COMPILATEURS ET INTERPRÈTES
2010 - 2011

JEREMY OTHIENO
othieno0@etu.unige.ch

Table des matières

I. Introduction

Le but du projet est d'adjoindre au langage Lista un traitement de listes linéaires simples de "Nombres au sens de Lista" sans limitation de longueur, ainsi qu'une extension supplémentaire à libre choix.

Les extensions doivent fonctionner avec les deux méthodes d'analyse lexico-syntaxique mises en pratique dans les étapes

- **l'analyse lexicale prédictive et descente récursive "manuelles"**
- **l'emploi de spécifications Flex et Bison**

La syntaxe des nombres sera la même que celle des nombres déjà utilisable dans le langage de départ. Il faut également un jeu minimal d'opérations prédéfinies sur ces listes linéaires de nombres.

En voici les étapes du projet en quelques mots

- **L'analyse lexicale** lit les caractères du source, et construit la séquence des terminaux la composant.
- **L'analyse syntaxique** vérifie que la structure de cette séquence est conforme à la syntaxe du langage.
- **L'analyse sémantique** contrôle la signification du code source.
- **L'évaluation directe** exécute un programme Lista sans passer par la machine Pilum, mais en utilisant des graphes sémantiques.
- **La machine Pilum** qui est une machine à pile permettant d'exécuter du code objet généré par les compilateurs du langage Lista.
- L'extension à choix qui sera une liste hétérogène, soit une liste qui stocke des valeurs de tout type offert par le langage Lista.

I.1. Installation de Lista de départ.

Le projet a été testé sur **`Linux version 2.6.32-5-amd64 (Debian 2.6.32-30)`**. Il m'a fallu installé **Flex** et **Bison** car ceux-ci n'étaient pas pré-installés.

I.2. Spécification des extensions.

Le document a pour but d'expliciter la démarche qui a été suivi pour implanter les listes d'éléments en Lista. On parle d'une **liste de nombres** au sens de Lista pour signaler une liste composée entièrement de nombres, et d'une **liste hétérogène** pour parler d'une liste contenant des valeurs de types quelconques. Au niveau du langage, ces deux listes ne sont pas distinguées! D'un point de vue relationnelle, on peut dire que

```
Listes de nombres ⊂ Liste hétérogène
```

Une liste est de la forme **[e_1, e_2, e_3, ... , e_{n-2}, e_{n-1}, e_n]** où

- les **crochets** sont les **délimiteurs** de la liste,
- les **éléments** de la liste sont de **type quelconque**, voir même des **expressions**,
- la **virgule** est le **séparateur** des éléments de la liste.

Les listes vides sont représentées par deux délimiteurs juxtaposés, soit "[]". Quelques opérateurs élémentaires sont définies pour la manipulation des listes:

- **`Car`** retourne le premier élément (la tête) d'une liste.
- **`Cdr`** retourne une liste, sans inclure son premier élément (la queue).
- **`Cons`** ajoute un élément au début d'une liste, tandis que
- **`Append`** ajoute l'élément à la fin.
- **`ConcatListes`** concatène deux listes pour en construire une nouvelle.
- **`LongueurListe`** retourne le nombre d'éléments d'une liste.

Quelques exemples de listes correctes sont

- **`[]`**, une liste vide.
- **`[1, 2, 3, 300, -4e+2, 0, -300]`**, une liste de nombres.
- **`[2 × 3 + 44, f(2), g("Hello, World")]`** où f et g sont des fonctions utilisateurs qui retournent des types définis dans le langage. On voit aussi qu'un des éléments est une expression.
- **`[Vrai, "Hello, World", 33, X, [[], 300, (1 + 2)/3, Faux]]`** où X est une variable définie. On constate aussi qu'on a une liste vide dans la liste.

En revanche, ces exemples sont incorrectes

- **`[[], [[], [300, (1 + 2)/3, Faux], "Hello, World!"]`** car il manque un délimiteur.
- **`[Vrai "Hello, World", 33]`** car il manque un séparateur entre le premier et deuxième élément de la liste.
- **`[1, X, f(3)]`** où la variable X ou la fonction f n'a pas été définie.

I.3. La syntaxe concrète d'une liste (EBNF).

<DelimiteurGauche>::= '['
<DelimiteurDroit>::= ']'
<Separateur>::= ','

<ElementListe>::= <ExpressionLista>
<ElementsListe>::= <ElementListe> | <ElementListe> <Separateur> <ElementsListe>

<Liste>::= <DelimiteurGauche> <DelimiteurDroit>
<Liste>::= <DelimiteurGauche> <ElementsListes> <DelimiteurDroit>

Il faut noter que **<ExpressionLista>** comprend toute expression acceptée par le langage Lista.

II. L'analyse lexicale et syntaxique

« L'analyse lexicale se trouve tout au début de la chaîne de compilation. C'est la tâche consistant à décomposer une chaîne de caractères en unités lexicales, aussi appelées *tokens*. Ces tokens, "produits" à la demande de l'analyseur syntaxique, sont ensuite consommés par ce dernier. » -- Wikipedia.

« L'analyse syntaxique consiste à mettre en évidence la structure d'un texte, généralement un programme informatique (...) » -- Wikipedia.

Le **lexique d'un langage** définit **les mots qui le composent**. On parle de **terminaux** pour les mots du langage. Les terminaux sont en général les identificateurs, les chaînes de caractères, les constantes numériques, les mots clés du langage ou les marqueurs syntaxiques propres au langage.

On se contente d'ajouter de **nouvelles marqueurs syntaxiques** au langage Lista, à savoir "**[**" et "**]**", pour désigner les délimiteurs des listes. La virgule "**,**" fait partie de l'ensemble de terminaux déjà prédéfinis dans le langage Lista de départ.

Le **syntaxe** du langage **régit la forme des phrases**: les phrases acceptables au vu de la définition syntaxique appartiennent au langage, les autres n'y appartiennent pas. Un analyseur syntaxique est basé sur un **accepteur**, fonction booléenne indiquant si le texte source est conforme aux règles de grammaires définissant le langage. De plus, **seule la forme du code source est prise en compte dans l'analyse syntaxique**: la signification du code n'est quant à lui vérifié que dans l'analyse sémantique.

La définition syntaxique d'un langage s'appuie sur

- les **terminaux**, définis au niveau syntaxique, et
- des règles de bonne forme pour des séquences de terminaux appelés **notions non-terminales**.

Une méthode d'analyse est **déterministe** si toute phrase du langage peut être acceptée sans devoir revenir sur une tentative. Une méthode d'analyse **descendant et déterministe** est dite **prédictive**. Dans une méthode prédictive, le flot du contrôle est calqué sur la grammaire du langage que l'on analyse.

II.1. La méthode prédictive et descente récursive

La première chose à faire est d'ajouter de nouveaux terminaux au langage. Les terminaux de Lista sont décrits par l'énumération `TerminalP`. J'ajoute quatre terminaux qui représente les nouvelles marqueurs syntaxique mentionnées tout à l'heure.

```
¦ Modification de l'énumération TerminalP [Lexique/DecodageDOptionsLexique.h, 0081]

enum TerminalP
   {
   t_FIN,                  /* 0, sera retourne par 'yylex ()' lorsque 'yywrap ()'
                              retournera lui-meme une valeur non-nulle */

   t_NOMBRE,               t_IDENT,             t_CHAINE,

   t_PAR_GAUCHE,           t_PAR_DROITE,
   t_EGALE,                t_VIRGULE,
   ...

   t_CROCHET_GAUCHE,       t_CROCHET_DROIT,

   t_POINT_VIRGULE,        t_INTERROGE
   };
```

Ensuite je modifie l'analyseur lexical de Lista qui est implanté dans la classe `AnalyseurLexicalLista`. Seules deux méthodes de cette classe nous intéressent: `AnalyseurLexicalLista::AccepterUn-Terminal` qui associe à un caractère lu son terminal correspondant défini dans le langage, et la méthode `AnalyseurLexicalLista::TerminalSousFormeTextuelle` de fonctionnalité triviale

```
¦ Modification de AnalyseurLexicalLista::TerminalSo... [Lexique/AnalyseurLexicalLista.cc, 0217 à 0220]

Chaine
AnalyseurLexicalLista :: TerminalSousFormeTextuelle (TerminalP leTerminalP)
   {
   switch (leTerminalP)
      {
      ...
      case t_CROCHET_GAUCHE:
         return "           [";
      case t_CROCHET_DROIT:
         return "           ]";
   ...
   }
```

```
¦ Modification de AnalyseurLexicalLista::AccepterUn... [Lexique/AnalyseurLexicalLista.cc, 0405 à 0410]

TerminalP
AnalyseurLexicalLista :: AccepterUnTerminal ()
   {
   ...
      case '[':
         fTerminalPCourant = t_CROCHET_GAUCHE;
         break;
      case ']':
         fTerminalPCourant = t_CROCHET_DROIT;
         break;
   ...
   }
```

Une fois que les terminaux sont mis en place, il faut modifier l'analyseur syntaxique pour que le langage soit capable d'accepter syntaxiquement une liste. L'analyseur syntaxique est implanté dans la classe `AnalyseurDescendantLista`. J'ajoute un nouvel accepteur syntaxique -- implanté dans la méthode `AnalyseurDescendantLista::Liste` -- qui a pour but de vérifier la forme syntaxique des listes.
Avant d'implanter l'accepteur, il faut noter que je considère une liste comme une expression. Par conséquent, il faut modifier la méthode `AnalyseurDescendantLista::Expression` pour inclure ce fait. La raison pour laquelle je considère une liste comme une expression est parce que cela me permet d'avoir des imbrications de listes, une propriété que j'ai défini pour les listes.

```
¦ Modification de AnalyseurDescendantLista [Syntaxe/AnalyseurDescendantLista.h, 0070]

class AnalyseurDescendantLista
   {
   ...
   void                         Expression ();
   void                         Terme ();
   void                         Facteur ();

   void                         Liste ();

   void                         AppelDeFonction ();
   void                         Arguments ();

   void                         ErreurSyntaxique (Chaine leMessage);
   };  // AnalyseurDescendantLista
```

```
¦ Modification de AnalyseurDescendantLista::Expr... [Syntaxe/AnalyseurDescendantLista.cc, 0210 à 0221]

void
AnalyseurDescendantLista :: Expression ()
{
   if (fTerminalPCourant == t_CROCHET_GAUCHE)
   {
      // Consommer le crochet gauche et appeler l'accepteur syntaxique des listes.
      Avancer();
      Liste();

      // Verifier que le terminal courant est un crochet droit, sinon il y a une erreur!
      if (fTerminalPCourant != t_CROCHET_DROIT)
         ErreurSyntaxique("']' attendu a la fin d'une liste!");
      else
         Avancer();
   }
   else
   {
      ...
   }
}  // AnalyseurDescendantLista :: Expression
```

Et pour finir l'implantation de l'accepteur. Tout ce que fait l'accepteur est de vérifier que chaque élément de la liste est suivi d'un séparateur et que la liste finit par le bon délimiteur. Aucune vérification des éléments de la liste est faite, parce qu'on n'a pas besoin ...

```
¦ Implantation de AnalyseurDescendantLista::Liste [Syntaxe/AnalyseurDescendantLista.cc, 0352 à 0370]

void
AnalyseurDescendantLista :: Liste()
{
   // On ne veut pas traiter le terminal ']'.
   if (fTerminalPCourant != t_CROCHET_DROIT)
   {
      // Accepter l'expression.
      Expression();

      if (fTerminalPCourant == t_VIRGULE)
      {
         // Consommer le separateur et faire une appel recursive pour traiter la sous-liste.
         Avancer();
         Liste();
      }
      else if (fTerminalPCourant != t_CROCHET_DROIT && fTerminalPCourant != t_VIRGULE)
         ErreurSyntaxique("Il manque une ',' ou un ']' apres l'element de la liste.");
   }
}
```

II.1.a. Traces d'exécution.

```
COMMANDE: ./ListaSyntPredictifDescenteRecursive -g -rsynt

Contenu du fichier source 'debugcode/syntaxe.lista':
-------------------------------------------------------------
A = [1, 2, 3];
B = [1, 2, "trois", 2*2, [Vrai], A];
-------------------------------------------------------------

        Ident      A
                   =
                   [
        Reel               1.000000
                   ,
        Reel               2.000000
                   ,
        Reel               3.000000
                   ]
                   ;
        Ident      B

--> Definition de fonction

                   =
                   [
        Reel               1.000000
                   ,
        Reel               2.000000
                   ,
        Chaine     trois
                   ,
        Reel               2.000000
                   *
        Reel               2.000000
                   ,
                   [
        Ident      Vrai
                   ]
                   ,
        Ident      A
                   ]
                   ;
        --- FIN ---

--> Definition de fonction

*** Code Lista lexicalement   correct   ***
*** Code Lista syntaxiquement correct   ***
```

II.2. Flex et Bison

Flex compile une grammaire régulière et produit le texte source dans un langage cible d'un analyseur du langage engendré par cette grammaire. Bison est similaire, mais traite le niveau syntaxique.

Comme au chapitre précédant, on commence par ajouter des notions terminaux. Les terminaux sont maintenant décrits par l'énumération `TerminalF`. Comme pour `TerminalP`, on ajoute les deux nouveaux terminaux -- identiques à ce qui a été ajouté dans `TerminalP`

```
¦ Modification de l'énumération TerminalFlexLista [Lexique/TerminauxFlexLista.h, 0032]

enum TerminalFlexLista
   {
   t_FIN,               // 0, sera retourne par 'yylex ()' lorsque 'yywrap ()'
                        // retournera lui-meme une valeur non-nulle

   t_NOMBRE,            t_IDENT,            t_CHAINE,

   t_PAR_GAUCHE,        t_PAR_DROITE,
   t_EGALE,             t_VIRGULE,

   t_PLUS,              t_MOINS,            t_FOIS,             t_DIVISE,
   t_CONCAT,

   t_CROCHET_GAUCHE,    t_CROCHET_DROIT,

   t_POINT_VIRGULE,     t_INTERROGE
   };
```

Ensuite, on ajoute les actions correspondantes à ces terminaux, pour l'analyseur lexicale comme pour l'analyseur syntaxique.

```
¦ Modification de l'ensemble des actions des terminaux [Lexique/LexiqueFlex.Flex, 0317 à 0326] et
¦ [Syntaxe/SyntaxeBison.Flex, 0325 à 0334]

...
"?"                         {
                            AppondreAuSourceCumule ();

                            return EnregistrerTerminal (t_INTERROGE);
                            }
"["                         {
                            AppondreAuSourceCumule ();

                            return EnregistrerTerminal (t_CROCHET_GAUCHE);
                            }
"]"                         {
                            AppondreAuSourceCumule ();

                            return EnregistrerTerminal (t_CROCHET_DROIT);
                            }
...
```

Et pour finir, la traduction d'un terminal en chaîne de caractères.

```
¦ Modification de TerminalSousFormeTextuelle [Lexique/LexiqueFlex.Flex, 0403 à 0406] et
¦ [Syntaxe/SyntaxeBison.Flex, 0412 à 0415]

Chaine
TerminalSousFormeTextuelle (TerminalFlexLista leTerminalFlexLista)
   {
   switch (leTerminalFlexLista)
      {
      ...
      case t_CROCHET_GAUCHE:
         return "           [";
      case t_CROCHET_DROIT:
         return "           ]";
   ...
   }   // TerminalSousFormeTextuelle
```

Ensuite j'ajoute les mêmes terminaux à la grammaire mais cette fois-ci en forme de tokens

```
¦ Modification des terminaux [Syntaxe/SyntaxeBison.Bison, 0057]

...
%token                          t_PLUS              t_MOINS
%token                          t_FOIS              t_DIVISE
%token                          t_CONCAT

%token                          t_CROCHET_GAUCHE    t_CROCHET_DROIT

%token                          t_POINT_VIRGULE     t_INTERROGE
```

Une fois que les terminaux ont été mis en place, il faut modifier un non-terminal existant, et en ajouter de nouveaux. On commence avec la modification du non-terminal `Expression`

```
¦ Modification du non-terminal Expression [Syntaxe/SyntaxeBison.Bison, 0202]

Expression
   :  t_MOINS Terme
   |  Expression t_PLUS Terme
   |  Expression t_MOINS Terme
   |  Terme
   |  Liste
   ;
```

Ensuite, j'ajoute des non-terminaux qui décriront la forme syntaxique des listes

- `DelimiteurListeGauche` vérifie qu'une liste commence par un crochet gauche, tandis que `DelimiteurListeDroit` vérifie que la même liste termine par un crochet droit.
- `ElementsListe` traversent une suite de non-terminaux récursivement tout en vérifiant que chacun est suivi du terminal `t_VIRGULE`, sauf dans le cas où la liste n'a qu'un seul élément.
- `Liste` regroupe les deux non-terminaux définis ci-dessus.

```
¦ Ajout des nouveaux non-terminaux [Syntaxe/SyntaxeBison.Bison, 0247 à 0267]

DelimiteurListeGauche
   :  t_CROCHET_GAUCHE
   ;

DelimiteurListeDroit
   :  t_CROCHET_DROIT
   |  error
      {
         std::cout << "--> ']' attendu a la fin d'une liste!" << std::endl;
      }
   ;

ElementsListe
   :  Expression
   |  Expression t_VIRGULE ElementsListe
   ;

Liste
   :  DelimiteurListeGauche DelimiteurListeDroit
   |  DelimiteurListeGauche ElementsListe DelimiteurListeDroit
   ;
```

Cette dernière modification marque la fin de ce qu'il faut modifier au niveau de l'analyse lexico-syntaxique. La prochaine étape serait de modifier l'analyse sémantique du langage.

II.2.a. Traces d'exécution

```
COMMANDE: ./ListaSyntFlexBison -g -rsynt

Contenu du fichier source 'debugcode/syntaxe.lista':
---------------------------------------------------------------
A = [1, 2, 3];
B = [1, 2, "trois", 2*2, [Vrai], A];
---------------------------------------------------------------

        Ident      A
                   =
                   [
        Reel               1.000000
                   ,
        Reel               2.000000
                   ,
        Reel               3.000000
                   ]
                   ;

--> Definition de fonction

        Ident      B
                   =
                   [
        Reel               1.000000
                   ,
        Reel               2.000000
                   ,
        Chaine     trois
                   ,
        Reel               2.000000
                   *
        Reel               2.000000
                   ,
                   [
        Ident      Vrai
                   ]
                   ,
        Ident      A
                   ]
                   ;

--> Definition de fonction

*** Code Lista lexicalement   correct   ***
*** Code Lista syntaxiquement correct   ***
```

III. L'analyse sémantique

La sémantique d'un langage est la **signification** véhiculée par les phrases de ce langage. Les aspects lexicaux et syntaxiques d'un langage ne sont qu'un support pour l'essentiel, à savoir la sémantique. On accepte syntaxiquement un **sur-langage** de celui que l'on veut compiler, quitte à **restreindre** ensuite ce que l'on peut accepter par des contrôles sémantiques.

Un exemple est qu'on peut **accepter** des fonctions **sans vérifier leur nombre de paramètres formels dans l'analyse syntaxique**. Or, **il faut que le nombre d'arguments dans les appels soit égal au nombre de paramètres formels**. L'analyse sémantique de Lista est chargée de faire cette correspondance.

L'analyse sémantique a pour but d'**effectuer les vérifications de signification** sur le code source en cours de compilation.

Pour cette analyse, on se base sur la définition du langage obtenue à l'analyse lexico-syntaxique, qui précise le sens des phrases bien formées syntaxiquement.

III.1. La méthode prédictive et descente récursive

Maintenant qu'une liste est acceptée syntaxiquement, on va enrichir la sémantique du langage en rajoutant une liste à l'ensemble de types prédéfinis en Lista. Le nouveau type aura pour nom `Liste`. En résumé, les étapes à suivre lors de l'implantation du type sont

1. L'implantation du type en dans le langage d'implantation de Lista, C++.
2. L'implantation du type en Lista, soit donner une description précise du type en Lista.
3. L'implantation de l'inférence du type. Il faut que le langage Lista soit capable de déterminer automatiquement qu'une valeur ou variable soit de type `Liste`.

III.1.a. Implantation des types en C++

Pour implanter les deux types en C++, je me base sur l'implantation **List** offert par la **Standard Template Library** (STL).

L'implantation complète de la structure de données se trouve dans le fichier **`Extensions/liste.cc`**. Voici sa déclaration complète

Remarque: Pour prendre en compte le nouveau extension dans le langage d'implantation, il ne faut pas oublier de modifier les makefiles nécessaires.

¦ Déclaration des types "Element", "IterateurListe" et "Liste" [TypesEtVersion/Types.h, 0060 à 0128]

```
// ---------------------------------------------------------------------------
// Liste de nombres
// ---------------------------------------------------------------------------

class Liste;
typedef Liste *            ListePTR;
typedef Chaine *           ChainePTR;


// Les types d'elements de la liste.
enum TypeElement
{
   kTypeNombre,            kTypeBooleen,
   kTypeChaine,            kTypeListe
};

// Un element de la liste.
typedef struct ElementDeListe
{
   union
   {
      Nombre               fNombre;
      bool                 fBooleen;
      ChainePTR            fChaine;
      ListePTR             fListe;
   };
   TypeElement             fType;
} Element;

// L'iterateur de la liste.
typedef std::list<Element>::iterator
                           IterateurListe;

class Liste
{
   private:
      std::list<Element>   fListe;
   public:
                           Liste();
                           Liste( std::list<Element> );
                           Liste( ListePTR );
                           ~Liste();

      IterateurListe       begin();
      IterateurListe       end();
      Element              car();
      Liste                cdr();
      void                 cons( Element& );
      void                 cons( Nombre& );
      void                 cons( Chaine& );
      void                 cons( bool& );
      void                 cons( Liste& );
      void                 cons( ListePTR );
      void                 append( Element& );
      void                 append( Nombre& );
      void                 append( Chaine& );
      void                 append( bool& );
      void                 append( Liste& );
      void                 append( ListePTR );
      void                 concat( Liste& );
      void                 concat( ListePTR );

      Nombre               taille();
      Chaine               str();              // Representation textuelle du contenu de la liste.
};
```

III.1.b. Implantation du type en Lista

Le langage d'implantation possède une structure capable de stocker et manipuler le nouveau type. Il en est temps que Lista soit capable de faire pareil. Pour ce faire, il est impératif de créer une description du type dans le langage Lista. Pour les besoins de l'analyse sémantique de Lista, les types suivants sont décrits par des sous-classes de `Type`

- Les types `TypeNombre`, `TypeBooleen`, `TypeChaine` et `TypeVide`, propres à Lista,
- `TypeInconnu`, pour décrire les constructions erronées, mal formée ou dont le type n'a pas pu être inféré,
- `TypeNonPrecise`, pour les cas de surcharge sémantique tels que des fonctions prédéfinies **`Si`** et **`Seq`**.

Le nouveau type `TypeListe` est ainsi ajouté à l'ensemble des sous-classes de `Type`

```
¦ Déclaration de TypeListe [Semantique/DescriptionDesTypes.h, 0151 à 0155]

class TypeListe : public Type
   {
public:
                              TypeListe ();
   };
```

```
¦ Implantation de TypeListe [Semantique/DescriptionDesTypes.cc, 0127 à 0139]

TypeListe :: TypeListe ()
   : Type (
      & LanguePredefinis :: TypeListe )
   {}
```

L'argument que prend le constructeur de la classe `Type` est une adresse d'une fonction qui retourne le nom du type prédéfini soit en français, soit en anglais, selon le choix de l'utilisateur.
Comme la méthode `LanguePredefinis::TypeListe` n'est pas encore définie, je la rajoute à la classe `LanguePredefinis`

```
¦ Modification de la déclaration de LanguePredefinis, LanguePredefinisFR et LanguePredefinisEN
¦ [LanguesPredefinis/LanguesPredefinis.h, 0050 à 0051, 0197 et 0294]

   virtual Chaine             TypeChaine () = 0;
                                 // virtuelle pure
   virtual Chaine             TypeVide () = 0;
                                 // virtuelle pure
   virtual Chaine             TypeListe () = 0;
                                 // virtuelle pure
   ...
   Chaine                     TypeChaine ();
   Chaine                     TypeVide ();
   Chaine                     TypeListe ();
   ...
```

```
¦ Implantation de TypeListe [LanguesPredefinis/LanguesPredefinis.cc, 0067 à 0069 et 0357 à 0359]

Chaine
LanguePredefinisFR :: TypeListe ()
   { return "TypeListe"; }

...
Chaine
LanguePredefinisEN :: TypeListe ()
   { return "ListType"; }
```

Enfin, `TypeListe` fait partie des types propres au langage Lista. Mais il reste un problème primordial: **le langage ne peut pas faire l'inférence de ce type**!

Dans la logique de déduction, l'inférence est un jugement qui consiste à tirer une conclusion d'une série de propositions reconnues pour vraies[1]. Par exemple, supposons que nous avons le code suivant

```
¦ Exemple de code source Lista.

x = [1, 2, 3, 4];
```

De ce que nous connaissons sur les listes, nous avons que

1. Toute liste commence avec un crochet gauche et termine avec un crochet droit,
2. les éléments d'une liste sont de type quelconque,
3. les éléments d'une liste sont séparés par une virgule.
4. Alors l'identificateur ***x*** correspond à une liste.

Dire que le langage ne peut pas faire l'inférence du type Liste, revient à dire, grosso modo, que le langage est incapable de faire une telle déduction.

III.1.c. L'inférence de type

Précisément, l'inférence de type consiste à déterminer automatiquement les types des différents identificateurs apparaissant dans un programme source d'après l'emploi qui en est fait, sans qu'ils soient indiqués explicitement.

```
¦ Exemple de code source C.

int main( void )
{
   int x = 10;                 // Le type 'int' est donné explicitement à la variable x.
   char *laChaine = "Hello";   // Idem pour le type 'char *' donné à la variable laChaine.

   return 0;
}
```

```
¦ Exemple de code source Lista.

x = 10;                        // Le type de la variable x (TypeNombre) est obtenu par l'inférence.
laChaine = "Hello";            // Idem pour le type TypeChaine donné à la variable laChaine.
```

Comme il a été indiqué précedemment, le langage Lista est incapable de déterminer si un identificateur donné prend le type `TypeListe`, e.g.

```
¦ Exemple de code source Lista.

L = [1, 2, 3];                 // Cette déclaration donne lieu à une erreur sémantique!
```

1 Définition tirée de wikipedia: http://fr.wikipedia.org/wiki/Inf%C3%A9rence

On va donc élargir les possibilités d'inférence de type de Lista. Les deux étapes à suivre sont

- la définition d'une nouvelle **variable logique** qui lie les types des identificateurs,
- et un **nouveau graphe sémantique** décrivant le type.

III.1.c.1. L'inférence de type: La variable logique

L'algorithme mis en place pour le langage Lista s'appuie sur la notion de **variable logique** que l'on rencontre entre autres en Prolog. Cette variable est initialement libre (sans valeur), et a pour but de lier, par une tentative d'unification (comme en Prolog), un type inconnu à un des autres types propres au langage. On appelle cette unification **l'identification**.

Un des **devoirs de l'analyse sémantique est de faire la correspondance entre les variables logiques de types à l'un des types existants** dans le langage Lista.

La classe `VariableLogiqueType` décrit une variable logique pouvant prendre un type comme valeur. On ajoute alors la variable logique `gTypeLogiqueListe` qui sera liée au type `TypeListe`.

```
¦ Déclaration de la nouvelle variable externe [Semantique/DescriptionDesTypes.h, 0187]

extern VariableLogiqueTypePTR               gTypeLogiqueNombre;
extern VariableLogiqueTypePTR               gTypeLogiqueBooleen;
extern VariableLogiqueTypePTR               gTypeLogiqueChaine;
extern VariableLogiqueTypePTR               gTypeLogiqueListe;

extern VariableLogiqueTypePTR               gTypeLogiqueVide;

extern VariableLogiqueTypePTR               gTypeLogiqueNonPrecise;

extern VariableLogiqueTypePTR               gTypeLogiqueInconnu;
```

```
¦ Déclaration de gTypeLogiqueListe [Semantique/DescriptionDesTypes.cc, 0165]

VariableLogiqueTypePTR        gTypeLogiqueNombre      = NULL;
VariableLogiqueTypePTR        gTypeLogiqueBooleen     = NULL;
VariableLogiqueTypePTR        gTypeLogiqueChaine      = NULL;
VariableLogiqueTypePTR        gTypeLogiqueListe       = NULL;

VariableLogiqueTypePTR        gTypeLogiqueVide        = NULL;

VariableLogiqueTypePTR        gTypeLogiqueNonPrecise  = NULL;

VariableLogiqueTypePTR        gTypeLogiqueInconnu     = NULL;
```

Pour qu'une variable logique soit utile, elle doit connaître le type qu'elle tentera d'unifer avec une certaine valeur. Pour ce faire, Lista a une instance de chaque type du langage. Une nouvelle instance de `TypeListe` est ajoutée et la nouvelle variable logique est initialisée

```
¦ Déclaration des nouvelles variables externes [DescriptionDesTypes.h, 0176]

...
extern TypePTR                      gTypeNombre;
extern TypePTR                      gTypeBooleen;
extern TypePTR                      gTypeChaine;
extern TypePTR                      gTypeListe;

extern TypePTR                      gTypeVide;
```

```
¦ Déclaration des nouvelles variables [Semantique/DescriptionDesTypes.cc, 0154 et 0165]

...
TypePTR                         gTypeBooleen          = NULL;
TypePTR                         gTypeChaine           = NULL;
TypePTR                         gTypeListe            = NULL;

TypePTR                         gTypeVide             = NULL;
...

VariableLogiqueTypePTR          gTypeLogiqueChaine    = NULL;
VariableLogiqueTypePTR          gTypeLogiqueListe     = NULL;
VariableLogiqueTypePTR          gTypeLogiqueVide      = NULL;
...
```

```
¦ Initialisation de gTypeListe et gTypeLog... [Semantique/DescriptionDesTypes.cc, 0187 et 0214 à 0217]

void
InitialiserDescriptionDesTypes ()
   {
   gTypeInconnu               = new TypeInconnu;
   gTypeNonPrecise            = new TypeNonPrecise;

   gTypeNombre                = new TypeNombre;
   gTypeBooleen               = new TypeBooleen;
   gTypeChaine                = new TypeChaine;
   gTypeListe                 = new TypeListe;

   gTypeVide                  = new TypeVide;

   // ---------------------------------------------------------------------------

   gTypeLogiqueInconnu     =
      new VariableLogiqueType (
         gLANGUE_Predefinis -> TypeInconnu (),
         gTypeInconnu );
   gTypeLogiqueNonPrecise  =
      new VariableLogiqueType (
         gLANGUE_Predefinis -> TypeNonPrecise (),
         gTypeNonPrecise );

   gTypeLogiqueNombre      =
      new VariableLogiqueType (
         gLANGUE_Predefinis -> TypeNombre (),
         gTypeNombre );
   gTypeLogiqueBooleen     =
      new VariableLogiqueType (
         gLANGUE_Predefinis -> TypeBooleen (),
         gTypeBooleen );
   gTypeLogiqueChaine      =
      new VariableLogiqueType (
         gLANGUE_Predefinis -> TypeChaine (),
         gTypeChaine );
   gTypeLogiqueListe =
      new VariableLogiqueType (
         gLANGUE_Predefinis -> TypeListe (),
         gTypeListe );

   gTypeLogiqueVide        =
      new VariableLogiqueType (
         gLANGUE_Predefinis -> TypeVide (),
         gTypeVide );
   }  //InitialiserDescriptionDesTypes
```

III.1.c.2. L'inférence de type: Le graphe sémantique

En Lista, les analyses lexicale, syntaxique et sémantique se font en une passe, produisant en sortie un

graphe sémantique pour **chaque définition de fonction et évaluation**. Un graphe sémantique (un graphe acyclique orienté) n'est autre qu'**une structure de données utilisée pour représenter la sémantique des instructions et expressions du langage**. Finalement, **ces graphes contiennent la même sémantique que le code source compilé**.

Les graphes sémantiques sont construits explicitement comme structures de données dans le langage d'implantation, par exemple pour les utiliser ensuite dans une passe ultérieure de compilation.
En Lista, les graphes sémantiques sont construits pour **l'exécution directe**: l'exécution des programmes Lista en évaluant directement les expressions sous leur forme de graphe sémantique!

La classe `DescrSemantique` décrit les noeuds sémantiques de Lista. Avant d'en rajouter une sous-classe qui représente le noeud sémantique d'une liste, il faut implanter une nouvelle structure de données capable de stocker une suite de descriptions sémantiques, `ListeDescrSemantique`

```
¦ Définition du type ListeDescrSemantique [Semantique/PreDeclarationsSemantique.h, 0042]
¦ Remarque: Il ne faut pas oublier d'inclure le header <list> dans le même fichier.

typedef std::list<DescrSemantiquePTR>  ListeDescrSemantique;
```

La raison pour laquelle j'ajoute cette structure est tout simplement parce que le contenu d'une liste n'est pas limitée à des valeurs primitives! Une liste peut contenir des expressions par exemple.

On passe à la nouvelle sous-classe de `DescrSemantique`, `ValeurListe` qui décrit le noeud sémantique d'une liste. L'explication des champs est la suivante

- La variable `fListeDeDescriptionsSemantique` est utilisée pour stocker la description sémantique (le graphe sémantique) de chacun des éléments de la liste. De cette façon, une liste peut ressembler à `[0, 1, 1+1, 2*3/9+4e-66, sin(26), Faux, [], "Hello, World"]`.

- Le constructeur de la classe prend un argument de type `ListeDescrSemantique` et donc une liste de descriptions sémantiques.

- La méthode `ValeurListe::Ecrire` sert à afficher le graphe sémantique de la structure (le contenu de `fListeDescrSemantique`).

- Les méthodes `ValeurListe::Evaluer` et `ValeurListe::Synthetiser` sont appelées lors de l'évaluation directe du graphe sémantique et de la synthèse de code objet Pilum, respectivement. Ces notions nous nous intéressent pas pour le moment et par conséquent, le corps des méthodes `Synthetiser` est laissé vide, tandis que les méthodes `Evaluer` retournent une valeur générique: `ValeurLista()`. Pour voir l'implantation complète, veuilliez-vous référer aux chapitres Évaluation Directe et Synthèse de Code Pilum.

Après avoir implanté les outils nécessaires pour l'inférence de `TypeListe` -- les graphes sémantiques -- il restera à modifier **l'analyseur sémantique** de façon à ce qu'il puisse les utiliser.

```
¦ Déclaration de la classe ValeurListe [Semantique/DescrSemantique.h, 0173 à 0189]

class ValeurListe : public DescrSemantique
   {
```

```
private:
   ListeDescrSemantique       fListeDeDescriptionsSemantique;
public:
                              ValeurListe ( ListeDescrSemantique );

   virtual void               Ecrire (int lIndentation);

   virtual ValeurLista        Evaluer (
                                 ContexteDEvaluationPTR leContexteDEvaluation );
#if defined SYNTHESE_PILUM
   virtual void               Synthetiser (
                                 ContexteDeSynthesePTR leContexteDeSynthese );
#endif /* SYNTHESE_PILUM */
   };
```

```
¦ Implantation de la classe ValeurListe [Semantique/DescrSemantique.cc, 0411 à 0519]

ValeurListe :: ValeurListe (ListeDescrSemantique descr)
   :  DescrSemantique (gTypeLogiqueListe)
   {
   fListeDeDescriptionsSemantique = descr;
   }

void
ValeurListe :: Ecrire (int lIndentation)
   {
   DescrSemantique::Ecrire( lIndentation );

   if (fListeDeDescriptionsSemantique.size() == 0)
      std::cout << "[ ]" << std::endl;
   else if (fListeDeDescriptionsSemantique.size() > 0)
      {
      std::cout << "[" << std::endl;

      ListeDescrSemantique::iterator it = fListeDeDescriptionsSemantique.begin();

      // Afficher le graphe semantique de tous les elements de la liste.
      ++ lIndentation;

      for (; it != fListeDeDescriptionsSemantique.end();  it++)
         (*it) -> Ecrire( lIndentation );

      -- lIndentation;

      DescrSemantique::Ecrire( lIndentation );
      std::cout << "]" << std::endl;
      }
   }

ValeurLista
ValeurListe :: Evaluer (ContexteDEvaluationPTR leContexteDEvaluation)
{
   return ValeurLista();
}

#if defined SYNTHESE_PILUM

void
ValeurListe ::
Synthetiser()
   {}

#endif /* SYNTHESE_PILUM */
```

III.1.c.3. L'inférence de type: L'analyseur sémantique

L'analyseur sémantique du langage Lista est chargé de la création des graphes sémantiques, ainsi que de faire l'inférence. La modification de l'analyseur sémantique se fait en deux étapes.

La première étape consiste à simplement rajouter une nouvelle méthode à la déclaration de la classe `AnalyseurListaPredictifDescenteRecursive`. La méthode `AnalyseurListaPredictifDescente-Recursive::Liste` aura pour but de créer le graphe sémantique d'une liste, et de faire des contrôles pour assurer que les éléments soient du bon type: des expressions quelconques pour une liste hétérogène, et surtout que tout élément de la liste est suivi soit d'un séparateur, soit d'un délimiteur.

```
¦ Modification de la déclaration de AnalyseurListaPredictifDescenteRecursive
¦ [Semantique/AnalyseurPredictifDescenteRecursiveLista.h, 0096]

   ...
   DescrSemantiquePTR         Facteur ();
   DescrSemantiquePTR         FacteurIdent ();
   DescrSemantiquePTR         Liste ();
   ...
```

```
¦ Implantation de Liste [Semantique/AnalyseurPredictifDescenteRecursiveLista.cc, 1575 à 1629]

DescrSemantiquePTR
AnalyseurListaPredictifDescenteRecursive :: Liste ()
{
   ListeDescrSemantique    listeDeDescriptionsSemantique;
   DescrSemantiquePTR      descr;

   // Consommer le terminal.
   Avancer();

   // Si le terminal courant est un crochet droit (une liste vide), consomme-le, sinon on traite
   // les elements de la liste.
   if (fTerminalPCourant != t_CROCHET_DROIT)
   {
      while (fTerminalPCourant != t_CROCHET_DROIT && fTerminalPCourant != t_FIN)
      {
         // Obtenir le graphe semantique d'un element de la liste.
         descr = Expression();

         // Ajouter la description semantique a la liste de descriptions.
         listeDeDescriptionsSemantique.push_back( descr );

         // Le terminal qui suit une expression doit etre un nombre. Par consequent, le terminal
         // courant devrait etre une virgule. Si c'est le cas ...
         if (fTerminalPCourant == t_VIRGULE)
         {
            // ... on consomme la virgule.
            Avancer();

            // Et il faut que ce que suit une virgule soit toujours une expression (!), et non
            // pas un crochet droit, ou une point-virgule.
            if (fTerminalPCourant == t_CROCHET_DROIT || fTerminalPCourant == t_POINT_VIRGULE)
            {
               ErreurSyntaxique("Expression attendue apres un ','");
               return gDescrSemantiqueInconnue;
            }
         }
         // Dans le cas ou une virgule ne suit pas l'expression, le terminal courant peut etre
         // un crochet droit. Si ce n'est pas un crochet droit, la structure est fausse.
         else if (fTerminalPCourant != t_VIRGULE && fTerminalPCourant != t_CROCHET_DROIT)
         {
            ErreurSyntaxique("',' ou ']' attendu apres l'expression");
            return gDescrSemantiqueInconnue;
         }
      }
      // Si on quitte la boucle et que le terminal courant ne soit pas un crochet droit, il y a
      // une erreur! On retourne alors la description semantique d'une valeur inconnue
      // (gDescrSemantiqueInconnue).
      if (fTerminalPCourant != t_CROCHET_DROIT)
         return gDescrSemantiqueInconnue;
```

```
   }
   // On a atteint le crochet droit. Le crochet est consomme, et la description semantique de la
   // liste retournee.
   Avancer();
   return new ValeurListe( listeDeDescriptionsSemantique );
}
```

Dans le cas d'une inférence réussie, la fonction retourne la description sémantique correspondant à la liste. Lors d'un echec, la description sémantique d'une valeur du type inconnu, `TypeInconnu`, est retournée, au moyen de la variable `gDescrSemantiqueInconnue`. Vous constatez aussi que la méthode définie ci-dessus ressemble à l'analyseur syntaxique, la seule différence est qu'elle stocke les éléments de la liste.

La deuxième étape de l'inférence consiste à ajouter un cas de traitement du terminal '**[**' à la fonction `AnalyseurListaPredictifDescenteRecursive::Facteur` qui permet à l'analyseur sémantique d'appeler `AnalyseurListaPredictifDescenteRecursive::Liste` lorsque ce terminal est lu

```
¦ Implantation de la méthode Liste [AnalyseurListaPredictifDescenteRecursive.cc, 0792 à 0798]

DescrSemantiquePTR
AnalyseurListaPredictifDescenteRecursive :: Facteur ()
   {
   switch (fTerminalPCourant)
      {
      ...
      case t_CROCHET_GAUCHE:
         {
         // On retourne la description semantique de la liste en cas d'inference reussie,
         // ou celle d'une valeur inconnue dans le cas contraire.
         return Liste ();
         }
      ...
      }  // switch
   }  // AnalyseurListaPredictifDescenteRecursive :: Facteur
```

III.1.c.4. L'inférence de type: Traces d'exécution

```
COMMANDE: ./ListaPredictifDescenteRecursive -ms -mi

Contenu du fichier source '../exemples/ListesExemple.lista':
-------------------------------------------------------------
A = [1, 2, 3];
B = [1, 2, 3, A, ["Hello, ", "World"]];
-------------------------------------------------------------

[VariableLogique_9 "fonction 'A'" -> TypeLogiqueLIBRE] decrit le type de la fonction 'A'


--> On lie la variable logique libre [VariableLogique_9 "fonction 'A'" -> TypeLogiqueLIBRE] a la valeur TypeListe

[VariableLogique_10 "fonction 'B'" -> TypeLogiqueLIBRE] decrit le type de la fonction 'B'


--> On lie la variable logique libre [VariableLogique_10 "fonction 'B'" -> TypeLogiqueLIBRE] a la valeur TypeListe

*** Code Lista lexicalement   correct   ***
*** Code Lista syntaxiquement correct   ***
*** Code Lista semantiquement correct   ***
```

III.1.d. Les fonctions prédéfinies.

Le type `Liste` **est acceptée sémantiquement** par le langage Lista. On se contente maintenant

d'enrichir la sémantique du langage en ajoutant de nouvelles fonctions prédéfinies. L'implantation des fonctions prédéfinies se fait en trois étapes

- l'ajout des identificateurs prédéfinis, correspondant à des fonctions données,
- l'ajout de nouveaux types de noeuds sémantiques pour les fonctions,
- l'ajout des fonctions prédéfinies selon leur arité.

Une différence importante par rapport à l'étape précedante et qu'il n'est pas nécessaire de faire l'unification de types des fonctions prédéfinies car elles ont un **typage statique**, i.e. leur type est défini lors de la compilation, épargnant l'analyseur sémantique la tâche de déterminer leur types. En revanche, c'est le cas pour les fonctions utilisateurs car elles maintiennent un typage dynamique. Pour ajouter une fonction prédéfinie à la sémantique du langage, il faut connaître

- **l'identificateur** correspondant à la fonction, soit son nom[2] dans le langage.
- **l'arité** de la fonction. Une fonction (opérateur) peut être zéroaire, unaire, binaire ou tertiaire.
- le **type** de la valeur retournée par la fonction, ainsi que celui de ses paramètres.

Pour le moment, il n'est pas nécessaire de connaître le comportement de la fonction lorsqu'elle est evaluée. J'en parlerai au chapitre suivant.

La fonction	Catégorie de l'opérateur	Type des paramètres	Type de la valeur de retour	Identificateur de la fonction en français (FR) et en anglais (EN)
car	Unaire	`TypeListe`	**`TypeNonPrecise`**	`Car` (FR et EN)
cdr	Unaire	`TypeListe`	`TypeListe`	`Cdr` (FR et EN)
longueur	Unaire	`TypeListe`	`TypeNombre`	`LongueurListe` (FR) `ListLength` (EN)
concat	Binaire	`TypeListe`, `TypeListe`	`TypeListe`	`ConcatListes` (FR) `ConcatLists` (EN)
cons	Binaire	**`TypeNonPrecise`**, `TypeListe`	`TypeListe`	`Cons` (FR et EN)
append	Binaire	**`TypeNonPrecise`**, `TypeListe`	`TypeListe`	`Append` (FR et EN)

Parmi les types propres au langage Lista, je rappelle que `TypeNonPrecise` est décrit pour les cas de surcharge sémantique. La fonction **`cdr`** retourne un élément de la liste, de type quelconque, tandis que les fonctions **`cons`** et **`append`** permettent d'ajouter des éléments de type quelconque à une liste. Ces types sont determinés lors de l'appel à la fonction, graçe à ses arguments d'appel.

2 En Lista, ces noms peuvent changer selon la langue choisie à l'exécution du compilateur.

En Lista

- les noms des identificateurs sont fournis par la classe `LanguesPredefinis`,
- toute fonction prédéfinie est une sous-classe de `FonctionPredefinie`. Cette super-classe contient le nom de la fonction, son genre ainsi que le type de variable logique qu'elle retourne.
- le noeud sémantique d'une fonction prédéfinie est implanté (selon l'arité de la fonction) dans les sous-classes `OperateurZeroaire`, `OperateurUnaire`, `OperateurBinaire` ou `Operateur-Ternaire`, elles-mêmes des sous-classes de `DescrSemantique`.

La première des choses est d'établir les identificateurs de nos fonctions prédéfinies. Pour cela, il faut modifier la classe `LanguesPredefinis` une fois encore

```
¦ Modification de la déclaration de LanguesPre... [LanguesPredefinis/LanguesPredefinis.h, 0158 à 0169]

class LanguePredefinis
   {
public:
   ...
   virtual Chaine             Pour () = 0;
                                 // virtuelle pure

   virtual Chaine             Car () = 0;
                                 // virtuelle pure
   virtual Chaine             Cdr () = 0;
                                 // virtuelle pure
   virtual Chaine             Cons () = 0;
                                 // virtuelle pure
   virtual Chaine             Append () = 0;
                                 // virtuelle pure
   virtual Chaine             ConcatListes () = 0;
                                 // virtuelle pure
   virtual Chaine             LongueurListe () = 0;
                                 // virtuelle pure

   };  // LanguePredefinis

...
¦ Modification de la déclaration de LanguePredefinisFR et LanguePredefinisEN
¦ [LanguesPredefinis/LanguesPredefinis.h, 0261 à 0266 et 0358 à 0363]

   ...
   Chaine                     Car ();
   Chaine                     Cdr ();
   Chaine                     Cons ();
   Chaine                     Append ();
   Chaine                     ConcatListes ();
   Chaine                     LongueurListe ();
   ...
```

```
¦ Implantation des nouvelles methodes de LanguesPredefinis
¦ [LanguesPredefinis/LanguesPredefinis.cc, 0294 à 0316 et 0294 à 0316]

Chaine
LanguePredefinisFR :: Car ()
   { return "Car"; }
Chaine
LanguePredefinisFR :: Cdr ()
   { return "Cdr"; }
Chaine
LanguePredefinisFR :: Cons ()
   { return "Cons"; }
Chaine
LanguePredefinisFR :: Append ()
   { return "Append"; }
```

```
Chaine
LanguePredefinisFR :: ConcatListes ()
   { return "ConcatListes"; }
Chaine
LanguePredefinisFR :: LongueurListe ()
   { return "LongueurListe"; }
...

Chaine
LanguePredefinisEN :: Car ()
   { return "Car"; }
Chaine
LanguePredefinisEN :: Cdr ()
   { return "Cdr"; }
Chaine
LanguePredefinisEN :: Cons ()
   { return "Cons"; }
Chaine
LanguePredefinisEN :: Append ()
   { return "Append"; }
Chaine
LanguePredefinisEN :: ConcatListes ()
   { return "ConcatLists"; }
Chaine
LanguePredefinisEN :: LongueurListe ()
   { return "ListLength"; }
...
```

Les solutions prises pour resoudre le problème de surcharge sémantique sont les suivantes

- Pour le **`Car`**, ajouter un argument au constructeur du graphe sémantique de cette fonction, qui prend le type logique du résultat après l'évaluation. Pour connaître ce type logique sans faire l'évaluation, j'ajoute une méthode qui traverse un graphe sémantique en essayant de trouver la description sémantique du premier élément d'une liste donnée comme argument à la fonction.
- Je crée un nouveau genre de fonction qui représente les fonctions qui manipulent les listes. Ce genre de fonction determine les types corrects des arguments d'appel, et créer le graphe d'appel de la fonction, avec les bons types, contrairement aux types non précisés.

Pour le cas de la fonction **`Car`**, j'ajoute trois méthodes

- `DescrSemantique::TraverserLeGrapheSemantique` qui, comme son nom l'indique, traverse un graphe sémantique pour trouver la description sémantique de la liste qui a été donné comme argument à une fonction **`Car`**. La méthode prend en compte les cas où l'on se trouve avec des imbrications de fonctions, e.g. "**Car(Cdr([1, 2, 3, 4]))**". Par conséquent, elle retourne la liste de descriptions sémantiques selon les opérations qui ont été faites sur celle-ci; dans l'exemple donnée, la liste retournée sera "**[2, 3, 4]**". Il faut remarquer que la méthode traverse le graphe sémantique, ***sans faire aucune évaluation du graphe sémantique***!

- `DescrSemantique::TypeLogiqueDuPremierElementDeListe` qui retourne le type de la variable logique du premier élément d'une liste de descriptions sémantiques, si ce dernier existe.

- `AppelDeFonctionUtilisateur::GrapheSemantiqueDuCorps` qui retourne le graphe sémantique du corps de la fonction.

```
¦ Modification de la déclaration de la classe AppelDeFonctionUtilisa... [FonctionsUtilisateur.h, 0135]

class AppelDeFonctionUtilisateur : public DescrSemantique
   {
public:
   ...
#if defined SYNTHESE_PILUM
   virtual void               Synthetiser (
                                 ContexteDeSynthesePTR leContexteDeSynthese );
#endif /* SYNTHESE_PILUM */

   DescrSemantiquePTR         GrapheSemantiqueDuCorps ();

private:

   FonctionUtilisateurPTR     fFonctionUtilisateur;

   DescrSemantiquePTR         * fArgumentsDAppel;
                                 // un tableau dynamique
   ...
   }; // AppelDeFonctionUtilisateur
```

```
¦ Implantation de AppelDeFonctionUtilisateur::GrapheSemanti... [FonctionsUtilisateur.cc, 0504 à 0508]

DescrSemantiquePTR
AppelDeFonctionUtilisateur::GrapheSemantiqueDuCorps ()
   {
   return fFonctionUtilisateur->GrapheSemantiqueDuCorps();
   }
```

```
¦ Modification de la déclaration de la classe DescrSemantique [DescrSemantique.h, 0053 à 0054]

class DescrSemantique
   {
public:
   ...
   void                       FaireLEvaluationDirecte (
                                 ContexteDEvaluationPTR leContexteDEvaluation );

   VariableLogiqueTypePTR     TypeLogiqueDuPremierElementDeListe ();
   ListeDescrSemantique       TraverserLeGrapheSemantique ();

#if defined SYNTHESE_PILUM
   virtual void               Synthetiser (
                                 ContexteDeSynthesePTR leContexteDeSynthese ) = 0;
   // virtuelle pure
...
```

¦ Implantation de TraverserLeGrapheSemantique et TypeLogiqueDuPre... [DescrSemantique.cc, 0118 à 0225]

```
VariableLogiqueTypePTR
DescrSemantique::TypeLogiqueDuPremierElementDeListe ()
{
   ListeDescrSemantique listeDescr = this->TraverserLeGrapheSemantique();
   if (listeDescr.size() > 0)
      return listeDescr.front()->TypeLogique();
   else
      return gTypeLogiqueInconnu;
}



// Traverser le graphe semantique pour trouver des descriptions semantique d'une liste.
ListeDescrSemantique
DescrSemantique :: TraverserLeGrapheSemantique ()
{
   ValeurListe* liste = dynamic_cast<ValeurListe*>( this );
   if ( liste )
   {
      return liste -> ListeDeDescriptionsSemantique();
   }
   else
   {
      // Les seuls operateurs unaire predefinies qui manipulent des listes et peuvent retourner des
      // valeurs de type liste sont Car et Cdr.
      OperateurUnaire* lOperateurUnaire = dynamic_cast< OperateurUnaire* >( this );
      if ( lOperateurUnaire )
      {
         if (dynamic_cast< Car* >( lOperateurUnaire ) != NULL)
         {
            // Si l'operateur s'agit du Car, on ne s'interesse qu'au premier element de la liste.
            // On remarque que cet element n'est pas toujours une liste, mais que l'on renvoit quand
            // meme une liste de descriptions semantique contenant que cet element.
            ListeDescrSemantique resultat =
                                    lOperateurUnaire -> Operande() -> TraverserLeGrapheSemantique();

            // On supprime tout sauf le premier element.
            ListeDescrSemantique::iterator it = resultat.begin();
            resultat.erase( ++it, resultat.end() );

            // Maintenant qu'on a l'element qu'on veut, il faut verifier qu'il s'agit d'une liste,
            // dans quel cas on retourne les descriptions semantique de cette liste meme,
            // sinon des appels au Car dans les pas recursifs ulterieurs ne marcheront pas (lorsque
            // on remonte dans la recursion).
            if (resultat.front() -> TypeLogique() != gTypeLogiqueListe)
               return resultat;
            else
               return resultat.front() -> TraverserLeGrapheSemantique();
         }
         else if (dynamic_cast< Cdr* >( lOperateurUnaire ) != NULL)
         {
            // Si l'operateur s'agit du Cdr, on ne s'interesse qu'a la queue de la liste. On efface
            // donc le premier element de la liste.
            ListeDescrSemantique resultat =
                                    lOperateurUnaire -> Operande() -> TraverserLeGrapheSemantique();

            resultat.pop_front();
            return resultat;
         }
      }

      // Les seuls operateurs unaire predefinies qui manipulent des listes sont Cons, Append et
      // ConcatListes.
      OperateurBinaire* lOperateurBinaire = dynamic_cast< OperateurBinaire* >( this );
      if ( lOperateurBinaire )
      {
         ListeDescrSemantique resultat =
                              lOperateurBinaire -> OperandeDroit() -> TraverserLeGrapheSemantique();
```

```
         // Dans le cas du Cons, on met l'operande de gauche en tete de la liste.
         if (dynamic_cast< Cons* >( lOperateurBinaire ) != NULL)
            resultat.push_front( lOperateurBinaire -> OperandeDroit() );

         // Dans le cas du Append, on met l'operande de gauche a la fin de la liste.
         else if (dynamic_cast< Append* >( lOperateurBinaire ) != NULL)
            resultat.push_back( lOperateurBinaire -> OperandeDroit() );

         else if (dynamic_cast< ConcatListes* >( lOperateurBinaire ) != NULL)
         {
            ListeDescrSemantique lOperandeGauche =
                           lOperateurBinaire -> OperandeGauche() -> TraverserLeGrapheSemantique();

            // Placer les elements de lOperandeGauche dans la liste resultat, en commencant de la
            // fin.
            for ( ListeDescrSemantique::reverse_iterator it = lOperandeGauche.rbegin();
                  it != lOperandeGauche.rend();
                  it ++ )
                     resultat.push_front( *it );
         }

         // Retourner le resultat.
         return resultat;
      }

      // Pour des situations tel que "L = [1, 2, 3]; X = Car(L);", il faut d'abord obtenir le
      // graphe semantique de L, puis le parcourir pour trouver le type logique voulu.
      AppelDeFonctionUtilisateur *lAppelDeFonctionUtil =
                                             dynamic_cast< AppelDeFonctionUtilisateur* >( this );
      if ( lAppelDeFonctionUtil )
         return lAppelDeFonctionUtil -> GrapheSemantiqueDuCorps() -> TraverserLeGrapheSemantique();

      // Pour des constructions comme "f(L) = Car(L);", le premier element de la liste L est inconnu
      // a priori. On ne fait rien ... (PS. La condition est ajoutee purement comme un rappel)
      if (dynamic_cast< EmploiParametrePTR >( this )) ;

      // Retourner une liste de descriptions semantique vide.
      return ListeDescrSemantique();
   }
}
```

Ensuite j'ajoute le nouveau genre de fonction `kFonctionListe`

```
¦ Modification de l'enumération GenreFonctionPredefinie [TableDesSymboles.h, 0176]

enum GenreFonctionPredefinie
   {
   kSi,                       kSequence,                kIterateur,

   kFonctionListe,

   kAutreFonctionPredefinie
   };
```

Une raison pour laquelle on a différents genres de fonctions est que pour certaines fonctions, les types logique des paramètres définis au niveau de la sémantique peuvent différés de ceux déterminés lors de l'appel de la fonction.

Prenons l'exemple du **`Cons`**. Si on regarde la définition de sa sémantique, on voit que le type logique de son premier argument n'est pas précisé puisque ce paramètre peut être de type quelconque. Donc, à l'appel de cette fonction, il faut déterminer le type de ce paramètre, i.e. de passer du type non-précisé à un type reconnu par Lista. Le genre `kFonctionListe` décrit l'ensemble comprenant les fonctions **`Cons`** et **`Append`** où le type du premier argument doit être déterminer à l'appel de la fonction.

Ajouter un nouveau genre de fonction implique l'ajout d'une méthode qui traite ce genre. La classe `AnalyseurListaPredictifDescenteRecursive` est alors modifiée pour pouvoir traiter ce nouveau genre de fonction. J'ajoute la méthode `AnalyseurListaPredictifDescente-Recursive::InstrListe` qui crée le graphe sémantique avec les bons types lors de l'appel de la fonction.

```
¦ Modification de la déclaration de la classe AnalyseurListaPredictifDescenteRecursive
¦ [Semantique/AnalyseurPredictifDescenteRecursiveLista.h, 0115 à 0116]

class AnalyseurListaPredictifDescenteRecursive : public AnalyseurSemantiqueLista
   {
public:
   ...
   DescrSemantiquePTR         InstrSequence (
                                 FonctionPredefiniePTR laFonctionPredefinie );

   DescrSemantiquePTR         InstrIteration (
                                 FonctionPredefiniePTR laFonctionPredefinie );

   DescrSemantiquePTR         InstrListe (
                                 FonctionPredefiniePTR laFonctionPredefinie );

   DescrSemantiquePTR         AppelFonctionUtilisateur (
                                 FonctionUtilisateurPTR laFonctionUtilisateur );
   ...
   }; // AnalyseurListaPredictifDescenteRecursive
```

```
¦ Implantation de InstrListe [Semantique/AnalyseurPredictifDescenteRecursiveLista.cc, 1343 à 1377]

DescrSemantiquePTR
AnalyseurListaPredictifDescenteRecursive::InstrListe( FonctionPredefiniePTR laFonction )
{
   // Les seules fonctions qui concernent les listes et qui soient traitees de maniere speciale
   // sont Cons et Append, i.e. leur premier argument peut etre de type quelconque.
   // L'idee est donc de ne pas mettre de contraintes sur celui-ci mais de quand-meme verifier que
   // le deuxieme argument est bien une liste.

   // Aucune contrainte sur le premier argument, on l'ecrit tout de suite dans le bloc d'arguments.
   DescrSemantiquePTR lOperandeGauche = Expression();

   // Un virgule doit suivre l'argument (puisqu'on est dans le cas d'un operateur binaire).
   TesterTerminal(   t_VIRGULE,
               MiseEnForme(   gLANGUE_Syntaxe -> FormatVirguleAttendueApresUnArgumentDAppel(),
                              1, laFonction -> Nom()
                           )
            );

   // Verifier les contraintes definies pour l'operande droit.
   DescrSemantiquePTR lOperandeDroit = Expression();
   VariableLogiqueTypePTR typeLogiqueOperandeDroit = laFonction -> TypesLogiquesDesParametres()[1];

   TesterLeTypeAttendu (
      typeLogiqueOperandeDroit -> ValeurDeLiaison (),
      lOperandeDroit -> TypeLogique (),
      gLANGUE_Syntaxe -> Expression ()
   );

   DescrSemantiquePTR* leBlocDArguments = new DescrSemantiquePTR[2];
   leBlocDArguments[0] = lOperandeGauche;
   leBlocDArguments[1] = lOperandeDroit;

   // Retourner la description semantique.
   return laFonction->CreerGrapheDAppelALaFonction( fGenreLectureAuClavier, leBlocDArguments );
}
```

Lorsque le genre `kFonctionListe` ainsi que la méthode `AnalyseurListaPredictifDescenteRecursive::InstrListe` sont crées, je modifie la méthode `AnalyseurListaPredictifDescenteRecursive::AppelFonctionPredefinie` qui crée le lien entre les deux.

```
¦ Modification de la methode AppelFoncti... [AnalyseurPredictifDescenteRecursiveLista.cc, 1018 à 1020]

DescrSemantiquePTR
AnalyseurListaPredictifDescenteRecursive :: AppelFonctionPredefinie (
   FonctionPredefiniePTR laFonctionPredefinie )
   {
   // IDENT a ete accepte

   GenreFonctionPredefinie
            leGenreFonctionPredefinie     =
               laFonctionPredefinie -> LeGenreFonctionPredefinie ();
   Chaine   leNomDeLaFonctionPredefinie   =
               laFonctionPredefinie -> Nom ();

   TesterTerminal (
      t_PAR_GAUCHE,
      MiseEnForme (
         gLANGUE_Syntaxe ->
            FormatParentheseAttendueAvantUnAppelDeFonctionPredefinie (),
         leNomDeLaFonctionPredefinie ));

   DescrSemantiquePTR   res;

   switch (leGenreFonctionPredefinie)
      {
      case kAutreFonctionPredefinie:
         res = AppelAutreFonctionPredefinie (laFonctionPredefinie);
         break;
      ...

      case kFonctionListe:
         res = InstrListe (laFonctionPredefinie);
         break;

      case kIterateur:
         res = InstrIteration (laFonctionPredefinie);
         break;
      }  // switch

   TesterTerminal (
      t_PAR_DROITE,
      MiseEnForme (
         gLANGUE_Syntaxe ->
            FormatParentheseAttendueApresUnAppelDeFonction (),
         leNomDeLaFonctionPredefinie ));

   return res;
   }  // AnalyseurListaPredictifDescenteRecursive :: AppelFonctionPredefinie
```

Je peux enfin passer à l'instauration de nouveaux noeuds sémantiques. Pour cela, deux nouveaux fichiers **`Listes.h`** et **`Listes.cc`** sont crées; et **`makefileSemantique`** est modifié pour en tenir compte de ces derniers. Dans ces deux fichiers, je définis des nouvelles sous-classes de `FonctionPredefinie`, qui ont pour but, parmi d'autres, de créer le graphe d'appel de leur fonction correspondante à partir d'une opérande donnée et du noeud sémantique défini de la fonction

¦ Déclaration de nouvelles fonctions prédéfinies [Semantique/Listes.h]

```
// ============================================================================
// Listes.h
// Jeremy OTHIENO
// ============================================================================

#ifndef __Listes__
#define __Listes__

#include "OperateursNAires.h"
#include "TableDesSymboles.h"
#include "ContexteDEvaluation.h"

// ----------------------------------------------------------------------------
// Manipulation des listes
// ----------------------------------------------------------------------------

class CarPredef : public FonctionPredefinie
{
public:
                                CarPredef( ChaineLanguePredefinisPFM );

   virtual DescrSemantiquePTR   CreerGrapheDAppelALaFonction
                                (
                                   GenreLectureAuClavier   leGenreLectureAuClavier,
                                   DescrSemantiquePTR*     leBlocDArguments
                                );
};

// ----------------------------------------------------------------------------

class CdrPredef : public FonctionPredefinie
{
public:
                                CdrPredef( ChaineLanguePredefinisPFM );

   virtual DescrSemantiquePTR   CreerGrapheDAppelALaFonction
                                (
                                   GenreLectureAuClavier   leGenreLectureAuClavier,
                                   DescrSemantiquePTR*     leBlocDArguments
                                );
};

// ----------------------------------------------------------------------------

class ConsPredef : public FonctionPredefinie
{
public:
                                ConsPredef( ChaineLanguePredefinisPFM );

   virtual DescrSemantiquePTR   CreerGrapheDAppelALaFonction
                                (
                                   GenreLectureAuClavier   leGenreLectureAuClavier,
                                   DescrSemantiquePTR*     leBlocDArguments
                                );
};

// ----------------------------------------------------------------------------

class AppendPredef : public FonctionPredefinie
{
public:
                                AppendPredef( ChaineLanguePredefinisPFM );

   virtual DescrSemantiquePTR   CreerGrapheDAppelALaFonction
                                (
                                   GenreLectureAuClavier   leGenreLectureAuClavier,
                                   DescrSemantiquePTR*     leBlocDArguments
                                );
};
```

```
// ---------------------------------------------------------------------------

class ConcatListesPredef : public FonctionPredefinie
{
public:
                                ConcatListesPredef( ChaineLanguePredefinisPFM );

   virtual DescrSemantiquePTR   CreerGrapheDAppelALaFonction
                                (
                                   GenreLectureAuClavier   leGenreLectureAuClavier,
                                   DescrSemantiquePTR*     leBlocDArguments
                                );
};

// ---------------------------------------------------------------------------

class LongueurListePredef : public FonctionPredefinie
{
public:
                                LongueurListePredef( ChaineLanguePredefinisPFM );

   virtual DescrSemantiquePTR   CreerGrapheDAppelALaFonction
                                (
                                   GenreLectureAuClavier   leGenreLectureAuClavier,
                                   DescrSemantiquePTR*     leBlocDArguments
                                );
};

// ---------------------------------------------------------------------------
// Operateurs unaires
// ---------------------------------------------------------------------------

class Car : public OperateurUnaire
{
public:
                            Car( Chaine, DescrSemantiquePTR, VariableLogiqueTypePTR );
   virtual ValeurLista      Evaluer( ContexteDEvaluationPTR );

#if defined SYNTHESE_PILUM
   virtual void             Synthetiser( ContexteDeSynthesePTR );
#endif /* SYNTHESE_PILUM */
};

// ---------------------------------------------------------------------------

class Cdr : public OperateurUnaire
{
public:
                            Cdr( Chaine, DescrSemantiquePTR );
   virtual ValeurLista      Evaluer( ContexteDEvaluationPTR );

#if defined SYNTHESE_PILUM
   virtual void             Synthetiser( ContexteDeSynthesePTR );
#endif /* SYNTHESE_PILUM */
};

// ---------------------------------------------------------------------------

class LongueurListe : public OperateurUnaire
{
public:
                            LongueurListe( Chaine, DescrSemantiquePTR );
   virtual ValeurLista      Evaluer( ContexteDEvaluationPTR );

#if defined SYNTHESE_PILUM
   virtual void             Synthetiser( ContexteDeSynthesePTR );
#endif /* SYNTHESE_PILUM */
};
```

```
// ------------------------------------------------------------------------
// Operateurs binaires
// ------------------------------------------------------------------------

class ConcatListes : public OperateurBinaire
{
public:
                              ConcatListes(  Chaine,
                                             DescrSemantiquePTR, DescrSemantiquePTR
                                          );
   virtual ValeurLista        Evaluer( ContexteDEvaluationPTR );

#if defined SYNTHESE_PILUM
   virtual void               Synthetiser( ContexteDeSynthesePTR );
#endif /* SYNTHESE_PILUM */
};

// ------------------------------------------------------------------------

class Cons : public OperateurBinaire
{
public:
                              Cons( Chaine,
                                    DescrSemantiquePTR, DescrSemantiquePTR
                                  );
   virtual ValeurLista        Evaluer( ContexteDEvaluationPTR );

#if defined SYNTHESE_PILUM
   virtual void               Synthetiser( ContexteDeSynthesePTR );
#endif /* SYNTHESE_PILUM */
};

// ------------------------------------------------------------------------

class Append : public OperateurBinaire
{
public:
                              Append(  Chaine,
                                       DescrSemantiquePTR, DescrSemantiquePTR
                                     );
   virtual ValeurLista        Evaluer( ContexteDEvaluationPTR );

#if defined SYNTHESE_PILUM
   virtual void               Synthetiser( ContexteDeSynthesePTR );
#endif /* SYNTHESE_PILUM */
};

#endif /* __Listes__ */
```

L'implantation complète se trouve dans le fichier **`Semantique/Listes.cc`**.

La description des identificateurs s'appuie sur la description des niveaux de déclarations. Un niveau de déclaration est décrit par **une table d'identificateurs déclarés dans ce niveau**, que nous appelons **dictionnaire**. **Le dictionnaire des identificateurs prédéfinis est créé et empilé le premier** dans la pile des dictionnaires. Il faut ainsi ajouter les nouveaux identificateurs au dictionnaire principale

```
¦ Modification de AnalyseurSemantiqueLista::InsererLesIdentificateurs
¦ [Semantique/AnalyseurSemantiqueLista.cc, 0314 à 0329]

void
AnalyseurSemantiqueLista :: InsererLesIdentificateursPredefinis ()
   {
   InsererLIdentificateurPredefini (
      new
         VraiPredef (& LanguePredefinis :: Vrai) );
   ...
```

```
   InsererLIdentificateurPredefini (
      new
         ProduitPredef (& LanguePredefinis :: Produit) );
   InsererLIdentificateurPredefini (
      new
         PourPredef (& LanguePredefinis :: Pour) );
   InsererLIdentificateurPredefini (
      new
         CarPredef (& LanguePredefinis :: Car) );
   InsererLIdentificateurPredefini (
      new
         CdrPredef (& LanguePredefinis :: Cdr) );
   InsererLIdentificateurPredefini (
      new
         ConsPredef (& LanguePredefinis :: Cons) );
   InsererLIdentificateurPredefini (
      new
         AppendPredef (& LanguePredefinis :: Append) );
   InsererLIdentificateurPredefini (
      new
         ConcatListesPredef (& LanguePredefinis :: ConcatListes) );
   InsererLIdentificateurPredefini (
      new
         LongueurListePredef (& LanguePredefinis :: LongueurListe) );
   }  //       AnalyseurSemantiqueLista :: InsererLesIdentificateursPredefinis
```

III.1.e. Traces d'exécution

```
COMMANDE: ./ListaPredictifDescenteRecursive -ms -mi -md -mg

Contenu du fichier source '../exemples/ListesFonctions.lista':
-------------------------------------------------------------
A = [1, 2, 3];
B = [1, 2, "trois", 2*2, [Vrai], A];
C = Car(A);
D = Cdr(B);
F = ConcatListes(B, B);
G = LongueurListe(B);
H = Cons(2, A);
I = Append(H, B);
-------------------------------------------------------------

[VariableLogique_9 "fonction 'A'" -> TypeLogiqueLIBRE] decrit le type de la fonction 'A'


--> On lie la variable logique libre [VariableLogique_9 "fonction 'A'" -> TypeLogiqueLIBRE] a la valeur TypeListe


Graphe semantique du corps de A :
        [
                1.000000
                2.000000
                3.000000
        ]
----------------
[VariableLogique_10 "fonction 'B'" -> TypeLogiqueLIBRE] decrit le type de la fonction 'B'

--> La variable logique [VariableLogique_3 "TypeNombre" -> TypeNombre] est deja liee a la valeur TypeNombre
--> La variable logique [VariableLogique_3 "TypeNombre" -> TypeNombre] est deja liee a la valeur TypeNombre

--> On lie la variable logique libre [VariableLogique_10 "fonction 'B'" -> TypeLogiqueLIBRE] a la valeur TypeListe


Graphe semantique du corps de B :
        [
                1.000000
                2.000000
                "trois"
                *
                        2.000000
                        2.000000
                [
                        Vrai
```

```
                ]
                Fonction 'A', declaree (eval:0,s:0), appelee (eval: 1, stat: 1)
        ]
-----------------
[VariableLogique_11 "fonction 'C'" -> TypeLogiqueLIBRE] decrit le type de la fonction 'C'


--> On lie la variable logique libre [VariableLogique_11 "fonction 'C'" -> TypeLogiqueLIBRE] a la valeur TypeNombre


Graphe semantique du corps de C :
        Car
                Fonction 'A', declaree (eval:0,s:0), appelee (eval: 1, stat: 1)
-----------------
[VariableLogique_12 "fonction 'D'" -> TypeLogiqueLIBRE] decrit le type de la fonction 'D'


--> On lie la variable logique libre [VariableLogique_12 "fonction 'D'" -> TypeLogiqueLIBRE] a la valeur TypeListe


Graphe semantique du corps de D :
        Cdr
                Fonction 'B', declaree (eval:0,s:0), appelee (eval: 1, stat: 1)
-----------------
[VariableLogique_13 "fonction 'F'" -> TypeLogiqueLIBRE] decrit le type de la fonction 'F'


--> On lie la variable logique libre [VariableLogique_13 "fonction 'F'" -> TypeLogiqueLIBRE] a la valeur TypeListe


Graphe semantique du corps de F :
        ConcatListes
                Fonction 'B', declaree (eval:0,s:0), appelee (eval: 1, stat: 1)
                Fonction 'B', declaree (eval:0,s:0), appelee (eval: 1, stat: 1)
-----------------
[VariableLogique_14 "fonction 'G'" -> TypeLogiqueLIBRE] decrit le type de la fonction 'G'


--> On lie la variable logique libre [VariableLogique_14 "fonction 'G'" -> TypeLogiqueLIBRE] a la valeur TypeNombre


Graphe semantique du corps de G :
        LongueurListe
                Fonction 'B', declaree (eval:0,s:0), appelee (eval: 1, stat: 1)
-----------------
[VariableLogique_15 "fonction 'H'" -> TypeLogiqueLIBRE] decrit le type de la fonction 'H'


--> On lie la variable logique libre [VariableLogique_15 "fonction 'H'" -> TypeLogiqueLIBRE] a la valeur TypeListe


Graphe semantique du corps de H :
        Cons
                2.000000
                Fonction 'A', declaree (eval:0,s:0), appelee (eval: 1, stat: 1)
-----------------
[VariableLogique_16 "fonction 'I'" -> TypeLogiqueLIBRE] decrit le type de la fonction 'I'


--> On lie la variable logique libre [VariableLogique_16 "fonction 'I'" -> TypeLogiqueLIBRE] a la valeur TypeListe


Graphe semantique du corps de I :
        Append
                Fonction 'H', declaree (eval:0,s:0), appelee (eval: 1, stat: 1)
                Fonction 'B', declaree (eval:0,s:0), appelee (eval: 1, stat: 1)
-----------------
*** Code Lista lexicalement   correct   ***
*** Code Lista syntaxiquement correct   ***
*** Code Lista semantiquement correct   ***

On purge le dictionnaire 'Identificateurs predefinis', contenant:
        'Append', Fonction predefinie:
                (TypeNonPrecise, TypeListe) -> TypeListe
        'ArcTan', Fonction predefinie:
                (TypeNombre) -> TypeNombre
        'Car', Fonction predefinie:
                (TypeListe) -> TypeNonPrecise
        'Cdr', Fonction predefinie:
                (TypeListe) -> TypeListe
```

```
        'ChaineVide', Fonction predefinie:
                (TypeChaine) -> TypeBooleen
        'ConcatChaines', Fonction predefinie:
                (TypeChaine, TypeChaine) -> TypeChaine
        'ConcatListes', Fonction predefinie:
                (TypeListe, TypeListe) -> TypeListe
        'Cons', Fonction predefinie:
                (TypeNonPrecise, TypeListe) -> TypeListe
        'ContientChaine', Fonction predefinie:
                (TypeChaine, TypeChaine) -> TypeBooleen
        'Cos', Fonction predefinie:
                (TypeNombre) -> TypeNombre
        'Different', Fonction predefinie:
                (TypeNombre, TypeNombre) -> TypeBooleen
        'E', Constante predefinie:
                TypeNombre

                                        REMARQUE DE L'AUTEUR: SORTIE TRONQUÉE

        'LireNombre', Fonction predefinie:
                (TypeChaine) -> TypeNombre
        'Log', Fonction predefinie:
                (TypeNombre) -> TypeNombre
        'LongueurChaine', Fonction predefinie:
                (TypeChaine) -> TypeNombre
        'LongueurListe', Fonction predefinie:
                (TypeListe) -> TypeNombre
        'Modulo', Fonction predefinie:
                (TypeNombre, TypeNombre) -> TypeNombre
        'Non', Fonction predefinie:
                (TypeBooleen) -> TypeBooleen
        'Ou', Fonction predefinie:
                (TypeBooleen, TypeBooleen) -> TypeBooleen

                                        REMARQUE DE L'AUTEUR: SORTIE TRONQUÉE

*** Le fichier de code objet Pilum "../exemples/ListesFonctions.valeur_PDR" a ete créé ***

*** Temps ecoule: 0 seconde(s) ***
```

C'est tout pour l'analyse sémantique! Le langage accepte **sémantiquement** une liste de nombres et une liste de valeurs différentes, ainsi que de nouvelles fonctions prédéfinies servant à manipuler ces listes. En résumé, la nouvelle hiérarchie des classes des graphes sémantiques Lista ressemble à

```
DescrSemantique
        OperateurZeroaire
                Hasard, ...
        OperateurUnaire
                Non, Pair, ..., Car, Cdr, LongueurListe
        OperateurBinaire
                Et, Ou, Seq, ..., Cons, Append, ConcatListes
        OperateurTernaire
                FormaterNombre, RemplacerChaine
        ValeurInconnue
        ValeurNombre
        ValeurLogique
        ValeurChaine
        ValeurVide
        ValeurListe
        EmploiParametre
                EmploiParametreParValeur
                EmploiParametreParNom
                EmploiParametreParBesoin
        AppelDeFonctionUtilisateur
        Si
        EmploiIndiceIteration
        Iteration
                Somme, Produit, Pour
```

III.2. Flex et Bison

Je commence par ajouter de nouveaux terminaux au grammaire, comme il a été fait pour l'analyse syntaxique

```
¦ Ajout de nouveaux tokens [Semantique/Semantique.Bison, 0081]

%token                          t_PLUS              t_MOINS
%token                          t_FOIS              t_DIVISE
%token                          t_CONCAT

%token                          t_POINT_VIRGULE     t_INTERROGE

%token                          t_CROCHET_DROIT     t_CROCHET_GAUCHE
```

```
¦ Ajout des nouveaux terminaux, sous forme de chaine [Semantique.Flex, 0328 à 0337]

"["                             {
                                AppondreAuSourceCumule ();

                                return EnregistrerTerminal (t_CROCHET_GAUCHE);
                                }
"]"                             {
                                AppondreAuSourceCumule ();

                                return EnregistrerTerminal (t_CROCHET_DROIT);
                                }
```

Comme toute à l'heure, il nous faut un moyen de connaître les terminaux sous leur forme textuelle; ainsi, je modifie la méthode `TerminalSousFormeTextuelle`. Une fois modifiée, j'ajoute des méthodes à l'analyseur Lista en Flex et Bison qui permettent de construire des listes.

```
¦ Ajout de nouveaux tokens [Semantique/SupportFlexBisonLista.cc, 0138 à 0141]

Chaine
TerminalSousFormeTextuelle (TerminalFlexBison leTerminalFlexBison)
   {
   switch (leTerminalFlexBison)
      ...
      case t_CROCHET_GAUCHE:
         return "          [";
      case t_CROCHET_DROIT:
         return "          ]";
      ...
      }  // switch
   }  // TerminalSousFormeTextuelle
```

```
¦ Modification de la déclaration de la classe AnalyseurListaFlexBison
¦ [Semantique/AnalyseurListaFlexBison.h, 0116 à 0118 et 0137]

class AnalyseurListaFlexBison : public AnalyseurSemantiqueLista
   {
public:
   ...
   void                         TraiterDebutListe ();
   void                         AjouterElementALaListe (DescrSemantiquePTR);
   ValeurListe*                 TraiterFinListe ();
   ...
private:
   ...
   std::deque<ListeDescrSemantique*>   fPileDesListes;

  }; // AnalyseurListaFlexBison
```

```
¦ Implantation des nouvelles méthodes [Semantique/AnalyseurListaFlexBison.h, 1101 à 1143]
void
AnalyseurListaFlexBison::TraiterDebutListe()
{
   // Creer une nouvelle liste sur la pile des listes.
   fPileDesListes.push_front( new ListeDescrSemantique() );
}

void
AnalyseurListaFlexBison::AjouterElementALaListe( DescrSemantiquePTR description )
{
   // Ajouter l'element a la derniere liste de descriptions semantiques empilee.
   fPileDesListes.front()->push_back( description );
}

ValeurListe*
AnalyseurListaFlexBison::TraiterFinListe()
{
   // Obtenir la liste de descriptions semantiques, et le type logique de la liste.
   ListeDescrSemantique* descriptions = fPileDesListes.front();

   // Creer le graphe semantique de la liste.
   ValeurListe* resultat = new ValeurListe( *descriptions, leTypeLogique );

   // Une fois qu'une liste est retournee, on n'a plus besoin de le garder sur la pile.
   fPileDesListes.pop_front();

   return resultat;
}
```

Je modifie l'analyseur sémantique. Vous constatez qu'elle ressemble beaucoup à l'analyseur syntaxique Flex et Bison, la seule différence est qu'en plus de la vérification de la forme des listes, le contenu de celles-ci est stockée pour être utilisé ultérieurement.

```
¦ Ajout de nouveaux tokens [Semantique/Semantique.Bison, 0102]

%type    <fGrapheSemantique>        Sequence
%type    <fGrapheSemantique>        SequenceDExpressions

%type    <fGrapheSemantique>        AppelDeFonction

%type    <fGrapheSemantique>        Liste
...

¦ Modification du non-terminal Expression [Semantique/Semantique.Bison, 0347 à 0352]

Expression
   :  t_MOINS
   ...

   |  Terme

   |  Liste
      { $$ = $1; }
   ;

¦ Déclaration de nouveaux non-terminaux [Semantique/Semantique.Bison, 0639 à 0675]

DelimiteurListeGauche
   :  t_CROCHET_GAUCHE
      {
         gAnalyseurListaFlexBison->TraiterDebutListe();
      }
   ;
```

```
DelimiteurListeDroit
   :  t_CROCHET_DROIT
   |  error
      {
         std::cout << "--> ']' attendu a la fin d'une liste!" << std::endl;
      }
   ;

ElementsListe
   :  Expression
      {
         gAnalyseurListaFlexBison->AjouterElementALaListe( $1 );
      }
   |  Expression
      {
         gAnalyseurListaFlexBison->AjouterElementALaListe( $1 );
      }
      t_VIRGULE ElementsListe
   ;

Liste
   :  DelimiteurListeGauche DelimiteurListeDroit
      {
         $$ = gAnalyseurListaFlexBison->TraiterFinListe();
      }
   |  DelimiteurListeGauche ElementsListe DelimiteurListeDroit
      {
         $$ = gAnalyseurListaFlexBison->TraiterFinListe();
      }
   ;
```

III.2.a. Traces d'exécution

```
COMMANDE: ./ListaFlexBison -ms -mc

Contenu du fichier source '../exemples/ListesFonctions.lista':
-----------------------------------------------------------------
A = [1, 2, 3];
B = [1, 2, "trois", 2*2, [Vrai], A];
C = Car(A);
D = Cdr(B);
F = ConcatListes(B, B);
G = LongueurListe(B);
H = Cons(2, A);
I = Append(H, B);
-----------------------------------------------------------------

*** Code Lista lexicalement   correct   ***
*** Code Lista syntaxiquement correct   ***
*** Code Lista semantiquement correct   ***

Code Pilum synthetisé (49 instructions):
----------------------------------------

   0:          Sauter                          7       // [VariableLogique_2 "suiteDeCorpsDeFonction" ->    7:
// -- Etiquette_2 --]
   1:     1:    // -- Etiquette_1 --
   1:          EmpilerFlottant                         1.000000
   2:          EmpilerFlottant                         2.000000
   3:          EmpilerFlottant                         3.000000
   4:          EmpilerEntier                   3
   5:          GenererListe
   6:          RetourDeFonction                0
   7:     7:    // -- Etiquette_2 --
   7:          Sauter                          21      // [VariableLogique_4 "suiteDeCorpsDeFonction" ->   21:
// -- Etiquette_4 --]
   8:     8:    // -- Etiquette_3 --
   8:          EmpilerFlottant                         1.000000
   9:          EmpilerFlottant                         2.000000
  10:          EmpilerChaine                   "trois"
  11:          EmpilerFlottant                         2.000000
  12:          EmpilerFlottant                         2.000000
```

```
  13:           FoisFlottant
  14:           EmpilerBooleen                  Vrai
  15:           EmpilerEntier                   1
  16:           GenererListe
  17:           Appel                           1       // [VariableLogique_1 "corpsDeFonction" ->    1:    // --
Etiquette_1 --]
  18:           EmpilerEntier                   6
  19:           GenererListe


                                        REMARQUE DE L'AUTEUR: SORTIE TRONQUÉE

  43:    43:    // -- Etiquette_14 --
  43:           Sauter                          48      // [VariableLogique_16 "suiteDeCorpsDeFonction" ->   48:
// -- Etiquette_16 --]
  44:    44:    // -- Etiquette_15 --
  44:           Appel                           39      // [VariableLogique_13 "corpsDeFonction" ->   39:    // --
Etiquette_13 --]
  45:           Appel                           8       // [VariableLogique_3 "corpsDeFonction" ->    8:    // --
Etiquette_3 --]
  46:           Append
  47:           RetourDeFonction                0
  48:    48:    // -- Etiquette_16 --
  48:           Halte
----------------------------------------

*** Le fichier de code objet Pilum "../exemples/ListesFonctions.valeur_FB" a ete créé ***

*** Temps ecoule: 0 seconde(s) ***
```

IV. Évaluation directe

Au chapitre précédant, on parle de l'évaluation des expressions en Lista grâce aux graphes sémantiques, sans vraiment implanter ce concept dans le langage: des **valeurs superflues** sont retournées au lieu de valeurs cohérentes. Ce chapitre détaille les étapes à suivre pour rendre possible l'évaluation **correcte** de ces graphes sémantiques.

J'ai choisi de faire l'évaluation directe avec la **méthode prédictive et descente récursive**.

Avant de continuer, il faut noter que le **typage statique de Lista et les contrôles** faits lors de l'analyse sémantique font que les opérations dans les graphes sémantiques manipulent des valeurs dont le type est nécessairement correcte. Il est **inutile** donc de gérer ces types dynamiquement lors de l'évaluation directe.

Les graphes sémantiques sont une forme objet contenant la sémantique des codes sources Lista et sont mieux adaptés à nos besoins que la forme source en texte, puisqu'ils nous permettent de les exécuter par évaluation directe; en d'autres mots, on peut exécuter des programmes Lista sans passer par Pilum. Pour l'évaluation directe, nous utilisons comme support la classe `ValeurLista`

```
¦ Modification de la déclaration de la classe ValeurLista [Semantique/ValeurLista.h]
```

```
class ValeurLista
   {
public:
                              ValeurLista ();

                              ValeurLista (Nombre leNombre);
                              ValeurLista (bool leBooleen);
                              ValeurLista (Chaine laChaine);
                              ValeurLista (Liste laListe);

   //
   // methodes de TEST DYNAMIQUE DE TYPE
   //

   Nombre                     CommeNombre ();
   bool                       CommeBooleen ();
   Chaine                     CommeChaine ();
   Liste                      CommeListe ();

   void                       Ecrire (int lIndentation);

   Nombre                     fNombre;
   bool                       fBooleen;
   Chaine                     fChaine;
   Liste                      fListe;

   TypePTR                    Type ();

private:

   TypePTR                    fType;   // type de la valeur, immutable

   }; // ValeurLista
```

```
¦ Implantation du nouveau constructeur de la classe Valeur... [Semantique/ValeurLista.cc, 0050 à 0054]

ValeurLista :: ValeurLista (Liste laListe)
   {
   fType = gTypeListe;
   fListe = laListe;
   }

...
¦ Implantation de la methode ValeurLista::CommeListe [Semantique/ValeurLista.cc, 0124 à 0131]

Liste
ValeurLista :: CommeListe ()
   {
   if ( dynamic_cast <TypeListe*> (fType) )
      return fListe;

   return Liste();   // superflu
   }  // ValeurLista :: CommeListe ()
```

Ensuite, je modifie la méthode `ValeurLista::Ecrire` qui sert à afficher une valeur de type `ValeurLista` à l'écran

```
¦ Modification de ValeurLista::Ecrire [Semantique/ValeurLista.cc, 0168 à 0169]

void
ValeurLista :: Ecrire (int lIndentation)
   {
   ...
   else if (dynamic_cast <TypeListe *>(fType) != NULL)
      std::cout << MiseEnForme( "%s", fListe.str() );
   ...
```

L'évaluation des graphes sémantiques s'appuie sur des **contextes d'évaluation** passés en paramètres

aux méthodes Evaluer. Un contexte d'évaluation est **une association entre les arguments d'un appel de fonction, repérés par leur numéro d'ordre, et leur valeur**. **Ces associations contiennent la manière d'évaluer le paramètre selon la stratégie d'évaluation**. Il faut noter que seuls les appels aux itérations prédéfinies et aux fonctions utilisateurs en créent de nouveaux, et que les méthodes `Evaluer` ne font qu'utiliser un pointeur sur un contexte d'évaluation reçu en paramètre. On ne se préoccupe pas du contexte lors de l'implantation des méthodes `Evaluer`.

Je modifie toutes les fonctions `Evaluer` présentées au chapitre précédent, pour qu'elles puissent rendre des valeurs cohérentes lors de l'évaluation directe. Comme exemple, voici comment une liste est évaluée

```
¦ Modification de ValeurListe::Evaluer [Semantique/DescrSemantique.cc, 0443 à 0480]

ValeurLista
ValeurListe :: Evaluer (ContexteDEvaluationPTR leContexteDEvaluation)
   {
   ValeurLista          leResultatDeLEvaluation;
   Liste                laListeEvaluee;
   Liste                listeResultat;
   std::list<DescrSemantiquePTR>::iterator it = fListeDeDescriptionsSemantique.begin();

   for (; it != fListeDeDescriptionsSemantique.end(); ++ it)
      {
      // Evaluer le graphe semantique.
      leResultatDeLEvaluation = (*it) -> Evaluer( leContexteDEvaluation );
      TypePTR leType = leResultatDeLEvaluation.Type();

      // Stocker le resultat de l'evaluation dans une liste de valeurs.
      if (leType == gTypeNombre)
         {
         Nombre leNombre = leResultatDeLEvaluation.CommeNombre();
         listeResultat.append( leNombre );
         }
      else if (leType == gTypeBooleen)
         {
         bool leBooleen = leResultatDeLEvaluation.CommeBooleen();
         listeResultat.append( leBooleen );
         }
      else if (leType == gTypeChaine)
         {
         Chaine laChaine = leResultatDeLEvaluation.CommeChaine();
         listeResultat.append( laChaine );
         }
      else if (leType == gTypeListe)
         {
         Liste laListe = leResultatDeLEvaluation.CommeListe();
         listeResultat.append( laListe );
         }
      }
   return ValeurLista( listeResultat );
   }
```

Ce qu'il faut comprendre est que la première étape est d'évaluer le graphe sémantique (les variables `fOperande`, `fOperandeGauche` ou `fOperandeDroit`) dans un contexte donné pour obtenir une nouvelle valeur de type `ValeurLista`. Le type `ValeurLista` est comme un ensemble de valeurs possibles dans le langage Lista. On fait un test dynamique de type avec une des méthodes `CommeNombre`, `CommeBooleen`, `CommeChaine` ou `CommeListeNombre`, pour produire une valeur sous son type exacte, soit `Nombre`, `Booleen`, `Chaine` ou `Liste`, respectivement. Une fois que cette valeur est obtenue, un traitement est fait sur celle-ci par une des méthodes définie dans le langage d'implantation. Les évaluations directes des fonctions prédéfinies sont implantées dans **`Semantique/Listes.cc`**.

IV.a. Traces d'exécution

```
COMMANDE: ./ListaPredictifDescenteRecursive -ms -es

Contenu du fichier source 'debugcode/syntaxe.lista':
------------------------------------------------------------
A = [1, 2, 3];
B = [1, 2, "trois", 2*2, [Vrai], A];


C = Car(A);
D = Cdr(B);
F = ConcatListes(B, B);
G = LongueurListe(B);
H = Cons(2, A);
I = Append(H, B);
J = Car(Car([["Grrrr...", 1, 2, 3], 0]));
K(x, y, z) = Append( x, Cons( y*x, Append( z+3, A ) ) );

? A;
? B;
? C;
? K(10, 20, 30);
------------------------------------------------------------

Debut de l'evaluation directe...
Valeur:
[1.000000, 2.000000, 3.000000]
----------------

*** Temps d'evaluation directe: 0 seconde(s) ***

Debut de l'evaluation directe...
Valeur:
[1.000000, 2.000000, "trois", 4.000000, [Vrai], [1.000000, 2.000000, 3.000000]]
----------------

*** Temps d'evaluation directe: 0 seconde(s) ***

Debut de l'evaluation directe...
Valeur:
        1.000000
----------------

*** Temps d'evaluation directe: 0 seconde(s) ***

Debut de l'evaluation directe...
Valeur:
[200.000000, 1.000000, 2.000000, 3.000000, 33.000000, 10.000000]
----------------

*** Temps d'evaluation directe: 0 seconde(s) ***

*** Code Lista lexicalement   correct   ***
*** Code Lista syntaxiquement correct   ***
*** Code Lista semantiquement correct   ***
```

IV.1. Le comportement des fonctions

Afin de comprendre les résultats des nouvelles fonctions prédéfinies après leur évaluation, il est nécessaire que je définisse leurs comportements spéciaux, les voici

- La fonction **`car`** produit une erreur sémantique si une liste vide est fournie comme argument d'appel.
- Similairement, la fonction **`cdr`** produit une erreur sémantique si son argument d'appel est une liste vide.
- Les fonctions **`cons`**, **`append`** et **`concat`** ne modifient aucun de leurs arguments d'appel. Par exemple, lorsqu'on concatène une liste à une autre, une nouvelle liste qui regroupe les éléments des deux liste est crée.

- La fonction **`concat`** concatène la liste donnée comme deuxième argument à la liste donnée en premier argument, e.g. ConcatListes([1, 2, 3], [4, 5]) retourne la liste ***[1, 2, 3, 4, 5]*** tandis que ConcatListes([4, 5], [1, 2, 3]) retourne la liste ***[4, 5, 1, 2, 3]***.

IV.2. Les stratégies d'évaluation[3].

On appelle **stratégie** d'enchaînement d'opérations la manière de choisir quelle opération est effectuée à chaque étape. Les différentes modes de passages de paramètres conduisent à des stratégies distinctes, qui ne sont pas équivalentes

- la stratégie "**passage par valeur**" pour l'évaluation des appels de fonction est **incomplète**: elle peut ne jamais se terminer, bien que l'expression soit calculable. Cela se traduit par ce qu'on appelle usuellement une ***recursion infinie***.

- la stratégie "**passage par nom**" est **complète**: elle garantit de trouver en un temps fini la valeur de l'expression à évaluer si elle est calculable.

- la stratégie "**passage par besoin**" (ou **l'évaluation parreseuse**) est une optimisation intéressante du passage par nom qui ne souffre pas de l'incomplétude du passage par valeur. Cette stratégie consiste à passer à la fonction appelée les arguments non évalués, comme dans le passage par nom, pour garantir l'obtention de la valeur de l'expression si elle est calculable. De plus, la valeur du paramètre n'est évaluée **que la première fois qu'elle est nécessaire, en la mémorisant pour les besoins ultérieures**, d'où le nom "*évaluation parresseuse*". L'intérêt de cette type de stratégie est qu'elle permet de **manipuler des structures de données infinies**.

Des exemples d'emploi de différents stratégies se trouvent dans **`exemples/ListesStrategie.lista`**.

3 Pour plus d'information, Polycopié Langages de programmation (EPFL 1992), page 55.

V. Pilum

La machine Pilum est principalement **une machine virtuelle à pile**. Les valeurs manipulées par Pilum peuvent être de différents types de **valeurs pures** ainsi que des **adresses** dans le code ou dans la pile d'exécution.

Les types manipulés sont `AdresseDansLeCode`, `AdresseDansLaPile`, `AccesStatique` et `TypeValeur-Pilum`. Le type `ValeurPilum` est l'unité d'encombrement des informations dans la pile de la machine Pilum.

Les codes opératoires des instructions de la machine Pilum sont décrites par le type énuméré **CodeOpPilum**. Les instructions, quant à elles, sont décrites par le type **InstructionPilum**. Les états d'exécution de la machine Pilum sont décrites par le type **EtatPilum**.

Le synthétiseur de code Pilum est implanté par la classe `SynthetiseurPilum`. Les méthodes comme `Commentaire`, `Zeroadique`, `Entier`, `Saut`, `AccesCellulePourValeur`, `AccesCellulePourAdresse`, `AccesCellulePourLienStatique`, et `EvaluerThunk` sont chargées de synthétiser une instruction du type correspondant.

Un **bloc d'activation** est un ensemble d'informations groupées en mémoire décrivant une instance, un appel particulier à une fonction. Comme on sort des appels de fonctions dans l'ordre inverse de celui où on y est entré, on utilise une pile des blocs d'activation, également appelée ***pile d'exécution***.

V.1. Extension de la machine Pilum

Le but de cette partie du chapitre est de detailler comment **étendre le jeu d'instructions ainsi que la structure du début du code objet** de la machine Pilum pour la rendre apte à manipuler des listes de nombres.
Premièrement, on a besoin d'**ajouter deux types qui représentent une liste nombres (homogène) et une liste hétérogène, à l'ensemble de types manipulés** par la machine Pilum. Comme la seconde extension s'agit d'une liste hétérogène, il est aberrant de créer deux types pour distinguer une liste de nombres et une liste hétérogène. On commence par la modification de l'énumé-ration `TypeValeurPilum` qui décrit l'ensemble de valeurs de la machine Pilum

```
¦ Modification de l'énumération TypeValeurPilum [Pilum/Pilum.h, 0051]

enum TypeValeurPilum
      {
      kValeurInconnue,
      ...
      kEntier,                  kFlottant,                kBooleen,
      kCaractere,               kChaine,
      kListe,

      kMarqueBlocDActivationDeFonction,
      kMarqueBlocDActivationDeThunk

      };
```

Ensuite, on modifie la méthode `ValeurPilum::TypeSousFormeDeChaine` pour la rendre capable de

retourner notre type sous la forme d'une chaîne de caractères

```
¦ Modification de ValeurPilum::TypeSousFormeDeChaine [Pilum/Pilum.cc, 0071 à 0072]

Chaine
ValeurPilum :: TypeSousFormeDeChaine ()
   {
   switch (fTypeValeurPilum)
      {
      case kValeurInconnue:
         return gLANGUE_Pilum -> ValeurInconnue ();
      ...
      case kCaractere:
         return gLANGUE_Pilum -> Caractere ();
      case kChaine:
         return gLANGUE_Pilum -> Chaine_ ();

      case kMarqueBlocDActivationDeFonction:
         return gLANGUE_Pilum -> MarqueBlocDActivationDeFonction ();
      case kMarqueBlocDActivationDeThunk:
         return gLANGUE_Pilum -> MarqueBlocDActivationDeThunk ();

      case kListe:
         return "TypeListe";
      ...
      }
   }
```

Pour que la machine Pilum puisse stocker une liste de valeurs sur la pile d'exécution, il faut que la structure `ValeurPilum`[4] contienne une structure capable de stocker cette liste.

Pour cela, il faut modifier l'union anonyme[5]; mais avant sa modification de l'union, il faut définir le type `ListePTR` puisque les objets, tels que les classes, ne peuvent pas faire partir des champs d'une union! Ce type a été défini lorsque je construisais ma structure de liste dans le langage d'implantation, cf. voir le fichier **`Types.h`**. Je modifie la structure `ValeurPilum`

```
¦ Modification de la déclaration du struct ValeurPilum [Pilum/Pilum.h, 0073 et 0088]

struct ValeurPilum
   {
                           ValeurPilum ();   // pour l'initialisation


   Chaine                  TypeSousFormeDeChaine ();

   // methodes de TEST DE TYPE DYNAMIQUE
   int                     CommeEntier ();
   double                  CommeFlottant ();
   bool                    CommeBooleen ();
   CharPTR                 CommeChaine ();
   ListePTR                CommeListe ();

   Chaine                  SousFormeDeChaine ();
   TypeValeurPilum         fTypeValeurPilum;

   union
      {
      AdresseDansLeCode    fAdresseDansLeCode;
      AdresseDansLaPile    fAdresseDansLaPile;
```

4 Rappel: `ValeurPilum` est l'unité d'encombrement des informations dans la pile d'exécution.

5 Un exemple sur l'usage des unions anonymes en C++: http://msdn.microsoft.com/en-us/library/35ect93t(VS.80).aspx

```
      long                    fEntier;
      double                  fFlottant;
      bool                    fBooleen;
      char                    fCaractere;
      ListePTR                fListe;
      CharPTR                 fChaine;
                                 // Chaine (std::string) est un objet,
                                 // et donc pas autorise dans une union
      }; // union

   }; // ValeurPilum
```

```
¦ Implantation de ValeurPilum::CommeListe [Pilum/Pilum.cc, 0172 à 0189]

ListePTR
ValeurPilum :: CommeListe()
   {
   switch (fTypeValeurPilum)
      {
      case kListe:
         return fListe;

      default :
         SortieSurErreur (
            MiseEnForme (
               gLANGUE_Pilum ->
                  FormatCommeChaine (),      // Valable pour cette fonction aussi.
               TypeSousFormeDeChaine () ),
            1 );
         return false;  // superflu
      }  // switch
   }  // ValeurPilum :: CommeListe ()
```

On modifie maintenant la méthode `ValeurPilum::SousFormeDeChaine` pour qu'elle puisse retourner le contenu d'une liste sous la forme d'une chaîne de caractères, avec une certaine mise en forme

```
¦ Modification de ValeurPilum::SousFormeDeChaine [Pilum/Pilum.cc, 0339 à 0340]

Chaine
ValeurPilum :: SousFormeDeChaine ()
   {
      ...
      case kListe:
         return MiseEnForme( "%s", fListe->str() );

      default:
         return
            MiseEnForme (
               gLANGUE_Pilum ->
                  FormatTypeValeurPilumInconnu (),
               fTypeValeurPilum );
      }  // switch

   return "";  // superflu
   }  // ValeurPilum :: SousFormeDeChaine
```

Le premier objectif est atteint; la machine **accepte** des listes mais n'a aucun moyen de les manipuler. Pour remédier ce problème, il faut **étendre le jeu d'instructions** de la machine, et pour cela on touche en premier le type énuméré `CodeOpPilum` pour y **rajouter de nouveaux codes opératoires**[6]

6 Un code opératoire: http://en.wikipedia.org/wiki/Opcode

```
¦ Modification de l'énumération CodeOpPilum [Pilum/Pilum.h, 0154 à 0158]

enum CodeOpPilum
   {
   iInstructionInconnue,

   iHalte,
   ...
   iLog,                    iExp,

   iGenererListe,
   iLireListe,
   iEcrireListe,
   iCar,                    iCdr,                    iCons,
   iAppend,                 iConcatListes,           iLongueurListe,

   iLongueurChaine,         iChaineVide,             iContientChaine,
   iConcatChaines,          iRepliquerChaine,        iRemplacerChaine

   }; // CodeOpPilum
```

Parmi les nouveaux codes opératoires de la machine Pilum, il est facile de discerner lesquels s'appliquent à nos six instructions définies précédemment. Par contre, vous remarquez les trois codes opératoires `iLireListe`, `iEcrireListe` et `iGenererListe`. Les deux premiers sont utilisés pour l'entrée/sortie des listes tandis que `iGenererListe` est utilisé pour créer une liste à partir d'un ensemble de valeurs sur la pile, et une taille designant le nombre d'éléments de la nouvelle liste. Comme pour les terminaux, il faut avoir une représentation des codes opératoires sous la forme d'une chaîne de caractères. Je modifie la méthode `InstructionPilum::CodeOpSousFormeDeChaine`

```
¦ Modification de InstructionPilum::CodeOpSousFormeDeChaine [Pilum/Pilum.cc, 0568 à 0585]

Chaine
InstructionPilum :: CodesOpSousFormeDeChaine ()
   {
   switch (fCodeOpPilum)
      {
      ...
      case iGenererListe:
         return "GenererListe";
      case iLireListe:
         return "LireListe";
      case iEcrireListe:
         return "EcrireListe";
      case iCar:
         return "Car";
      case iCdr:
         return "Cdr";
      case iCons:
         return "Cons";
      case iAppend:
         return "Append";
      case iConcatListes:
         return "ConcatListes";
      case iLongueurListe:
         return "LongueurListe";

      ...
      }  // switch
   }  // InstructionPilum :: CodesOpSousFormeDeChaine
```

V.1.a. L'interprète Pilum

L'interprète de la machine Pilum, implanté dans la méthode `Pilum::Executer`, permet de lancer l'exécution du code à partir d'une certaine adresse. Pour que l'interprète soit capable d'exécuter les nouvelles instructions de la machine Pilum, il faut modifier sa **boucle d'interprétation**. Cette modification est faite dans le fichier **`Pilum.cc`**, des lignes **`2615`** à **`2852`**.

V.2. Synthèse de code Pilum

Comme pour l'évaluation directe, dans ce chapitre je compléte les méthodes `Synthetiser` pour chacun des graphes sémantiques. Ces méthodes construisent le code Pilum, et par conséquent, dépendent des instructions fournis par la machine Pilum. Mais avant de passer à l'implantation de ces méthodes, je modifie la méthode `SynthetiseurPilumLista::SynthetiserEvaluation` qui affiche le résultat d'évaluation d'une liste par la machine Pilum

```
¦ Modification de SynthetiseurPilumLista::SynthetiserEvaluation [SynthesePilumLista.cc, 0356 à 0357]

   ...
   else if (dynamic_cast <TypeNombre *> (leType) != NULL)
      Zeroadique (iEcrireFlottant);

   else if (dynamic_cast <TypeBooleen *> (leType) != NULL)
      Zeroadique (iEcrireBooleen);

   else if (dynamic_cast <TypeChaine *> (leType) != NULL)
      Zeroadique (iEcrireChaine);

   else if (dynamic_cast <TypeListe *> (leType) != NULL)
      Zeroadique (iEcrireListe);

   else
      {
      // RIEN A FAIRE
      }
   ...
```

Et pour marquer la fin du projet, l'implantation des méthodes `Synthetiser`

```
¦ Modification de ValeurListe::Synthetiser [Semantique/DescrSemantique.cc, 0497 à 0518]

void
ValeurListe::Synthetiser( ContexteDeSynthesePTR leContexteDeSynthese )
{
   ListeDescrSemantique::iterator it = fListeDeDescriptionsSemantique.begin();

   // Synthetiser chacun des elements de la liste dans le context donne. Chaque element sera empile
   // sur la pile de donnees.
   for (; it != fListeDeDescriptionsSemantique.end(); it++)
      (*it)->Synthetiser( leContexteDeSynthese );

   // Ensuite empiler le nombre d'elements de la liste sur la pile.
   long laTaille = fListeDeDescriptionsSemantique.size();
   leContexteDeSynthese->LeSynthetiseurPilumLista()->Entier( laTaille );

   // Une fois que les donnees sont empilees, l'instruction qui reconstruit la liste est empilee a
   // son tour dans la pile d'execution.
   leContexteDeSynthese->LeSynthetiseurPilumLista()->Zeroadique( iGenererListe );
}

¦ Modification de Car::Synthetiser [Semantique/Listes.cc, 0178 à 0186]
```

```
void
Car::Synthetiser( ContexteDeSynthesePTR leContexteDeSynthese )
{
   // Synthetiser l'operande dans le contexte donne et ajouter le code operatoire 'iCar'.
   fOperande->Synthetiser( leContexteDeSynthese );
   leContexteDeSynthese->LeSynthetiseurPilumLista()->Zeroadique( iCar );
}


¦ Modification de Cdr::Synthetiser [Semantique/Listes.cc, 0205 à 0213]

void
Cdr::Synthetiser( ContexteDeSynthesePTR leContexteDeSynthese )
{
   // Synthetiser l'operande dans le contexte donne et ajouter le code operatoire 'iCdr'.
   fOperande->Synthetiser( leContexteDeSynthese );
   leContexteDeSynthese->LeSynthetiseurPilumLista()->Zeroadique( iCdr );
}


¦ Modification de Cons::Synthetiser [Semantique/Listes.cc, 0263 à 0271]

void
Cons::Synthetiser( ContexteDeSynthesePTR leContexteDeSynthese )
{
   // Synthetiser les operandes dans le contexte donne et ajouter le code operatoire 'iCons'.
   OperateurBinaire::Synthetiser( leContexteDeSynthese );
   leContexteDeSynthese->LeSynthetiseurPilumLista()->Zeroadique( iCons );
}


¦ Modification de Append::Synthetiser [Semantique/Listes.cc, 0322 à 0330]

void
Append::Synthetiser( ContexteDeSynthesePTR leContexteDeSynthese )
{
   // Synthetiser les operandes dans le contexte donne et ajouter le code operatoire 'iAppend'.
   OperateurBinaire::Synthetiser( leContexteDeSynthese );
   leContexteDeSynthese->LeSynthetiseurPilumLista()->Zeroadique( iAppend );
}


¦ Modification de ConcatListes::Synthetiser [Semantique/Listes.cc, 0355 à 0363]

void
ConcatListes::Synthetiser( ContexteDeSynthesePTR leContexteDeSynthese )
{
   // Synthetiser les operandes dans le contexte donne et ajouter le code operatoire 'iConcatListes'.
   OperateurBinaire::Synthetiser( leContexteDeSynthese );
   leContexteDeSynthese->LeSynthetiseurPilumLista()->Zeroadique( iConcatListes );
}


¦ Modification de LongueurListe::Synthetiser [Semantique/Listes.cc, 0382 à 0390]

void
LongueurListe::Synthetiser( ContexteDeSynthesePTR leContexteDeSynthese )
{
   // Synthetiser l'operande dans le contexte donne et ajouter le code operatoire 'iLongueurListe'.
   fOperande->Synthetiser( leContexteDeSynthese );
   leContexteDeSynthese->LeSynthetiseurPilumLista()->Zeroadique( iLongueurListe );
}
```

V.2.a. Traces d'exécution

```
COMMANDE: ./Pilum_SANS_DEBUG -pc
```

```
Contenu de la memoire du code (35 instructions)
-----------------------------------------------

   0:  Sauter                          8
   1:  EmpilerFlottant                         1.000000
   2:  EmpilerFlottant                         2.000000
   3:  EmpilerFlottant                         3.000000
   4:  EmpilerEntier                   3
   5:  EmpilerBooleen                  Faux
   6:  GenererListe
   7:  RetourDeFonction                0
   8:  Sauter                          24
   9:  EmpilerFlottant                         1.000000
  10:  EmpilerFlottant                         2.000000
  11:  EmpilerChaine                   "trois"
  12:  EmpilerFlottant                         2.000000
  13:  EmpilerFlottant                         2.000000
  14:  FoisFlottant
  15:  EmpilerBooleen                  Vrai
  16:  EmpilerEntier                   1
  17:  EmpilerBooleen                  Vrai
  18:  GenererListe
  19:  Appel                           1
  20:  EmpilerEntier                   6
  21:  EmpilerBooleen                  Vrai
  22:  GenererListe
  23:  RetourDeFonction                0
  24:  EcrireFinDeLigne
  25:  EmpilerChaine                   "Valeur:"
  26:  EcrireChaine
  27:  EcrireFinDeLigne
  28:  Appel                           9
  29:  EcrireListe
  30:  EcrireFinDeLigne
  31:  EmpilerChaine                   "================="
  32:  EcrireChaine
  33:  EcrireFinDeLigne
  34:  Halte

-----------------------------------------------

Valeur:
[1.000000, 2.000000, "trois", 4.000000, [Vrai], [1.000000, 2.000000, 3.000000]]
=================

*** Temps d'execution par Pilum: 0 seconde(s) ***
```

Références

- Cours Compilateurs et Interprètes, M. Jacques MENU
- Documentation Lista
  http://cui.unige.ch/DI/cours/CompInterpretes/ListaSource/doc/html/
- Polycopié Langages de programmation (EPFL 1992)
  http://cui.unige.ch/DI/cours/CompInterpretes/LANGAGES%20BOOK.pdf