibles.

Ces définitions serviront également à donner la représentation externe des constantes utilisées dans les types construits (arborescent ou énumérés), la représentation interne de ces constantes étant indifférente est laissée au soin du compilateur *LET (cf. 3.5 et 7).

2.5.3 Variables globales

Des variables accessibles dans toutes les notions peuvent être déclarées de la manière suivante (cf. 5) :

déclaration de variables : symbole («VARIABLES :»), liste de variables , symbole («.»).

2.5.4 Mise en page et lexicographie

La mise en page du texte *LET est libre c'est-à-dire que espaces, tabulations, sauts de lignes et commentaires peuvent être insérés devant ou derrière un caractère spécial ou entre deux noms. Notons que les commentaires ne sont pas autorisés entre deux morceaux de nom de notion. Un commentaire est encadré par des signes «».
Les mots-clés (i.e. qui ont une signification particulière en StarLet) sont réservés c'est-à-dire qu'ils ne peuvent pas être utilisés comme noms d'affixe ou de notion. Ils sont toujours écrits en majuscule. Ce sont les mots:

AFFIXES	MODULE
AVEC	NOTIONS
CARAC	RACINE
CHAINE	REEL
CONSTANTES	VARIABLES
ENTIER	VRAI

Les variables inutiles sont précédées d'un blanc souligné (cf. 5.4).

Les éléments de base du langage sont définis par les règles suivantes :

chiffre / _Carac : chiffre(_Carac).
lettre / _Carac : lettre(_Carac).
caractère / _Carac : caractère quelconque(_Carac).
entier décimal : chiffre , entier décimal ; chiffre .
suite de lettres : lettre , suite de lettres ; lettre.
suite de lettres et de chiffres : lettre, autres lettres et chiffres.
autres lettres et chiffres : lettre, autres lettres et chiffres ;
chiffre, autres lettres et chiffres ;
VRAI.
suite de lettres chiffres tirets apostrophes et espaces : lettre, autres lctae.
autres lctae : lettre, autres lctae ; chiffre, autres lctae ;
symbole(«-») , autres lctae ; symbole(«_»), autres lctae ;
symbole(«'») , autres lctae ; symbole(« «), autres lctae ;
VRAI.

2.5.5 Exemple auto-référentiel

Voici la grammaire d'analyse de *LET en *LET.

Remarquer les notions définissant les listes avec ou sans séparateurs : elles permettent de simplifier l'écriture d'autres définitions (à comparer avec les règles regroupées dans l'annexe A).

L'analyseur lexicographique s'appuie sur des fonctions prédéfinies qui gèrent un texte source avec retour en arrière si nécessaire : «symbole», «c'est un», «lettre», «chiffre», «caractère quelconque», «blancs», ou encore «ce n'est pas un». De même, la liste en écho est faite grâce à la notion prédéfinie «objet».

```
RACINE :
        programme , fin de texte , objet(«bon») ;
        objet(«faux»)
AFFIXES :
L       ::      LL de AFFOUNOT ;
                paragraphe ; varouconst ;
                ARGOUPARF ;
                variable ; affixe ;
                déclaration de VARCONST.
LL      ::      alternant ;
                élément ;
                règle .
AFFOUNOT :: affixe ; notion .
NOTION :: action ; test .
ARGOUPARF :: argument ; paramètre .
VARCONST :: constante ; variable .
SEP :: CHAINE .
CAR :: CARAC .

NOTIONS : $ architecture générale $ ?liste de (>L) sépare par (>SEP) :
        un (L) , symb(SEP) , liste de (L) sépare par(SEP) ;
        un(L) . ?liste de (>L) :
        un (L) , liste de (L) ;
        un(L) . ?programme : symb(«RACINE») , symb(« :») ,
        développement de (notion) , symb(«.») ,
                liste de (paragraphe). ?un (>paragraphe) /AFFOUNOT :
        titre (AFFOUNOT) , liste de (règle de AFFOUNOT) ;
        titre (VARCONST),
        liste de (déclarations de VARCONST) séparé par («,»),
                symb(«.»). ?titre (affixe>) : symb («AFFIXES») , symb(« :») . ?titre
(notion>) : symb («NOTIONS») , symb(« :») . ?titre (variable>) : symb («VARIABLES») ,
symb(« :») . ?titre (constante>) : symb («CONSTANTES») , symb(« :») .
```

NOTIONS : $ écriture des règles : entêtes $?un (>règle de AFFOUNOT) :
entête de (AFFOUNOT) , symb(« :») ,
développement de (AFFOUNOT) , symb(«.») . ?un (>règle de notion) :
entête de (notion) , symb(«.») . ?entête de (>affixe) : liste de (affixe) sépare par («=«) . ?entête de (>notion) /_NOTION :
marque de (_NOTION) , nom de notion et (paramètre) ,
variables locales . ?marque de (test>) : symb (« ?») .
marque de (action>) : symb (« !») ; VRAI .
variables locales : symb («/») , liste de (variable) sépare par («,») ; VRAI .

NOTIONS : $ développements $?développement de (>affixe) :
symb («ENTIER») ; symb («CHAINE») ; symb («CARAC») . ?développement de (>AFFOUNOT) :
liste de (alternant de AFFOUNOT) sépare par(« ;») .

?un (>alternant de affixe) : liste de (élément de affixe) . ?un (>alternant de notion) : liste de (élément de notion) sépare par («,») . ?un (>alternant de notion) : symb («VRAI») . ?un (>élément de affixe) : nom d'affixe . ?un (>élément de notion) : nom de notion et (argument) .

NOTIONS : $ nom de notion paramètre $?nom de notion et (>ARGOUPARF) :
nom de notion , groupe de (ARGOUPARF) ,
nom de notion et(ARGOUPARF) ;
nom de notion , groupe de (ARGOUPARF) ;
nom de notion .

?groupe de (>ARGOUPARF) :
symb(«(«) , liste de (ARGOUPARF) sépare par («,»), symb(«)») . ?un(>argument) : liste de (varouconst) . ?un(>paramètre) : symb(«>«) , liste de (varouconst) ;
liste de (varouconst) , symb(«>«) . ?un (>varouconst) : variable ; nom d'affixe ; valeur constante . ?un (>affixe) : nom d'affixe . ?un (>variable) : variable . ?un (>déclaration de variable) : variable . ?un (>déclaration de constante) :
nom d'affixe, symb(«=«), valeur constante.

3 La confrontation

3.1 Généralités

L'appel d'une règle s'accompagne de «calculs» des valeurs à affecter aux variables paramètres en entrée ; ceci est fait par **confrontation** des paramètres formels avec les arguments. Cette confrontation peut échouer (impasse due à un test de confrontation) lorsque la valeur actuelle des arguments n'est pas applicable aux paramètres formels.

De même il y a une confrontation des paramètres en sortie avec les arguments correspondants pour affecter, si cela est possible, des valeurs à des variables qui apparaissent dans ces arguments.

3.1.1 Différentes sortes de confrontations

Nous distinguerons plusieurs sortes de confrontations, suivant la structure des règles d'affixes utilisées :

- confrontation d'arbres : construction et exploration d'arbres d'affixes (cf. 2.2).
- confrontation de types prédéfinis : affectations et tests de valeurs entières ou réelles, de caractères ou de chaînes.
- confrontation d'énumérations, lorsque les valeurs possibles sont toutes des phrases d'affixes élémentaires, c'est-à-dire composées d'un seul affixe terminal.

3.1.2 Tests statiques

Les tests de confrontation ne sont pas totalement effectués à l'exécution. Un tri des règles acceptables est effectué préalablement à l'interprétation ou à la compilation. On évite ainsi de tenter des confrontations qui ne peuvent jamais réussir.
De plus, cela permet de constater des cas d'erreurs de programmation lorsque pour un appel aucune règle ne peut réussir (**impasse systématique**), ou lorsqu'une succession d'appels ne peut réussir par suite du remplacement uniforme.

Ce contrôle a priori est semblable au contrôle des types des arguments dans un langage autorisant la synonymie des fonctions, tel ADA, à ceci près qu'il peut y avoir plusieurs règles sélectionnées pour un même appel.

3.1.3 Remplacement uniforme des paramètres

Lorsqu'une variable apparaît plusieurs fois dans les paramètres d'une notion, les valeurs des arguments correspondants doivent être identiques. Cela implique des affectations ou des tests de valeur suivant le sens des paramètres.

Ainsi, dans l'exemple du paragraphe 2.4.2, la règle 4 :

reste de somme (>absy, absy>) : VRAI.

lorsqu'elle s'applique, renvoie comme résultat (2° paramètre), la valeur reçue comme 1° paramètre.

Une autre application courante est le test de l'égalité de deux valeurs. La règle suivante s'applique si les deux arguments de type **absy** sont égaux.

égal (>absy,>absy) : VRAI.

De même si une variable apparaît plusieurs fois en argument (d'une notion ou de plusieurs notions appelées successivement) en position de résultat, cela implique que les différentes valeurs obtenues en résultat doivent être identiques.

Avec la règle 2 du paragraphe 2.4.2 (entête **somme(absy>)**), le développement suivant échoue si les deux appels ne renvoient pas la même valeur c'est-à-dire si l'on ne trouve pas deux fois de suite la même expression arithmétique.

E / absy1 : somme (absy1), somme (absy1)

3.2 Confrontation d'arbres

3.2.1 Principe

Chaque valeur d'affixe (sauf cas particuliers : Cf. 3.3, 3.4) est représentée par l'arbre syntaxique qui reflète la manière dont cette valeur (phrase d'affixe) a été construite.
La confrontation des arguments avec les paramètres de types arborescents, permet la **construction** de ces valeurs lorsqu'une expression d'affixes de valeurs connues (variables instanciées ou constantes) est confrontée avec une variable à instancier.

A l'inverse, lorsqu'une variable instanciée est confrontée avec une phrase d'affixe, il se produit un **éclatement** de la valeur pour déterminer les valeurs à affecter aux variables qui apparaissent dans la phrase.

La construction d'arbre a donc lieu lorsqu'une expression d'affixe est présentée en arguments face à un paramètre variable en entrée (cons1), ou lorsqu'une expression d'affixe est en position de paramètre en sortie (cons2).
L'éclatement a lieu lorsqu'une expression d'affixe est en position de paramètre en entrée (eclat1) ou lorsqu'une expression d'affixe est en argument face à un paramètre en sortie (eclat2).

	Paramètre en entrée	**Paramètre en sortie**
Paramètre expression		
Argument variable	Eclatement	Construction
Paramètre variable		
Argument expression	Construction	Eclatement

(non encore réalisé)

Illustrons les différents cas de figure sur des extraits de l'exemple donné au paragraphe 2.4.2 augmenté de quelques définitions complémentaires :

```
AFFIXES :
affec :: nom absy.

NOTIONS :
$       Cons1 : construction d'une valeur de type absy transmise lors de l'appel
        récursif de reste de somme          $ ? reste de somme ( >absy1, absy >) / absy2 :
        symbole («+») , produit (absy2),
                reste de somme (absy1 plus absy2, absy).
$       Cons2 : construction d'un résultat de type affect à partir d'un nom et
        d'un absy                           $
instruction (nom absy>) :
        variable (nom), symbole (« :=«), somme (absy).
$       Eclat1 : éclatement d'une valeur de type absy en ses éléments absy, op, absy1
        lors des appels récursifs de génération     $
génération ( >absy op absy1) :
        génération (absy), génération (op), génération (absy1).
$       Eclat2 : l'appel de somme ci-dessous n'accepte qu'un résultat comportant
        une addition.                       $
E / absy1,absy2 : somme (absy1 plus absy2), génération (absy1), ...
```

Ce dernier mécanisme n'étant pas encore implanté, il faut le remplacer par :

```
E / absy1,absy2 : somme (absy3), traiter (absy3), ....
traiter (>absy1 plus absy2) : génération (absy1), ...
```

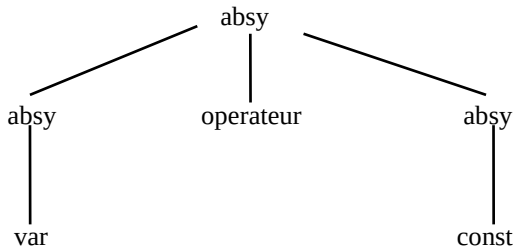
L'éclatement est alors réalisé à l'entrée de la règle **traiter**.
Notons que l'expression d'affixe peut se limiter à une seule variable, mais qu'il y aura quand même confrontation si le type de cette variable est un sous-type du type de l'autre variable.

Exemple :

```
absy : var ; cste ; absy opérateur absy.
var : nom type .
constante : ENTIER.
Nom = type : CHAINE.
```

Si **absy** est confronté à **var**, il y a construction ou éclatement d'un absy absy–>var. Le travail effectué lors de la confrontation peut se faire sur un nombre quelconque de niveaux de l'arbre. La structuration obtenue (qui n'a pas à être indiquée par un parenthésage) est calculée d'après les règles d'affixes.

Exemple :
La construction de **absy** suivant l'expression **var opérateur constante** donne lieu à la création de l'arbre :



3.2.2 Exemples

Lecture d'une liste d'entiers, ***inversion*** de cette liste en mémoire et impression du résultat.

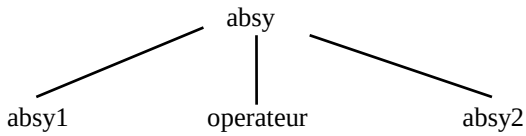
```
 RACINE : essai d'inversion, a la ligne.
AFFIXES :       l :: vide ; x l.
            x :: ENTIER.
            car :: CARAC.
NOTIONS :
 essai d'inversion / l,l1 :
  lirel(l) , l'inverse de (l) devant (vide) est (l1) , écrirel(l1).
l'inverse de( > x l) devant (> l1) est (l2 > ) :
  l'inverse de (l) devant (x l1) est (l2 ).
l'inverse de(>vide)devant(>l9)est(l9>) : VRAI.
?lirel(x l>) : lire(x), écrire(x), écrire(' '), lire(l).
lire(vide>) : VRAI.
écrirel(>x l) : écrire(x) , écrire(' ') , écrirel(l).
écrirel(>vide) :VRAI.
écrire(>x).       écrire(>car).          ?lire(x>).       a la ligne.
```

Voici un exemple de ***Tri arborescent binaire***

```
RACINE : essai de tri, a la ligne.
AFFIXES :       x :: ENTIER.
            l :: vide ; l x l.
            car :: CARAC.
NOTIONS :
essai de tri / l : l'adjonction d'entiers dans(vide)donne(l) , écrirel(l). ?l'adjonction d'entiers
dans (> l ) donne ( l1 > ) / x,l2 :
            lire(x) , l'insertion de (x) dans (l) donne(l2) ,
                    l'adjonction d'entiers dans (l2) donne (l1).
l'adjonction d'entiers dans (> l) donne (l>) : VRAI.
l'insertion de(>x) dans (>vide) donne (vide x vide>) : VRAI. ?l'insertion de (> x) dans (>l1 x1
l2) donne(l3 x1 l2>) :
            inf(x,x1) , l'insertion de(x)dans(l1)donne(l3).
l'insertion de(>x) dans (>l1 x1 l2) donne (l1 x1 l3>) :
            l'insertion de(x)dans(l2)donne(l3).
écrirel(> l1 x l) : écrirel(l1) , écrire(x) , écrire(' ') , écrirel(l).
écrire(> vide) : VRAI.
écrire( > x).  écrire( :car).  a la ligne. ?inf( > x, > x1).    ?lire(x>).
```

3.2.3 Tests statiques

La confrontation d'une variable avec une expression d'affixe ne peut réussir que si cette expression est **compatible** avec une des structures du type de la variable.
Une expression d'affixes e est **compatible** avec une structure d'un affixe a s'il existe des règles d'affixes telles que e dérive de a. Cette condition est vérifiée une fois pour toute lors de l'analyse du texte, par une analyse syntaxique suivant la grammaire des affixes.
Ainsi, avec les règles d'affixes déjà données, l'expression **absy1 opérateur absy2** est compatible avec la variable **absy3** et l'arbre construit sera :



Une valeur d'affixe *a* est **compatible** avec une expression d'affixe *e* s'il existe des règles d'affixes telles que *e* puisse être produit à partir de *a* et si la valeur actuelle de *a* a été construite selon ces règles.
Ainsi, une valeur de la variable **absy3** est compatible avec **absy1 opérateur absy2** pourvu que **absy3** ne soit pas une feuille de type constante.
Ce tri préalable à l'exécution permet de donner le même nom à des règles qui traitent des objets de type différents, sans craindre de ralentir le programme par des essais de confrontation inutiles.

Exemple :

```
AFFIXES :      le :: vide ; e le.
               e :: ENTIER
               ln :: vide ; n ln.
               n :: CHAINE
NOTIONS :
écrire (> e le) : ...
écrire (> n ln) : ...
A : ... écrire (le) , écrire(ln) ...
```

Dans la règle **A**, le premier appel ne donnera lieu qu'à l'essai de la première règle **écrire**, le second appel qu'à celui de la seconde règle.
Par contre, cette homonymie des règles permet de n'écrire qu'une fois des traitements à effectuer sur des variables de types différents mais qui possèdent une structure en commun.

```
écrire (> vide) : ...
```

Cette troisième règle sera essayée pour les deux appels de **écrire**.

3.2.4 Tests dynamiques

Les tests statiques assurent que les constructions d'arbres pourront toujours réussir. Par contre lors d'un éclatement il reste à vérifier que la valeur actuelle de la variable a bien la structure voulue.
Dans l'exemple précédent, l'appel **écrire (le)** s'accompagne d'un test pour déterminer si la variable **le** est de la forme **e le** (on exécute alors la première règle) ou de la forme **vide** (on exécute alors la troisième règle).

3.2.5 Ambiguïté de la grammaire des affixes

La grammaire des affixes est ambiguë si une phrase du langage d'un affixe peut être construite de deux manières différentes par les règles qui définissent cet affixe. Le vérificateur (version 1) ne cherchant qu'une possibilité de construction ou d'éclatement, il peut alors se faire qu'on n'arrive pas à éclater un arbre car on essaie la mauvaise construction.

Exemple **:**

```
A : B ; C .
B : C  y .
C : x y ; x .
```

Les règles d'affixes étant utilisées dans l'ordre où elles sont données, la construction x y -> A

donne

```
      A
      B
   C     y
X
```

et non pas

```
      A
      C
   x     y
```

L'éclatement en deux temps : A -> C puis C->x y ne réussit pas, alors qu'à la lecture d'un programme, on peut s'attendre à le voir réussir.

3.3 Traitements des énumérations

3.3.1 Principe

Nous venons de voir, que dans le cas général, une phrase d'affixe est représentée par l'arbre syntaxique correspondant à la manière dont elle a été construite. Ce procédé permet de conserver toute l'information induite par les traitements, mais s'avère inutilement coûteuse pour un affixe variable dont toutes les phrases se réduisent à un seul affixe constant. Dans ce cas, le type de l'affixe est dit **énuméré** et les valeurs des variables de ce type sont représentées par des entiers choisis par le compilateur.
Le passage d'une valeur en paramètre s'accompagne alors d'un test pour savoir si cette valeur appartient à l'ensemble des valeurs permises pour ce type. Ce traitement particulier permet l'intersection implicite des types d'affixe.
Exemple **:**

```
x : A ; B ; C .
y : C ; D .
```

Les types **x** et **y** possèdent la valeur **C** en commun

3.3.2 Exemple

Lecture d'un jour de la semaine, et écriture de «oui» si c'est un jour familial (ni les parents ni les enfants ne travaillent).
Pour adapter ce programme à des coutumes différentes, il suffit de modifier les règles d'affixes. L'appel de familial fait un premier tri entre les jours chômés et ouvrés. Dans la première règle de familial l'appel de classe permet de trier les jours de classe. Finalement le programme écrit 'oui' si le jour est chômé et n'est pas un jour de classe (en fait le Dimanche).

```
RACINE : essai, a la ligne.
AFFIXES :       mot :: CHAINE.
          ouvre :: lundi ; mardi ; mercredi ; jeudi ; vendredi .
          chôme :: samedi ; dimanche .
          jour :: ouvre ; chôme .
          classe :: lundi ; mardi ; jeudi ; vendredi ; samedi .
NOTIONS :
essai/jour :      lire jour(jour), familial (jour) ;
                  écrire(«quel calendrier utilisez-vous ?»). ?lire jour (jour>)/mot : lire(mot) ,
traduire(mot)en(jour).
traduire(>«lundi») en (lundi>) : VRAI.
traduire(>«mardi») en (mardi>) : VRAI.
traduire(>«mercredi») en (mercredi>) : VRAI.
traduire(>«jeudi») en (jeudi>) : VRAI.
traduire(>«vendredi»)en (vendredi>) : VRAI.
traduire(>«samedi») en (samedi>) : VRAI.
traduire(>«dimanche») en (dimanche>) : VRAI.
familial(>chôme) :      classe(chôme), écrire chaîne(«non») ;
                  écrire chaîne(«oui»).
familial(>_jour) :  écrire(«non»).
classe(>_classe) : VRAI . ?lire(mot>).  écrire( :mot).  a la ligne.
```

3.3.3 Tests statiques

Les confrontations ne sont possibles a priori qu'entre :

- une variable et une constante appartenant à l'ensemble des valeurs de cette variable ;
- deux variables dont les types ont une intersection non vide ;
- deux constantes identiques.

3.3.4 Tests dynamiques

A l'exécution, les trois cas précédents de confrontation se réduisent selon le sens des paramètre aux cas suivants (c : constante ; x, y : variable) :

- c –> x affectation de valeur
- x –> c test : valeur actuelle de x = c
- x –> y test : valeur actuelle de x appartient au type de y puis affectation de la valeur
  Dans ce cas le test est inutile si les types x et y sont identiques.
- c –> c rien à faire

3.4 Traitement des types prédéfinis

3.4.1 Principe

Les affixes d'un type prédéfini (entier, réel, caractère, chaîne) ne peuvent recevoir que des valeurs de ce type. Une telle variable ne peut donc être confrontée qu'avec une variable du même type ou une constante de ce type (respectivement suite de chiffre, caractère entre apostrophes, chaîne entre guillemets, suivant les conventions du langage C).

3.4.2 Tests statiques

Le choix des règles utilisables est donc fait en vérifiant qu'une variable est confrontée à une variable ou une constante de même type, et qu'une constante est confrontée à une constante identique ou à une variable de son type.

3.4.3 Tests dynamiques

Suivant le sens des paramètres, l'exécution se réduit à :

c –> x	affectation de valeur
x –> y	affectation de valeur (sans test car les types sont identiques)
x –> c	test : valeur actuelle de x = c
c –> c	rien à faire

3.5 Traitement particulier des chaînes

Non encore implanté (cf. provisoirement paragraphe précédent)

3.6 Remplacement uniforme

3.6.1 Principe

Les variables (paramètres ou locales) utilisées dans un programme *LET sont des inconnues dont il s'agit de trouver une valeur satisfaisante pour la règle lorsqu'elle est appelée. Cela implique que toutes les occurrences d'une variable dans une règle (aussi bien dans les paramètres formels que dans les arguments des notions appelées) représentent une même variable avec une même valeur.
Notons que lorsqu'une règle est appelée plusieurs fois en cours d'exécution, la valeur d'une variable donnée peut être différente d'un appel à l'autre. Ceci permet entre autre des appels récursifs se terminant.
Pratiquement cela entraîne soit des affectations de valeurs déjà connues à des variables non encore instanciées soit des tests d'égalité de valeurs préalablement calculées.

3.6.2 Exemples

Des applications simples du principe de remplacement uniforme sont :

```
AFFIXES :      L :: e, L ; vide .
               e : ENTIER.
```

```
1)      si ( > L) égal ( > L ) : VRAI .
```

Test de l'égalité de deux expressions de type **L** ;
Lors d'un appel **si (L) égal (L1)**, les valeurs des deux arguments seront comparées par une procédure adaptée aux règles définissant **L**.

```
2)      affecter ( > L ) a ( L > ) : VRAI .
```

Donner la valeur du premier argument au deuxième ;

Utilisations possibles : affecter (vide) a (L)
affecter (1 L) a (L1)

```
3)      Le premier ( > x _L ) est ( x > ) : VRAI .
```

Extraction du premier élément d'une liste non vide ;

```
4)      si le premier de ( > x _L ) est ( > x ) : VRAI .
```

Test du premier élément d'une liste non vide.

3.6.3 Tests et affectations dynamiques

Les situations particulières à noter sont :

- Variable apparaissant deux fois dans les paramètres formels.
    - Une fois en entrée, une fois en sortie : affectation de valeur
    - Deux fois en entrée : test d'égalité des arguments correspondant
    - Deux fois en sortie : deux résultats identiques.
- Variable déjà instanciée (paramètre en entrée, ou variable ayant déjà reçue une valeur) en argument en position de résultat : la valeur reçue doit être identique à la valeur déjà connue.

3.6.4 Tests statiques des enchaînements

La règle du remplacement uniforme permet d'éliminer statiquement des enchaînements de règles qui ne peuvent jamais réussir. En effet, un simple contrôle local permet de repérer lorsqu'une règle donne (ou reçoit) une certaine valeur ou une certaine structure à une variable alors qu'une règle appelée après attend une autre valeur ou structure. On dit alors que l'appel est impossible à cause de son contexte.

Les enchaînements correspondants sont éliminés, ce qui accélère l'exécution. Si toutes les possibilités sont éliminées, une erreur est signalée.

Exemples : avec **L :: x L ; vide .**

lire (x L >) : ....
lire (vide >) : ....
écrire (> x L) : ....
écrire (> vide) : ....
exemple / l > lire (L) , écrire (L) .

Dans l'essai, si la première règle de **lire** réussit, seule la première règle de **écrire** sera essayée. Il en sera de même pour la seconde. Nous présentons les problèmes liés à la cohérence des déclarations des tests et actions ainsi que le contrôle des flots de données et de l'ordre d'enchaînement des essais.

4 Enchaînement des essais : tests, actions et indéterminisme local.

4.1 Introduction

L'interprétation algorithmique d'un texte *LET est destiné à l'analyse de langages appartenant à une classe plus large que celle traitée par les analyseurs descendants classiques. Les reprises locales permettent d'analyser des langages LL(k) sans avoir à se préoccuper des retours sur les variables du programme ni même sur les textes source et objet (grâce à l'analyseur lexicographique de bas niveau prédéfini, Cf. 6.4 et 7).

De plus, l'inclusion de règles prédicatives (purement sémantiques) à la grammaire permet de remettre en cause par des tests sémantiques une analyse syntaxique qui a réussi. Cela permet d'analyser correctement des langages ambigus syntaxiquement mais dont la sémantique statique permet de lever l'ambiguïté. Un texte *LET est en fait une grammaire définissant la syntaxe contextuelle d'un langage.

Ce chapitre explicite le déroulement d'un programme *LET en fonction des tests et actions qu'il contient.

4.2 Définitions

4.2.1 Exécution d'un programme

L'exécution d'un programme consiste en l'essai de sa racine. Celle-ci étant obligatoirement définie par une règle unique, il s'agit donc d'essayer le développement de cette règle.

4.2.2 Essai d'un développement

L'essai d'un développement consiste en l'essai successif de ses alternants jusqu'à l'obtention d'un succès (dans ce cas les alternants suivants ne seront pas essayés) ou l'échec de tous ses alternants (d'où échec du développement).

4.2.3 Essai d'un alternant

L'essai d'un alternant revient à chercher une succession de règles définissant les notions utilisées dans cet alternant, et telles que leurs essais successifs réussissent.
Les règles de la première notion appelée sont essayées dans l'ordre où elles sont données en utilisant les valeurs des paramètres en entrée. Lorsque l'appel de l'une d'elle réussit, les règles de la deuxième notion sont essayées en utilisant les valeurs fournies par l'appel réussi

précédemment. Et ainsi de suite jusqu'à l'obtention d'une succession d'appels positifs ou l'échec de toutes les successions de règles possibles.

Remarque :
Lorsque l'appel d'une règle réussit, les autres règles de la même notion ne sont pas éliminées pour autant. Si ce premier appel positif ne permet pas de réussir complètement l'alternant, les autres règles seront essayées à leur tour (Cf. 4.2.5).

4.2.4 Réussite et échec d'un appel de règle

Un appel de règle réussit si :

1) la confrontation des paramètres en entrée avec les arguments correspondants réussit,
2) le développement de la règle réussit,
3) la confrontation en sortie réussit.

Un appel de règle peut donc échouer soit à cause des confrontations, soit à cause de son développement.

4.2.5 Tests et actions

Une action est une règle dont le corps réussit toujours. Notons tout de suite que l'appel d'une règle action peut quand même échouer car les confrontations en entrée ou en sortie peuvent comporter des tests.
Un test est une règle dont le corps peut échouer, c'est-à-dire qui peut rendre un résultat négatif même après réussite des confrontations. Un test est repéré par un point d'interrogation en tête de sa description.

Grammaticalement, une action est une règle dont le développement peut être vide, il sera donc toujours trouvé dans le texte analysé.
Les qualificatifs «test» et «action» s'applique normalement aux règles. Par extension, on les appliquera aux appels de notions de la manière suivante : un appel de notion est une action s'il réussit pour toutes les valeurs possibles de ses paramètres effectifs, c'est-à-dire s'il existe une règle action applicable pour toute valeur de ces paramètres.

4.2.6 Indéterminisme local

Lorsqu'une règle r réussit pour l'appel d'une notion n dans un alternant a, elle fournit généralement des résultats qui servent à instancier des variables encore inconnues. Ces variables vont être utilisées pour l'essai des notions appelées plus loin dans l'alternant a. Si ces valeurs font échouer toutes les règles d'une notion m appelée après n dans l'alternant a, le résultat positif de la règle r est remis en cause et les instanciations effectuées alors sont effacées. Cela donne une certaine possibilité d'indéterminisme. Toutefois, lorsqu'un résultat positif est obtenu pour un alternant, les autres possibilités sont abandonnées. C'est pourquoi l'on parle d'**indéterminisme local**.

On peut considérer que les règles différentes définissent autant de pseudo-notions différentes. Une notion est alors l'union de ses pseudo-notions. Un appel de notions est alors équivalent à une mise en facteur de ces pseudo-notions.

Exemple :

A(...) : ... .
A(...) : ... .
B(...) : ... .
B(...) : ... .
C(...) : A(...) , B(...) .

est équivalent à : **C(...) : (A_1(...) ; A_2(...)) , (B_1(...) ; B_2(...)) .**
soit : **C(...) : A_1(...) , B_1(...) ;**
A_1(...) , B_2(...) ;
A_1(...) , B_2(...) ;
A_2(...) , B_2(...) .

Rappelons que le contrôle statique des successions d'appels (Cf. 3.1.2) peut amener à simplifier ce développement si des règles de A et B sont incompatibles (Cf. 3.6.4).

Remarque :

a) Le fait qu'une règle ne puisse fournir qu'un seul résultat positif à chaque appel s'applique également aux règles à plusieurs alternants. Il n'est donc pas indifférent de regrouper ou non des alternants en une seule règle.

N(>x) : A(x) ; B(x) .	ne peut fournir qu'un résultat ;
N(> :x) : A(x) .	si la première règle réussit, on peut quand même
N(>x) : B(x).	être amené à essayer la deuxième.

b) Par contre, si l'on est sûr que **A(x)** et **B(x)** sont incompatibles (si **A(x)** réussit **B(x)** échoue et réciproquement), le regroupement des alternants en une seule règle diminue la taille du programme fabriqué par le compilateur StarLet/GL.

Exemple utile :

```
AFFIXES :
bool :: oui ; non .
x :: ... .
y :: ... .
NOTIONS : ?non a(> x ) : a'( x , non ) .
a' ( > x , bool > ) / _y :   a ( x , _y ) , affec ( oui, bool ) ;
                             affec( non, bool).
affec(> bool, bool>) : VRAI . ?a(>x, y>) : ... .
```

La règle **a'** ne pouvant fournir qu'un résultat :
si **a** réussit, **a'** renvoie oui et son appel est rejeté dans **non a**,
si **a** échoue, **a'** renvoie non et son appel est accepté.

4.3 Cohérence des tests et des actions

4.3.1 Principe général : règles à un alternant.

Une règle déclarée action doit réussir quelque soient les valeurs affectées, lors de son appel, à ses variables paramètres en entrée. Pour cela, il doit exister une séquence de règles actions correspondant aux notions appelées dans son développement pour chaque combinaison possible de valeurs pour ces variables paramètres en entrée.
Une règle déclarée test doit pouvoir échouer pour au moins une combinaison de valeurs de ses variables paramètres en entrée. Pour cela, son développement doit contenir une notion pour laquelle il n'existe pas de règle applicable ou uniquement des règles tests applicables pour cette combinaison de valeurs.

Exemples :

```
AFFIXES :       x :: ENTIER .
                liste :: x liste ; vide .

NOTIONS :
                éditer( >vide) : VRAI .
                éditer( >x liste) : écrire(x) , éditer(liste) .
```

La première règle est bien une action car son développement est VRAI.
La deuxième règle est bien une action car :

- **écrire** est une action applicable quelle que soit la valeur de **x** ;
- la première règle **éditer** est une action applicable si **liste** est de la forme **x liste** et la deuxième règle **éditer** est une action applicable si **liste** est de la forme **vide**, ce qui couvre bien toutes les valeurs possibles de **liste**.

```
NOTIONS :
                verif(> _x _liste) : VRAI .
                essai(> liste) : verif(liste) , éditer(liste) .
```

Cette dernière règle est un test car si **liste** est de la forme **vide**, il n'y a pas de règle applicable à l'appel de **verif**.

```
NOTIONS :
                ?verif(> x liste) : non égal(x,0) , verif(liste) .
                verif(> vide) : VRAI .
                essai(> liste) : verif(liste) , éditer(liste) .
```

Avec ces déclarations, **essai** est un test car si **liste** est de la forme **x liste** la seule règle applicable à l'appel de **verif** est un test.

4.3.2 Rôle des variables locales

Dans l'application des énoncés précédents, il faut tenir compte des variables locales aux règles. En effet, certaines valeurs des paramètres en entrée peuvent entraîner l'instanciation de variables locales par des valeurs qui peuvent être rejetées par des appels de notions suivantes, ce qui implique que le développement est un test.

Exemples :

```
NOTIONS : ?traiter( >liste) / liste1 : trier( liste,liste1 ) , verif( liste1 ) .
trier( >liste,liste1> ) : ... .
?verif( >x liste) : non égal(x,0) , verif(liste) .
verif( >vide) : VRAI .
```

Si le résultat de l'action **trier** contient un zéro, l'appel du test **verif** sera rejeté, donc **traiter** est un test.

4.3.3 Alternants successifs

Lorsqu'un développement est composé de plusieurs alternants, ils sont essayés l'un après l'autre jusqu'à ce que l'un d'eux réussisse et alors, les alternants suivants sont abandonnés. En conséquence, si un alternant réussit toujours (Cf. 4.3.1), les suivants ne seront jamais essayés. Un alternant qui n'est pas le dernier doit donc pouvoir échouer.

4.4 Cas particuliers

4.4.1 La pseudo-notion VRAI

L'appel de la pseudo-notion VRAI réussit toujours. En conséquence, un alternant VRAI ne peut être que le dernier d'une règle et cette règle est alors une action. Ce cas correspond généralement aux notions facultatives, c'est-à-dire aux notions dont le développement peut être vide dans le langage analysé.

Exemple :

```
variables locales :     symb(«/»), liste de (variables) séparées par(«,») ;
                        VRAI .
```

Il n'y a pas obligatoirement des variables locales dans une règle.
Remarquons que, vu l'imbrication de la syntaxe et de la sémantique, des notions au développement vide vis-à-vis du langage source, peuvent être des tests. Ce sont alors des prédicats sémantiques qui servent à orienter l'analyse de langages intermédiaires pour l'analyse sémantique ou la génération de code.

4.4.2 Récursivité

La récursivité peut amener une situation ou des règles récursives peuvent être déclarées au choix comme des tests ou des actions sans, apparemment violer la règle de cohérence des tests et des actions.
Cela se présente lorsqu'une notion récursive n'appelle, à part elle-même, que des actions couvrant toutes les valeurs possibles des variables manipulées.

L'exemple le plus simple est le suivant :

```
AFFIXES :      liste :: vide ; x liste.
               x :: ENTIER.
NOTIONS :
( ?)parcourir(> x liste) : traiter(x), parcourir(liste).
parcourir( > vide) : VRAI .
traiter( > x) : ...
```

La règle **traiter** étant une action pour toute valeur de **x**, la deuxième règle de **parcourir** étant une action pour la forme **vide** de **liste**,
Si l'on déclare que la première règle est un test, cela est cohérent car elle peut échouer pour certaines valeurs de liste mais si elle est déclarée comme une action, cela est aussi cohérent car elle réussira toujours.
En réalité, les règles de ce type réussissent toujours et doivent donc être déclarées comme actions. Si la deuxième règle est un test ou si elle n'existe pas, la première est bien évidemment un test.
Ce cas très simple est détecté par le vérificateur qui le signale comme une erreur si la première règle est déclarée comme un test. Mais la même situation avec une règle supplémentaire n'est pas détectée.

Exemple :

```
AFFIXES :      liste :: vide ; x liste ; y liste .
               x :: ENTIER.
               y :: CARAC.
NOTIONS :
( ?)parcourir(> x liste) : traiter(x), parcourir(liste).
( ?)parcourir(> y liste) : traiter(y), parcourir(liste).
parcourir(> vide) : VRAI.
traiter(>x) : ....
traiter(>y) : ....
```

Les deux premières règles peuvent être, conjointement, déclarées comme test ou comme action sans que le vérificateur ne signale d'erreur.
Comme précédemment, ces règles sont en réalité des actions et devraient être déclarées comme telles.

4.4.3 Négation

De même que l'essai d'une règle ne donne qu'un résultat et non l'ensemble des résultats (c'est-à-dire des valeurs des paramètres en sortie pour lesquelles la règle est satisfaite), de même il n'est pas possible d'obtenir la négation d'une règle, c'est-à-dire une deuxième règle qui donnerait l'ensemble des valeurs pour lesquelles la première n'est pas satisfaite. Il n'est même pas possible d'obtenir comme résultat de la deuxième règle une seule valeur qui ne satisfait pas la première car tout échec efface les valeurs calculées.
On peut cependant écrire la négation booléenne d'une règle c'est-à-dire une deuxième règle qui permette de répondre à la question : est-ce que la règle échoue pour une certaine valeur de ses paramètre en entrée ?

Le procédé est le suivant :
Soit **R(> x_1, ... , > x_n , y_1 > , ... , y_m >) : ... .**

Ecrivons la règle qui permet de savoir si **R** réussit, sans chercher à connaître ses résultats y_i :

```
R'( > x1, ... , > xn, bool> ) / _y1, ... , _ym :
        R(x1, ... ,xn,_y1, ... ,_ym) , affec(vrai, bool) ;
        affec (faux, bool) .
```

Pour tester si **R** a échoué avec les valeurs données $x_1, \ldots, x_n$, il suffit alors d'appeler **R'** de la manière suivante : **R'(x_1, ... , x_n , faux)**

4.4.4 Ordre des règles

Les règles étant essayées dans l'ordre ou elles sont données, celui-ci n'est indifférent que lorsque les règles sont indépendantes parce qu'elles traitent des valeurs différentes des paramètres en entrée.

Exemples :

```
traiter( > vide) : ... .
traiter( > x liste) : ... .
```

peuvent être inversées, de même que :

```
?traiter( > x) : positif(x), ... .
?traiter( > x) : négatif(x), ... .
```

Il faut par contre faire attention dans certaines situations :
règle poubelle :

```
?traiter( > x liste) : ...
traiter( > liste) : ... .
```

La deuxième règle, certainement destinée à traiter les valeurs qui auraient fait échouer l'appel de la première (y compris la valeur vide), ne doit pas être placée devant car elle empêcherait l'appel de l'autre règle. Remarquons qu'une optimisation effectuée par le vérificateur permet de se rendre compte que l'autre règle est inutile si elle est placée derrière.

4.4.5 Dernière notion d'un alternant

Pour la dernière notion appelée dans un alternant, la réussite de l'essai d'une règle entraîne la réussite de l'alternant en question et la sortie de la règle appelante. Les autres règles de la notion appelée ne seront donc pas essayées. L'appel d'une notion en fin de développement rend donc ses règles exclusives, c'est-à-dire qu'une seule d'entre elles donne un résultat. Cette situation peut être forcée pour tous les appels, quel que soit leur position, par l'emploi d'une règle intermédiaire.

> l'appel **traiter(...)** est remplacé par **traiter'(...)**
> avec **traiter'(...) : traiter(...).**
> La règle **traiter'** ne fournissant qu'un résultat, une seule des règles «traiter» en fournira un.

5 Flots de données

5.1 Déclarations des variables

Toutes les variables doivent être déclarées avant d'être utilisées. Nous considérons trois sortes de variables.

- les **variables paramètres** sont déclarées par leur apparition dans les paramètres formels d'une règle.
- les **variables locales** doivent apparaître dans la liste des variables locales (après un «/») dans l'entête d'une règle.
- les **variables globales** sont déclarées dans le paragraphe «VARIABLES» (Cf. 2.5.3).

Les variables paramètres et locales sont propres à une règle. Deux règles peuvent avoir des variables de même nom mais ce sont des variables différentes. De plus, la valeur d'une variable locale ou paramètre est propre à un appel de la règle.
En cas d'appels successifs ou récursifs d'une même règle, ses variables peuvent avoir des valeurs différentes d'un appel à l'autre. Les variables globales sont communes à toutes les règles d'un module et éventuellement d'autres modules (cf. 6).

5.2 Sens des paramètres

En STARLET/GL, les paramètres ont obligatoirement un sens «entrée» ou «sortie». Cela signifie que les règles ne sont jamais réversibles comme elles peuvent l'être en PROLOG par exemple. Cette contrainte permet un contrôle automatique du flot des données et impose au programmeur une vision claire de ce flot.
Les variables en STARLET sont des variables au sens de la programmation en logique : en ce sens, elles ne peuvent être modifiées. Lorsque ces contraintes seront trop fortes pour la réalisation envisagée (structure de données construites progressivement), on utilisera des variables globales (cf 5.5) qui ne subissent pas ce contrôle et peuvent ainsi être modifiées.

5.3 Règles sur les variables locales et paramètres

Le contrôle est très strict sur les variables locales :

- toute variable utilisée doit avoir d'abord reçu une valeur
- toute valeur rangée dans une variable doit être ensuite utilisée

Un paramètre en entrée est initialisé à l'appel de la règle. La première utilisation d'une variable locale ou paramètre en sortie doit être en position de sortie pour qu'elle soit initialisée.
La valeur d'un paramètre en sortie est utilisée pour renvoyer un résultat à la fin de l'essai d'une règle. Un paramètre en entrée ou une variable locale doit être au moins une fois en position d'entrée pour que sa valeur soit utilisée.

5.4 Variables Inutiles

Il arrive fréquemment qu'une variable soit nécessaire à l'expression d'une règle alors que sa valeur n'est pas utilisée. Ceci se présente chaque fois que l'on veut extraire une partie d'une structure, fixer une contrainte de type sans soucis de la valeur dans ce type ou faire une utilisation partielle des résultats d'une règle. Pour éviter les messages d'erreurs dus à leur non utilisation, de telles variables doivent avoir un nom qui commence par un blanc souligné («_»).

Exemples :

```
AFFIXES :      liste :: elem liste ; vide .
```

1) Règle fournissant le premier d'une liste non vide

```
NOTIONS :
                premier ( > elem _liste , elem > ) : VRAI .
```

2) Si l'on dispose d'une règle

```
NOTIONS :
                décomposer ( > elem liste , elem > , liste > ) : VRAI .
```

On pourra prendre le premier d'une liste (sans s'occuper de la suite) de la manière suivante :

```
NOTIONS :
        exemple ( > liste ) / _liste1, elem :
                décomposer ( liste, elem, _liste1 ), traiter (elem).
```

3) Contrainte de forme : on désire traiter une liste non vide

```
NOTIONS :
        non vide ( > _elem _liste) : VRAI .
```

5.5 Test de confrontation en sortie

Une variable ne peut être modifiée mais elle peut être réinstanciée, à condition que ce soit avec la même valeur (remplacement uniforme). Cela implique un test (le plus souvent dynamique) lors de la réception de la deuxième valeur. De même une constante peut être un argument en position de sortie : ce qui signifie que la règle appelée ne sera acceptée que si le résultat fourni est cette valeur.

Exemples :

```
AFFIXES :
genre :: masc ; fem ; indef .
nombre :: sing ; plur .
NOTIONS :
nom ( genre > , nombre > ).
article ( genre > , nombre > ) .
$ l'article et le nom doivent être de même genre. Le nom déterminera
                        le genre si l'article ne l'indique pas (les) $
groupe / genre , nombre :
        article (indef, nombre ) , nom (genre, nombre ) ;
        article (genre , nombre ) , nom (genre, nombre ) .
```

5.6 Variables Globales

Les variables globales peuvent être utilisées dans n'importe quelle règle, elles peuvent être modifiées. Ce ne sont pas des variables (inconnues) au sens de la programmation en logique et de ce fait, elles ne sont pas concernées par les contrôles effectuées par le vérificateur.

Exemple

6 Compilation séparée

6.1 Modules

Le texte d'un programme StarLet peut être découpé en modules. La syntaxe d'un module (qui ne contient pas la racine : Cf. 2.5.1) est la suivante :

module : symbole («MODULE :») , nom de module , symbole («.») , paragraphes , partie secrète .
nom de module : suite de lettres et de chiffres .

Le nom d'un module doit être identique au radical du nom du fichier qui le contient. Dans une application, un module et un seul doit contenir la racine du programme.

6.2 Importations

Les déclarations (non secrètes Cf. 6.3) contenues dans un module peuvent être importées dans d'autres modules. Il suffit dans ces derniers de faire figurer le nom du module importé dans un paragraphe «« suivant la syntaxe :

importation de module : symbole(«AVEC :») , liste de modules , symbole(«.») .
liste de modules : nom de module , symbole(«,») , liste de modules ; nom de module .

6.3 Déclarations secrètes

Certaines déclarations sont locales à un module : ce sont toutes celles qui sont placées après le titre «SECRET :» .

partie secrète : symbole(«SECRET :») , paragraphes ; VRAI .

Il faut remarquer que, pour assurer la cohérence des compilations des différents modules, les déclarations exportées ne peuvent utiliser des déclarations secrètes :

- une procédure publique ne peut avoir un paramètre d'un type secret ;
- un affixe (type) public ne peut avoir dans son développement un affixe dont la déclaration est secrète.

6.4 Fonctions externes

Il est également possible d'utiliser des fonctions écrites en C qui doivent être déclarées par leur en-tête seule dans le paragraphe des notions :

règle de notion : déclaration de notion externe.
déclaration de notion externe : en-tête de notion , effets de bords , symbole(«.») .

Le nom de la fonction C doit être celui de la notion StarLet dans lequel on a remplacé les caractères spéciaux (espaces, tirets, apostrophes) par un souligné et inséré un souligné à la place de chaque paramètre. En outre, lorsqu'une notion externe dispose de plusieurs règles, le nom de chaque fonction est suffixé par un souligné et le numéro de la règle.

Exemples :

inf égal(>a,>b). donne inf_egal__
non égal(>Entier,>Entier1).
non égal(>Carac,>Carac1). donne non_egal___1
non_egal___2

Les fonctions réalisant des actions ne doivent pas renvoyer de résultat (type void). Les fonctions réalisant des tests doivent renvoyer un entier avec la convention habituelle en C pour les valeurs booléennes : 0 pour l'échec et tout autre pour un succès.

Les paramètres de types prédéfinis ont le type correspondant en C : char, char *, long int, float. Les variables de type énumération sont de type «long int» (se reporter aux listes de compilations pour connaître les représentations des différentes valeurs de l'énumération).

Les arbres sont toujours accédés par des pointeurs sur un tableau d'entier longs (Cf. zzpt dans /usr/include/starlet.h). L'élément 0 de ce tableau contient sur 16 bits (poids forts) des informations pour la gestion mémoire et sur 16 bits (poids faibles) le numéro de l'alternant d'affixe concerné. Les éléments suivants du tableau contiennent les éléments variables de l'alternant d'affixe.

Lorsqu'un paramètre est en sortie, on transmet en fait un pointeur sur un objet du type décrit ci-dessus et la fonction fabriquant le résultat doit réserver la place si nécessaire (chaînes et arbres)

Exemple :

```
AFFIXES :    le :: e le ; vide .
             e :: ENTIER .
NOTIONS :    fabriquer une liste(> : e, le >).
```

```
/*        Programme C  */
#include <starlet.h>
void fabriquer_une_liste__(e, le)
int e ;zzpt * le ;
 {        zzpt le1 ;
          if(e>0)
                    {le1= ...
                    * le=malloc(12) $ liste non vide $
                    (*le)->zztbl [0]= 1<<16 + 1 ;
                    (*le)->zztbl [1]=e ;
                    (*le)->zztbl [2]=le1 ; }
          else
                    {*le=malloc(4) ; $liste vide $
                    (*le)->zztbl [0]= 1<<16 + 2 ;
                    $ l'élément «vide» n'occupe pas de place $}
```

Les effets de bords

Lorsqu'une fonction externe modifie une variable globale, il est possible d'automatiser les reprises (restitutions d'anciennes valeurs lors des échecs d'alternants).
Ces variables doivent avoir pour nom zz suivi d'une lettre et pour type «long int». Il suffit alors d'introduire la déclaration qui spécifie qu'une fonction externe modifie cette variable :

effets de bords : symbole(«%») , lettre , effets de bords ; VRAI .

Remarquez que les variables zzs, zzo, zzl sont utilisées par les fonctions prédéfinies (Cf. 7).

7 Les notions prédéfinies

Un certain nombre de notions sont fournies ; écrites en C pour plus d'efficacité, elles sont accessibles via une interface chargée pour chaque compilation : /usr/include/starlet.isl.

Ces fonctions réalisent :

- un analyseur lexicographique pour grammaires LL(k) ;
- un jeu d'entrées et sorties conversationnelles et sur fichiers ;
- des opérations arithmétiques et sur chaînes de caractères.

Les déclarations de ces notions externes particulières, utilisent les règles d'affixes suivantes :

```
Carac : CARAC .
Chaîne : CHAINE .
Entier : ENTIER .
Reel : REEL.
 : Carac LC ; vide .
```

Ces affixes peuvent être utilisés pour d'autres variables ou dans d'autres règles d'affixes, en particulier pour des équivalences (règles simples, Cf. 2.1). Le nom **vide** peut être utilisé comme constante dans d'autres règles d'affixes.

Exemple :

mot : LC .

Ces fonctions prédéfinies sont destinées à faciliter l'écriture de prototypes de traducteurs. Nous ne prétendons pas couvrir tous les besoins ; il ne faudra donc pas hésiter à modifier ces fonctions pour les adapter à des besoins particuliers (analyseur LL(1) ...) ou à en écrire d'autres (Cf. 6.4).

7.1 L'analyseur LL(k)

Cet ensemble de fonctions gère un tampon dans lequel des retours en arrière sont effectués à chaque changement d'alternant (Cf. 6.4 %s).
Les variables nécessaires et le tampon sont initialisés au démarrage du programme, le fichier source est normalement stdin, et le texte source est lu jusqu'à la fin de fichier (CTRL D au clavier).
Des fonctions permettent de modifier le comportement de l'analyseur :

?ouvrir source(>chaîne).

Le paramètre indique le chemin d'accès du fichier à ouvrir ; il y a échec si l'ouverture n'est pas possible (Cf. 7.4 perror).

fixer fin de texte(>Carac).

Permet de spécifier le caractère qui indique la fin du texte ; avec les fonctions suivantes, cela permet de traiter un texte par morceau (ligne par ligne par exemple).

re init lex.

Réinitialise l'analyseur en effaçant tous les caractères déjà reconnus. Pour l'instant, cette fonction doit être utilisée (à des endroits où l'on est sur qu'il n'y aura plus de retours) lorsque l'on veut traiter des textes longs (plus de 10000 octets).

sauter source.

Ignore tous les caractères, même s'ils n'ont pas été reconnus, jusqu'à, et y compris, la fin du texte.

Les fonctions de reconnaissance suivantes examinent le texte source à la position actuelle.

?symbole(>Chaîne)%s.

Reconnaît la présence de la chaîne et avance derrière celle-ci en cas de réussite ;

?c'est un(>Chaîne).

Cherche la chaîne mais n'avance en aucun cas ;

?ce n'est pas un(>Chaîne).

Réussit si la chaîne n'est pas présente ;

?lettre(Carac>)%s.
?minuscule(Carac>)%s.
?majuscule(Carac>)%s.
?chiffre(Carac>)%s.

Réussissent si le caractère actuel est du type voulu, renvoient ce caractère et avancent ;

?chiffre(Entier>)%s.

Renvoie la valeur binaire du chiffre (0-9) ;

?caractère quelconque(Carac>)%s.

N'échoue que si on est en fin de texte ;

?fin de texte.

Réussit si on est en fin de texte ;

blancs%s.

Avance tant qu'il y a des caractères espaces, tabulations ou fin de ligne.

Position(Entier>)%s.

Renvoie un entier caractérisant la position dans le texte source.

extraire source(>Entier,Chaine>)%s.

Place dans la chaîne les caractères extraits du texte source depuis la position caractérisée par l'entier, jusqu'à la position courante non comprise. Permet d'extraire d'un seul tenant les caractères composants un nom, une constante entière, etc.

numero colonne(Entier >)%s.

Si le source est divisé en ligne séparées par des retours-chariot, on obtient la position dans la ligne actuelle, compte tenu des tabulations.

7.2 Ecritures avec reprises

Deux fichiers (liste et objet) sont gérés à l'aide d'un tampon dans lequel des reprises sont effectuées en cas d'échecs d'alternants.

Notez que ces fichiers sont liés au fichier source en ce sens que les notions «re init lex» et «sauter source» vident ces tampons et donc suppriment toute possibilité de retour en arrière du point actuel. Les tampons sont également vidés automatiquement en fin d'exécution. Initialement, le fichier objet est stdin et le fichier liste est stderr.

Les fonctions d'accès à ces fichiers sont :

?ouvrir liste(>Chaîne)
?ouvrir objet(>Chaîne).

Cf. «ouvrir source» ci-dessus ;

Ecritures de chaînes de caractères, de caractères, d'entiers et de réels sur les fichiers objet et liste :

objet(>Chaîne)%o.
objet (>Carac)%o.
objet (>Entier)%o.
objet (>Reel)%o.
liste(>Chaîne)%l.
liste (>Carac)%l.
liste (>Entier)%l.
liste (>Reel)%l.

7.3 Entrées et sorties sans reprises

Un jeu de notions permet de lire et d'écrire sur les fichiers standard ou sur d'autres fichiers sans tampon donc sans possibilité de reprises même en cas d'échec.
Les notions d'accès aux fichiers standard stdin et stdout sont :

?lire(Chaîne>).
?lire ligne (Chaîne>).

Lecture jusqu'à la fin de la ligne

?lire (Carac>).
?lire (Reel>).
?lire (Entier>).

échouent si on ne trouve pas de chiffres. Les 4 règles de lecture échouent par ailleurs s'il n'y a plus rien à lire (fin de texte).

écrire (>Chaîne).
écrire (>Carac).
écrire (>Entier).
écrire (>Reel).
a la ligne.

Les notions d'accès à d'autres fichiers sont :

ouvrir(>Chaine, Entier>,>Chaine1).

le premier paramètre est le nom du fichier, le troisième indique le sens d'accès désiré («rwa» comme en C), la fonction attribue au fichier un numéro permettant d'utiliser les fonctions suivantes :

fermer(>Entier).
?lire (chaîne>)dans (>Entier).
?lire (Carac>)dans(>Entier).
?lire (Entier>)dans(>Entier1).
?lire (Reel>)dans(>Entier).
écrire (>chaîne)dans(>Entier).
écrire (>Carac)dans(>Entier).
écrire (>Entier)dans(>Entier1).
écrire (>Reel)dans(>Entier).

7.4 Fonctions arithmétiques

add(>Entier1,>Entier2,Entier3>).
add(>Reel1,>Reel2,Reel3>).
sous(>Entier1,>Entier2,Entier3>).
sous(>Reel1,>Reel2,Reel3>).
mul(>Entier1,>Entier2,Entier3>).
mul(>Reel1,>Reel2,Reel3>).
?div(>Entier1,>Entier2,Entier3>).
?div(>Reel1,>Reel2,Reel3>).

ces deux fonctions échouent si le diviseur est nul ;

convertir(>Entier, Reel>).
Convertir(>Reel,Entier>).
?non égal(>Entier1,>Entier2).
?non égal(>Reel1,>Reel2).
?non égal (>Carac1,>Carac2).
?non égal (>Chaine1,>Chaine2).
?sup(>Entier1,>Entier2).
?sup (>Reel1,>Reel2).
?inf(>Entier1,>Entier2).
?inf (>Reel1,>Reel2).

Remarquez que «égal» se définit simplement par

égal(>Entier,>Entier) : VRAI .

7.5 Traitement des chaines

concatener(>Chaine1,>Chaine2,Chaine3>.

Concaténation de chaînes.

convertir(>LC, chaîne>).

Transforme une liste de caractères en chaîne.

longueur(>Chaîne,Entier>).

Renvoie dans Entier la longueur de Chaine

?entre(>Carac1,>Carac2,>Carac3).

Teste si le code ASCII de Carac1 est entre (et y compris) les codes de Carac2 et Carac3.

extraire(>Chaine,>Entier,Carac>).(>LC, chaîne>).

Extrait un caractère d'une chaîne

extraire(>Chaine, >Entier1, >Entier2, Chaine1)

Extrait de Chaine, Entier2 caractères, à partir de la position Entier1, et les range dans Chaine2

7.6 Notions système

nbarg(Entier>).

Donne le nombre d'arguments de la commande de lancement du programme ;

?arg no(>Entier, chaîne>).

Donne, s'il existe, l'argument de la phrase d'appel dont le rang est indiqué par **Entier**;

perror(>chaîne).

Edition de la dernière erreur système.

exit(>).

Sortir du programme en renvoyant le code spécifié.

system(>chaîne).

Exécuter la commande écrite en « shell ».

getenv(>chaîne1,Chaine2>).

Récupérer la valeur d'une variable de l'environnement

ANNEXES

A Grammaire hors-contexte de STARLET

---

programme : *symbole (" RACINE : ") , développement de notion , paragraphes.*

paragraphes : *paragraphe , paragraphes ;*
VRAI.

paragraphe : *symbole ("AFFIXES :"), règles d'affixes ;*
symbole ("NOTIONS :"), règles de notions ;
déclaration de variables ;
définitions de constantes ;
importation de modules .

---

règles d'affixes : *règle d'affixe, règles d'affixes ;*
règle d'affixe.

règle d'affixe : *affixe variable , symbole (":"), développement d'affixe ;*
affixe variable, symbole ("=") , règle d'affixe.

développement d'affixe : *affixe prédéfini ;*
suite d'alternants d'affixe .

suite d'alternants d'affixe : *alternant d'affixe , symbole (";") ,*
suite d'alternants d'affixe ;
alternant d'affixe.

alternant d'affixe : *affixe , alternant d'affixe ; affixe.*

affixe : *affixe variable ; affixe constant .*

affixe variable : *suite de lettres.*

affixe constant : *suite de lettres .*

affixe prédéfini : *symbole ("ENTIER") ; symbole ("REEL") ;*
symbole ("CARAC") ; symbole("CHAINE").

---

règles de notions : règle de notion, règles de notions ;
règle de notion.

règle de notion : en-tête de notion, symbole (":"),
développement de notion.

en-tête de notion : marque, notion et paramètres formels,
déclaration de variables locales.

marque : marque de test ; marque d'action.

marque de test : symbole ("?").

marque d'action : *symbole ("!") ; VRAI.*

notion et paramètres formels : *nom de notion, paramètres formels,*
suite notion et paramètres formels ;
nom de notion.

suite notion et paramètres formels : *nom de notion, paramètres formels ,*
suite notion et paramètres formels ;
nom de notion ;
VRAI.

paramètres formels : *symbole (“(“), liste de paramètres, symbole (“)“).*

liste de paramètres : *paramètre, symbole (“,“), liste de paramètres ; paramètre.*

paramètre : *paramètre en entrée ; paramètre en sortie.*

paramètre en entrée : *symbole (“:“), expression d’affixes.*

paramètre en sortie : *expression d’affixes, symbole (“:“).*

déclaration de variables locales : *symbole (“/“), liste de variables ; VRAI .*

développement de notion : *alternant de notion, symbole (“;“), développement de notion ; alternant de notion, symbole (“.“) ; symbole (“VRAI“), symbole (“.“).*

alternant de notion : *utilisation de notion, symbole (“,“), alternant de notion ; utilisation de notion.*

utilisation de notion : *nom de notion et arguments.*

nom de notion et arguments : *nom de notion, arguments, suite de la notion et arguments ; nom de notion.*

suite de la notion et arguments : *nom de notion, arguments, suite de la notion et arguments ; nom de notion ; VRAI.*

arguments : *symbole (“(“), liste d’arguments, symbole (“)“).*

liste d’arguments : *argument, symbole (“,“), liste d’arguments ; argument.*

argument : *expression d’affixes.*

déclaration de notion externe : *en-tête de notion, effets de bords, symbole(“.“) .*

effets de bords : *symbole(“%“) , lettre , effets de bords ; VRAI .*

---

définitions de constantes : *symbole (“CONSTANTES :“), liste de définitions de constantes .*

liste de définitions de constantes : *définition de constante, symbole (“,“) , liste de définitions de constantes ; définition de constante.*

définition de constante : *affixe constant , symbole (“=“), constante .*

déclaration de variables : *symbole (“VARIABLES :“), liste de variables , symbole (“.“).*

module : *symbole (“MODULE :“) , nom de module , symbole (“.“) , paragraphes , partie secrète .*

nom de module : *suite de lettres et de chiffres .*

importations de module : *symbole(“AVEC :“) , liste de modules , symbole(“.“) .*

liste de modules : *nom de module , symbole(“,“) , liste de modules ; nom de module .*

---

liste de variables :	*variable, symbole (“,“), liste de variables ;* *variable.*
nom de variable :	*affixe variable, suffixe .*
suffixe :	*entier décimal ; VRAI.*
variable :	*nom de variable ;* *symbole (“_“) , nom de variable.*
constante :	*entier décimal ;* *symbole (“’ “) , caractère, symbole (“’ “) ;* *symbole (“\ “»), suite de caractères, symbole (“\ “»).* *suite de caractères : caractère, suite de caractères ;* *VRAI.*
expression d’affixes :	*atome affixe, expression d’affixes ;* *atome affixe.*
atome affixe :	*variable ; affixe constant ; constante.*
nom de notion :	*suite de lettres chiffres tirets apostrophes et espaces.*
chiffre / _Carac :	*chiffre(_Carac).*
lettre / _Carac :	*lettre(_Carac).*
caractère / _Carac :	*caractère quelconque(_Carac).*
entier décimal :	*chiffre , entier décimal ; chiffre .*
suite de lettres :	*lettre , suite de lettres ; lettre.*
suite de lettres et de chiffres :	*lettre, autres lettres et chiffres.*
autres lettres et chiffres :	*lettre, autres lettres et chiffres ;* *chiffre, autres lettres et chiffres ; VRAI.*
suite de lettres chiffres tirets apostrophes et espaces :	*lettre, autres lctae.*
autres lctae :	*lettre, autres lctae ;* *chiffre, autres lctae ;* *symbole(“-“) , autres lctae ;* *symbole(“_“), autres lctae ;* *symbole(“”“) , autres lctae ;* *symbole(“ “), autres lctae ;* *VRAI.*

B Références Bibliographiques

[BEN 85] **J. BENEY & J.F. BOULICAUT**.
Des spécifications grammaticales à la programmation logique : Le compromis STARLET,
Rapport Interne INSA Lyon, Juin 1985.

[BEN 86] **J. BENEY & J.F. BOULICAUT**.
STARLET : Un langage pour une Programmation Logique Fiable.
Rapport Interne INSA Lyon, Juin 1986.

[CLE 77] **J.C CLEAVELAND & R.C UZGALIS**.
Grammars for Programming Languages.
Elsevier, Programming Languages Series 4, 1977.

[DER 88] **P. DERANSART & M. JOURDAN & B. LOHRO**.
Attribute Grammars : Definitions, Systems and Bibliography.
Springer-Verlag, LNCS 323, August 1988, 232 p.

[KOS 71] **C.H.A. KOSTER**.
Affix grammars. in ALGOL 68 Implementation (J.E. Peck ed.),
North-Holland 1971.

[MAL 84] **J. MALUSZYNSKI**.
Towards a programming language based on the notion of two-level grammars.
Theoretical Computer Science, 1984, Vol.28, p.13-43.

[MAR 76] **M. MARCOTTY& H.F. LEDGARD & G.V. BOCHMANN**.
A Sampler of Formal Definition.
ACM Computing Surveys, 1976, Vol. 8, n°2, p. 191-276.

[PER 80] **F.C.N PEREIRA & D.H.D WARREN**.
Definite Clause Grammars for Language Analysis : a survey of the formalism and a comparison with Augmented Transition
Networks. Artificial Intelligence, 1980, Vol. 13, p. 231-278.

[SIM 81] **M. SIMONET**.
W-grammaires et logique du premier ordre pour la définition et l'implantation des langages.
Thèse d'état, USMG Grenoble, Juillet 1981.

[WAT 83] **D.A WATT & O.L MADSEN**.
Extended Attribute Grammars.
The Computer Journal, 1983, Vol. 26, n° 2, p. 142-153.

[WIJ 76] **A. VAN WIJNGAARDEN & Al.**
Revised Report on the algorithmic language ALGOL 68.
Springer-Verlag, 1976.
La version primitive de ce rapport a été traduite en français par le groupe ALGOL de l'AFCET : définition du langage algorithmique ALGOL 68, publié chez Hermann, 1972.

C Manuel d'utilisation du compilateur starlet sous unix

Le compilateur StarLet analyse et vérifie un module de programme StarLet, fabrique un module C s'il n'y a pas d'erreurs et lance alors le compilateur C.

Ligne de commande

Le compilateur StarLet est activé par la commande

star [-m] [-i repertoire] ***nom_du_module_à_compiler***

où les options signifient :

-m : le compilateur est appelé par makesl (voir ci-dessous)

-i répertoire : les condensés (.isl, voir ci-dessous) doivent être rangés dans le répertoire spécifié.

Fichiers

Les fichiers manipulés ont comme nom celui du module suivi d'un suffixe indiquant leur rôle :

.sl : fichier source, contient le texte du programme ;

.ls : liste de compilation, avec les éventuelles erreurs ;

.isl : StarLet condensé qui sera relu lors de la compilation d'autres modules ;

.c, **.h** : programme C fabriqué par le compilateur StarLet, à son tour compilé pour obtenir un programme exécutable (.o).

Variables d'environnement

Le comportement du compilateur peut être contrôle par les variables suivantes :

CC : nom du compilateur C

par défaut : cc

exemple : gcc ou /usr/ucb/cc

CFLAGS : paramètres à transmettre au compilateur C

par défaut : -O

exemple : -I/usr/local/include pour aller chercher les fonctions prédéfinies dans un répertoire non standard.

Bibliothèque de règles prédéfinies

Les règles prédéfinies (chapitre 7) sont décrites dans deux fichiers chargés par le compilateur :

starlet.isl : description des règles en StarLet

starlet.h : description des fonctions C correspondantes

Assembleur de modules

Le programme **makesl** permet de mettre à jour un programme composé de plusieurs modules StarLet (sans avoir à rédiger un fichier makefile). Il utilise les directives AVEC contenues dans les modules StarLet.

Appel :

makesl *nom_du_module_racine* **[-C** *nom_de_module_écrit_en_C* **]**

L'assembleur de modules crée un fichier nommé makefilesl destiné à être relu par l'utilitaire **make**. Les valeurs des variables **CC** et **CFLAGS** sont transmises au compilateur StarLet. L'accès à celui-ci peut être précisé par la variable **SLC** (défaut : **starlet**).
De plus, la variable **SLLIB** permet de préciser où se trouve la bibliothèque StarLet (libstarlet.a). Sa valeur par défaut dépend de l'installation.

Installation du compilateur sur un système Unix

Le kit StarLet comprend différents fichiers à placer dans les répertoires du système qui sont, par exemple :

- le compilateur : starlet — /usr/local/bin
- la bibliothèque de règles prédéfinies (chapitre 7) sous forme de :
  - fichiers objets : *predef.o, starpred.o , lexpred.o* — /usr/local/include
    dans ce cas, il faudra indiquer leur emplacement lors de l'édition de liens ; mais ils peuvent être réunis en une bibliothèque : libstarlet.a placée dans /usr/local/lib
- une description des règles en StarLet : starlet.isl — /usr/local/include
- une description en C : starlet.h — /usr/local/include

Pour de nouveaux systèmes ou machines, il peut être fourni un ensemble de modules écrits en C (.c et .h). Lors de leur compilation, on peut préciser un autre répertoire pour *starlet.isl.* Les répertoires pour libstarlet.a et starlet.h peuvent être modifiés et indiqués au compilateur C et au relieur par l'intermédiaire de **CFLAGS.**