Grenoble INP
Année Spéciale Informatique

Projet Compilation

Matthieu Moy
(Transparents originaux de Catherine Oriat)
Ensimag/Laboratoire Verimag
Matthieu.Moy@imag.fr

I. Introduction

1. Présentation du projet

Buts du projet :

- écrire un compilateur « zéro-défaut » pour le langage Cas ;
- utiliser des générateurs d'analyseurs lexical et syntaxique (Aflex et Ayacc) ;
- travailler en équipe.

2. Le langage Cas

« Cas » signifie « Compilation Année Spéciale ».

Exemple de programme Cas

```
-- Calcul de la factorielle
program
   n, fact : integer ;
begin
   write("Entrer un entier : ") ;
   read(n) ;
   fact := 1 ;
   while n >= 1 do
      fact := fact * n ;
      n := n - 1 ;
   end ;
   write("fact(", n, ") = ", fact) ;
   new_line ;
end.
```

Spécifications du langage

a) Lexicographie (Lexicographie.txt page B-1)

La lexicographie définit les *mots* (ou *unités lexicales*) du langage Cas.

Exercice
Les chaînes suivantes sont-elles des identificateurs du langage Cas ?

```
toto, toto_1, toto__1, 2_a,  _toto
```

Les chaînes suivantes sont-elles des constantes entières du langage Cas ?

```
12, -12, 12e2, 12.5e2, 12e+2
```

Les chaînes suivantes sont-elle des constantes réelles du langage Cas ?

```
0.12, .12, 1.5e+3, 1.5e-3, 1e-2, 12, 1.2e++2
```

b) Syntaxe hors-contexte (Syntaxe.txt page B-3)

La syntaxe hors-contexte définit les phrases du langage Cas.

Exercice
Ecrire un programme Cas qui ne fait rien.

c) Syntaxe contextuelle (Context.txt page B-6)

La syntaxe contextuelle (ou sémantique statique) du langage Cas définit :

- les règles de déclaration des identificateurs ;
- les règles d'utilisation des identificateurs ;
- les règles de typage des expressions.

Exercice

Faire la liste de tous les messages d'erreurs contextuelles. Pour chaque message d'erreur, donner un exemple de programme Cas incorrect.

3. Vue d'ensemble du compilateur

Le compilateur Cas comporte trois passes
(le programme va être parcouru trois fois).

Cela permet de bien décomposer les problèmes.

a) Passe 1

- **Analyse lexicale**
  Consiste à décomposer un programme Cas, donné sous la forme d'une suite de caractères, en une suite de mots (ou unités lexicales).
  A chaque unité lexicale reconnue est associée un *lexème* (ou « *token* »).

  Exemple
  La suite de caractères « `x := 2 * (a + b) ;` » correspond à la suite de lexèmes :

```
Idf_Lex ("x")
Affect_Lex
Constent_Lex (2)
'*'
'('
Idf_Lex ("a")
'+'
Idf_Lex ("b")
')'
';'
```

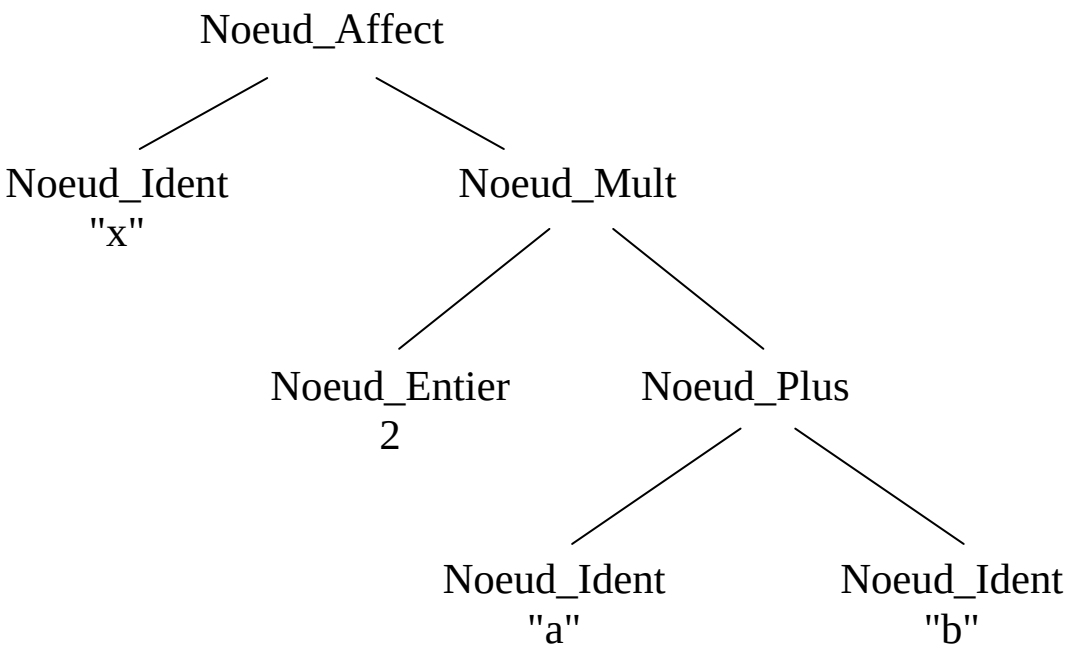
  Le type Token, qui déclare les différents « tokens », est défini dans syntax_tokens.ads page C-24.

- **Analyse syntaxique**
  Consiste à déterminer si une suite de mots est une phrase du langage et à construire un arbre abstrait du programme.

- **Construction de l'arbre abstrait du programme**

  Arbre abstrait correspondant à l'instruction

```
x := 2 * (a + b) ;
```



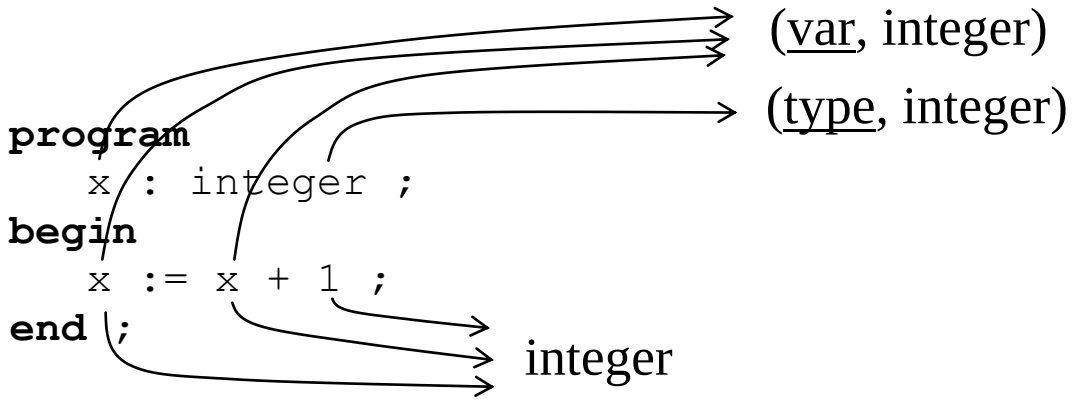
b) Passe 2 : vérifications contextuelles et décoration de l'arbre abstrait

But de la passe 2

- vérifier qu'un programme Cas est contextuellement correct ;
- décorer l'arbre abstrait du programme.

On décore les identificateurs avec leur définition et les expressions avec leur type.

Exemple



Principe

On construit un *environnement*, qui associe à tout *identificateur* sa *définition*.

Dans le langage Cas, on distingue

- les identificateurs de types ;
- les identificateurs de constantes ;
- les identificateurs de variables.

Les identificateurs de types et de constantes sont uniquement des identificateurs prédéfinis
(on ne peut déclarer que des identificateurs de variable).

Vérifications contextuelles

Les vérifications contextuelles sont réalisées par un parcours de l'arbre abstrait du programme.

Le parcours des *déclarations* permet de construire l'environnement.

Le parcours des *instructions* permet de vérifier que les identificateurs sont utilisés conformément à leur déclaration, et que les expressions sont bien typées.

Lors de ce parcours, l'arbre abstrait est décoré, afin de préparer la passe 3.

c) Passe 3

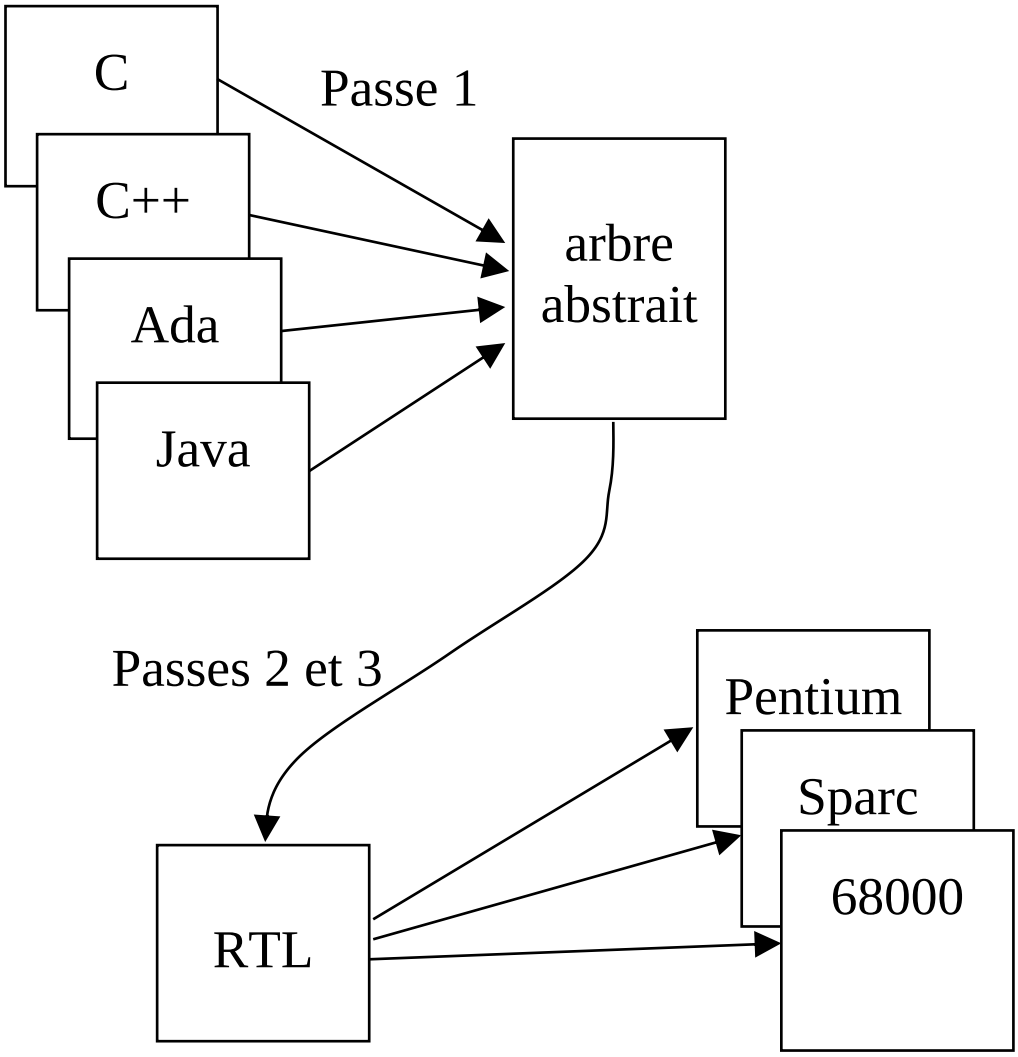
La passe 3 consiste à parcourir l'arbre abstrait décoré une seconde fois et à produire du code exécutable.

On produit du code pour une machine abstraite proche du 68000 *(Machine_Abstraite.txt page B-18)*.

Intérêts d'utiliser une machine abstraite :

- faire abstraction des particularités de bas niveau des langages assembleurs (comme par exemple les problèmes d'alignement en 68000) ;
- permettre la production de code assembleur réel pour plusieurs machines similaires
  (écrire facilement plusieurs « back-ends » de compilateurs).

Exemple de gcc



Le compilateur gcc peut compiler des programmes écrits en C, C++, Ada ou Java.

gcc utilise une structure d'arbre unique pour tous ces langages.

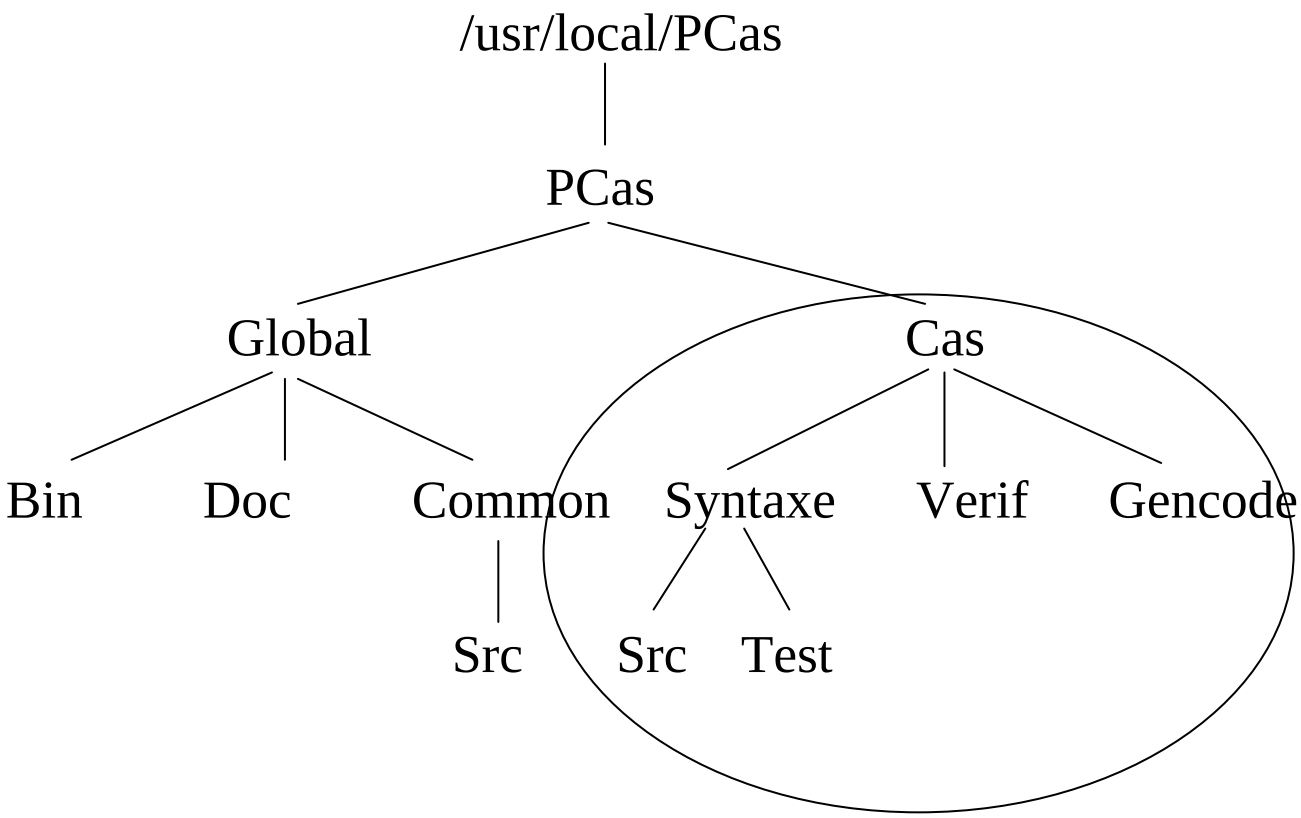
gcc produit du code « RTL » (Register Transfer Language), code pour une machine abstraite dont la syntaxe est proche du Lisp.
Il y a plusieurs « back-ends » pour différentes machines.

4. Environnement de programmation

(Environnement.txt page A-2)

Projet est développé en binômes (pour les passes 1 et 2) sur ensisun.

Organisation des répertoires
Sur ensisun :



Répertoire à récupérer via Git.

Commandes du projet

Compilation

- make
- make clean
  (à ne pas faire avant chaque compilation !)

Vérification des accolades dans un fichier Aflex ou Ayacc

- accolades

Extraction des commentaires

- commentaire

Debug

- adastack ou adadebug

exemple : `adastack test_synt fich.cas`

Commandes auxiliaires (utilisées dans les makefiles)

- gnatmake –g –gnato –I *bibliothèques*
- gnaflex
- gnayacc

Mise en place de l'environnement

SeanceMachine1.txt page A-4.

5. Planning

Les programmes sont ramassés automatiquement via Git.

- Fin de la semaine de stage : terminer la passe 1, commencer la passe 2.
  Passe 1 à rendre le mercredi 15 février 2012, 18h00

- 15 février - 4 avril : terminer la passe 2.
  Passe 2 à rendre le 4 avril 2012, 18h00.

- Fin du stage de mai : terminer la passe 3.

II. Passe 1 : analyse lexicale et syntaxique, construction de l'arbre abstrait

1. Aflex

a) Introduction

Aflex : générateur d'analyseur lexical pour Ada (dans la lignée de Lex, générateur d'analyseur lexical pour C).

Aflex permet, à partir d'un fichier d'entrée qui décrit la forme d'un ensemble d'unités lexicales, de générer un analyseur lexical.

Commande : `gnaflex fich.l`

Cette commande produit 5 fichiers :

- `fich.adb` : unité Ada correspondant à `fich.l`
- `fich_dfa.ads, fich_dfa.adb` : paquetage `Fich_DFA`
- `fich_io.ads, fich_io.adb` : paquetage `Fich_IO`.

Dans fich.adb se trouve la fonction principale de l'analyseur lexical, qui permet de lire le lexème suivant :

```
function YYLex return Token ;
```

- `Token` est un type énuméré, défini soit par l'utilisateur, soit par l'outil Ayacc (comme c'est le cas dans le projet).
  `Token` contient au minimum les valeurs `Error` et `End_Of_Input`.

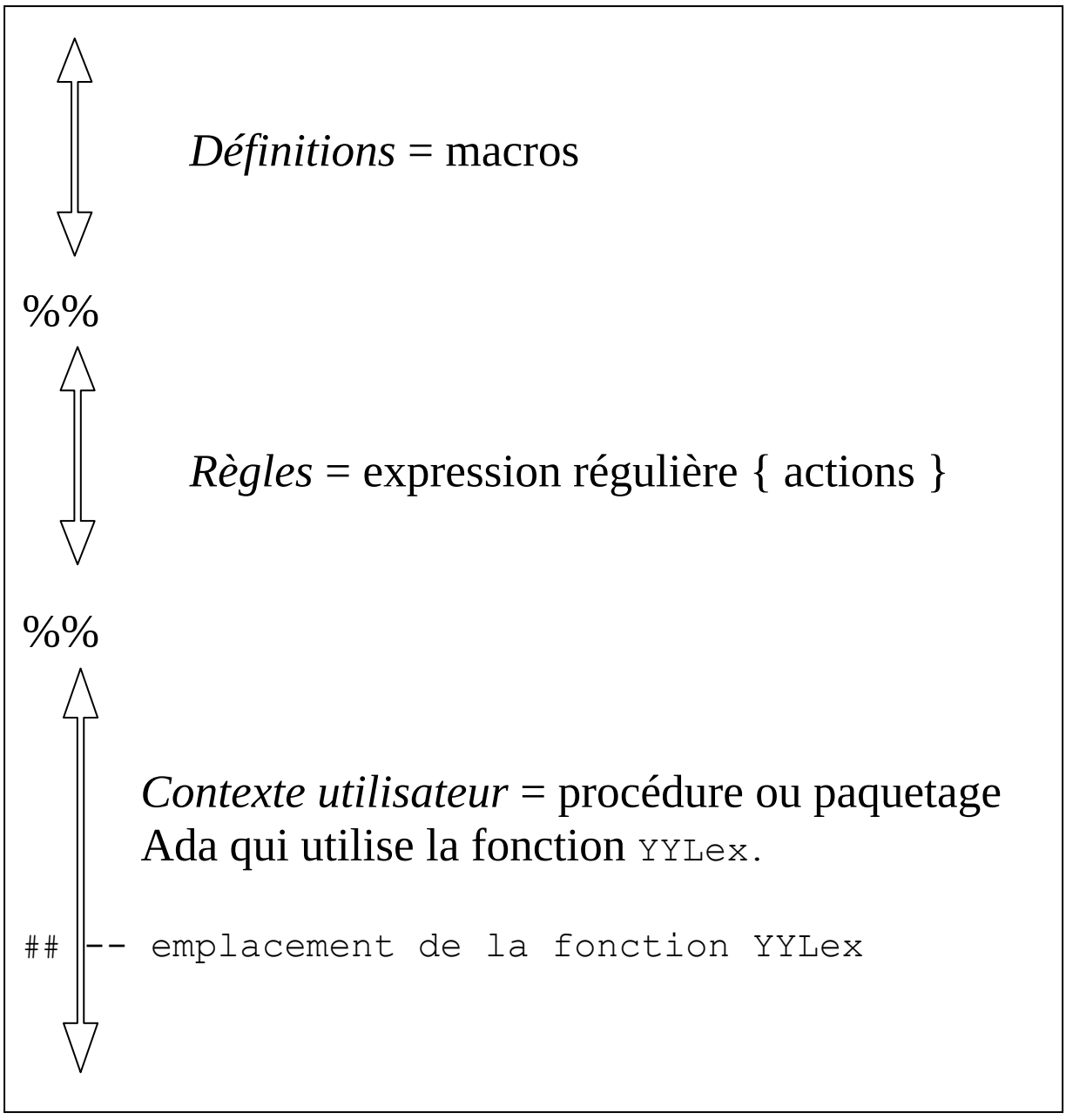
- La fonction `YYLex` lit des caractères sur l'entrée standard, les partitionne en unités lexicales, exécute les actions correspondantes et retourne un lexème correspondant à l'unité lexicale reconnue.

Dans fich_dfa.ads, on trouve la fonction

```
function YYText return String ;
```

qui retourne la chaîne qui vient d'être reconnue.

b) Forme du fichier fich.l



Expressions régulières

x	Le caractère ‘x’
"("	Le caractère ‘(’
\(	Le caractère ‘(’
":="	Le caractère ‘:’ suivi du caractère ‘=’

Ensemble des caractères spéciaux (appelés « opérateurs » dans Aflex) :

" \ { } [ ] ^ $ < > ? . * + | ( ) /

{IDF}	La définition IDF
*x**	*x* répété 0, 1, ... *n* fois
x+	*x* répété 1, 2, ... *n* fois
x?	*x* optionnel (0 ou 1 fois)
x|*y*	*x* ou *y*
[abc]	‘a’, ‘b’ ou ‘c’
[A-Za-z]	caractères entre ‘A’ et ‘Z’ et ‘a’ et ‘z’
[^abc]	tout caractère sauf ‘a’, ‘b’ et ‘c’
\n	retour à la ligne
\t	tabulation
\040	espace
\041	‘!’
\042	‘"’
\134	‘\’
\176	‘~’
.	Tout caractère sauf \n

Définitions

Dans la partie « Définitions », on associe à des noms des expressions régulières.

Exemple

```
CHIFFRE [0-9]
LETTRE [A-Za-z]
IDF {LETTRE}({LETTRE}|{CHIFFRE}|"_")*
```

Remarques

- Les expressions régulières ne doivent pas contenir d'espaces.
- Dans les noms, les minuscules/majuscules sont significatives : `CHIFFRE` et `chiffre` sont deux noms différents.

Règles

Une règle est une expression régulière suivie d'une suite d'actions :

expression régulière { *actions* }

Une règle associe à une expression régulière des actions à effectuer.

Les actions sont des instructions Ada.

Exemple

```
":="    { return Affect_Lex ; }
"/="    { return Diff_Lex ; }
```

Reconnaissance du préfixe le plus long possible
Lorsqu'il y a une ambiguïté sur la règle à appliquer, comme

```
toto        { action1 ; }
tototi      { action2 ; }
```

on reconnaît le préfixe le plus long possible. Si la chaîne d'entrée est `tototi`, on applique `action2`, si la chaîne d'entrée est `totota`, on applique `action1`.

Lorsque les deux chaînes sont de la même longueur, on applique la première règle :

```
toto        { action1 ; }
tot.        { action2 ; }
```

Si la chaîne d'entrée est `toto`, on applique `action1`.

Exemple
Affichage des commentaires d'un programme Cas.

Remarque
`write("-- Ceci n'est pas un commentaire") ;`

```
-----------------------------------------------
-- Exemple simple d'utilisation de Aflex
-- Ecrit les commentaires d'un programme Cas
-----------------------------------------------

-- Caractères imprimables
CAR_IMP [\040-\176]
-- Commentaires
COMMENT "--"({CAR_IMP}|\t)*
-- Caractères d'une chaîne
CHAINE_CAR [\040\041\043-\176]
-- Chaîne de caractères
CHAINE \"({CHAINE_CAR}|\"\")*\"

%%

{CHAINE}        {   null ; }
{COMMENT}       {   Put_Line(YYText) ;
                    return Comm ; }
.|\n            {   null ; }

%%

with Ada.Text_IO ;
use Ada.Text_IO ;

procedure Comment is

   -- Type Token, contenant les valeurs
   -- End_Of_Input et Error.
   type Token is (End_Of_Input, Error, Comm) ;
   Tok : Token ;

## -- La fonction YYLex sera générée ici

begin
   loop
      Tok := YYLex ;
      exit when Tok = End_Of_Input ;
   end loop ;
end Comment ;
```

2. Mise en oeuvre de l'analyse lexicale

a) Les types de base

Le paquetage `Types_Base` (page C-2) définit :

- le type `Entier`, pour les littéraux entiers qui apparaissent dans un programme Cas ;
- le type `Reel`, pour les littéraux réels qui apparaissent dans un programme Cas ;
- le type `Chaine`, pour les littéraux chaînes et les identificateurs qui apparaissent dans un programme Cas.

Remarque sur le type Chaine

On Ada, on a le type `String` qui est un tableau non contraint de caractères.

```
type String is
   array(Positive range <>) of Character ;
```

Pour déclarer une variable de type `String`, on doit en connaître la taille.

```
S : String(1 .. 10) ;
S : String(T'Range) ;
            -- où T est de type String.
```

Dans le projet, pour faciliter la manipulation de chaînes de différentes tailles, on utilise le type abstrait `Chaine`.

```
package Types_Base is
   -- ...
   type Chaine is private ;
   Chaine_Indef : constant Chaine ;
private
   type Structure_Chaine ;
   type Chaine is access Structure_Chaine ;
   Chaine_Indef : constant Chaine := null ;
end ;
```

Les conversions entre les types String et Chaine se font à l'aide de deux fonctions :

```
function Creation_Chaine(S : String)
   return Chaine ;
function Acces_String(C : Chaine)
   return String ;
```

Quand on rencontre une `String`, on la convertit immédiatement en une `Chaine`. Ensuite, on ne manipule que des `Chaine`, sauf lors d'un affichage :

```
C : Chaine ;
C := Creation_Chaine("toto") ;
Put_Line(Acces_String(C)) ;
```

b) Paquetage Tables

Le paquetage Table permet d'associer des informations à des chaînes de caractères (page C-6).

Le corps du paquetage est implémenté à l'aide d'une table à adressage dispersé.

Ce paquetage sert :

- en passe 1 pour implémenter le dictionnaire, qui sert à associer aux mots réservés leur token ;
- en passe 2 pour implémenter l'environnement, qui sert à associer aux identificateurs leur définition.

c) Implémentation de l'analyse lexicale

L'analyse lexicale est réalisée par la fonction

```
function YYLex return Token ;
```

générée par Aflex. Cette fonction est utilisée par l'analyseur syntaxique, généré par l'outil Ayacc (générateur d'analyseur syntaxique).

Le type `Token` est produit par Ayacc.

`Un Token` ne contient pas toute l'information nécessaire contenue dans un lexème (chaînes de caractères, numéros de lignes).

Ayacc fournit le type `YYStype`, et la variable

```
YYLVal : YYStype ;
```

qui permet :

1. de stocker des informations supplémentaires associées aux lexèmes ;
2. de stocker des informations nécessaires associées aux non-terminaux de la grammaire (lors de l'analyse syntaxique).

Le type `YYSType` est défini dans le fichier syntax_token.ads, page C-24.

Le type `YYSType` est un type article avec discriminant (certains champs sont présents ou non suivant la valeur du champ discriminant).

Le discriminant est de type `Type_Valeur`.

```
type Type_Valeur is
  (Lex_Idf,    --> pour les identificateurs
   Lex_Entier, --> pour les entiers
   Lex_Reel,   --> pour les réels
   Lex_Chaine, --> pour les chaînes de caract.
   Lex_Autre,  --> pour les autres lexèmes
   NT) ;       --> pour les non-terminaux

type YYSType ...
   ...
  when Lex_Idf =>
    Num_Ligne_Idf : Natural ;
    -- No de ligne où apparaît l'identificateur
    Val_Idf : Chaine ;
    -- Chaîne de caractères qui correspond à
    -- l'identificateur
```

Exemple de valeur de type `YYStype` :

```
(Lex_Idf, Ligne_Courante, C) ;
```

où :

```
Ligne_Courante : Natural ;
C : Chaine ;
```

Finalement, les actions de l'analyse lexicale :

1. modifient la valeur de `YYLVal` ;
2. renvoient une valeur de type `Token`.

Exemples de règles

```
[ \t]        { null ; }
\n           { Ligne_Courante :=
                 Ligne_Courante + 1 ; }
{COMMENT}    { null ; }
":="         { YYLVal :=
                 (Lex_Autre,  Ligne_Courante) ;
               return Affect_Lex ; }
".."         { YYLVal : =
                 (Lex_Autre, Ligne_Courante) ;
               return Doublepoint_Lex ; }
{ENTIER}     { YYLVal := (Lex_Entier,
                      Ligne_Courante,
                      Valeur_Entier(YYText)) ;
          -- fonction Valeur_Entier à définir
               return Constent_Lex ; }
```

Regarder les fichiers lexico.ads page C-15 et lexico.l page C-16.

d) Traitement des mots réservés dans l'analyse lexicale

Une solution

On écrit une règle par mot réservé.

```
IF      { YYLVal :=
             (Lex_Autre, Ligne_Courante) ;
          return If_Lex ; }
WHILE   { YYLVal :=
             (Lex_Autre, Ligne_Courante) ;
          return While_Lex ; }
```

Aflex crée alors un gros automate.

Autre solution

On utilise un dictionnaire, qui stocke tous les mots réservés avec leur token correspondant.

Au départ : on initialise le dictionnaire avec tous les mots réservés.

On reconnaît alors les mots réservés comme des identificateurs et on consulte ensuite le dictionnaire pour savoir s'il s'agit d'un mot réservé.

dictionnaire.adb (spécification page C-13, corps page C-14).

```
-- Ajoute l'association (S -> Tok) dans le
-- dictionnaire.
procedure Ajouter_Mot_Reserve
             (S : in String ; Tok : in Token);


-- Retourne :
--   - le token associé à C, si C fait partie
--     du dictionnaire
--   - le token Idf_Lex, sinon.
function Acces_Token(C : Chaine) return Token ;
```

Reconnaissance des identificateurs (dans lexico.l)

```
{IDF} { declare
           Code : Token ;
           C : Chaine ;
        begin
           Chercher_Dict(YYText, Code, C) ;
           if Code = IDf_Lex then
              YYLVal :=
                 (Lex_Idf, Ligne_Courante, C) ;
           else
              YYLVal :=
                 (Lex_Autre, Ligne_Courante) ;
           end if ;
           return Code ;
        end ;
     }
```

Définir `Chercher_Dict` (qui consulte le dictionnaire, après avoir mis la chaîne de caractères en minuscules).

Le paquetage Ada.Characters.Handling

contient la fonction

```
function To_Lower(Item : String)
   return String ;
```

qui permet de passer une chaîne de caractères en minuscules.

⇒ *Travail à effectuer : compléter lexico.l*

Programme de test fourni : test_lex page C-25.

3. Les arbres

a) Syntaxe abstraite (ArbreAbstrait.txt page B-11)

Syntaxe abstraite du langage Cas
définit la représentation intermédiaire utilisée par les compilateurs (ou interprètes) du langage. Cette syntaxe abstraite est définie par une grammaire d'arbres.

Exercice

Donner les arbres abstraits correspondant aux programmes Cas suivants :

- ```
  program
     a : integer ;
     b, c : boolean ;
  begin
     a := 1 ;
     b := a + 1 ;
  end ;
  ```

- ```
  program
     t : array[1..10] of array[1..5] of
  integer ;
  begin
     t[10][5] := 0 ;
  end ;
  ```

b) Spécification du paquetage Arbre (page C-8)

Le paquetage Arbre définit :

- un type énuméré pour les différents noeuds possibles : `Noeud_Affect`, `Noeud_Chaine`, `Noeud_Plus`, `Noeud_Moins`...

- un type `Arbre` ;

- la constante `Arbre_Indef` : arbre indéfini.
  Lorsqu'on construit un arbre, on peut utiliser temporairement `Arbre_Indef`. Lorsqu'on a fini de construire l'arbre, celui-ci ne doit plus contenir de valeurs indéfinies.

- des constructeurs d'arbres, qui permettent de construire des arbres connaissant ces fils. Les constructeurs sont préfixés par `Creation_`.

  Exemples :

```
A1 := Creation_Ident(C, Num_Ligne) ;
A2 := Creation_Entier(4, Num_Ligne) ;
A3 := Creation2(Noeud_Plus, A1, A2, N) ;
```

- des sélecteurs, qui permettent de décomposer un arbre. Les sélecteurs sont préfixés par `Acces_`.

  Exemples :
  ```
  Acces_Entier(A2) → 4
  Acces_Fils1(A3) → A1
  Acces_Fils2(A3) → A2
  ```

- des mutateurs, qui permettent de modifier un arbre. Les mutateurs sont préfixés par `Changer_`.

  Exemples :
  ```
  Changer_Fils1(A3, Creation_Entier(3)) ;
  ```

- des décors, qui serviront lors de la passe 2.

- des procédures d'affichage.

  ```
  procedure Afficher_Arbre
    (A : in Arbre; Niveau : in Natural := 0);
  ```

  permet d'afficher un arbre, avec un certain niveau de détail pour les décors. Si niveau = 0, les décors ne sont pas affichés.

  ```
  procedure Decompiler
    (A : in Arbre; Niveau : in Natural := 0);
  ```
  permet de « décompiler » un arbre abstrait, c'est-à-dire d'afficher le programme Cas correspondant.

c) Implantation des arbres, «sémantique de partage»

Un arbre est un pointeur, ce qui implique qu'on a une « sémantique de partage ».

Exemple

```
A, B, C : Arbre ;
A := Creation2(Noeud_Plus,
               Creation_Entier(1),
               Creation_Entier(2)) ;
B := A ;
Changer_Fils1(B, Creation_Entier(4)) ;

Acces_Fils1(A) → Noeud_Entier(4)
```

Donc la modification de l'arbre `B` a modifié l'arbre `A`.

L'affectation entre deux arbres est une affectation entre <u>pointeurs</u>.

```
C := Creation2(Noeud_Plus,
Creation_Entier(4), Creation_Entier(2)) ;

A = B → true
A = C → false
```

L'égalité entre deux arbres est une égalité entre <u>pointeurs</u> (et non une égalité structurelle).

La plupart des types abstraits utilisés dans le projet sont implémentés à l'aide de pointeurs. Pour tous ces types, on a donc une « sémantique de partage ».

4. Ayacc

a) Introduction

Ayacc
Générateur d'analyseurs syntaxiques pour Ada, dans la lignée de Yacc, générateur d'analyseurs syntaxiques pour C.

A partir d'une grammaire hors-contexte, accompagnée d'un ensemble d'actions Ada, Ayacc produit une procédure `YYParse`.

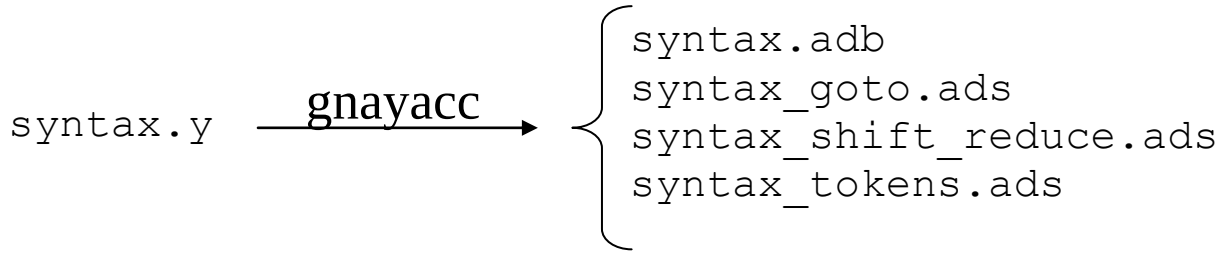
`YYParse` réalise l'analyse syntaxique et effectue les actions associées aux règles de la grammaire.

La procédure `YYParse` utilise

- **`function`** `YYLex` **`return`** `Token ;`
  qui peut être soit définie explicitement, soit être produite par Aflex ;

- **`procedure`** `YYError(S : String) ;`
  qui sera appelée chaque fois qu'une erreur de syntaxe est détectée.

En pratique :

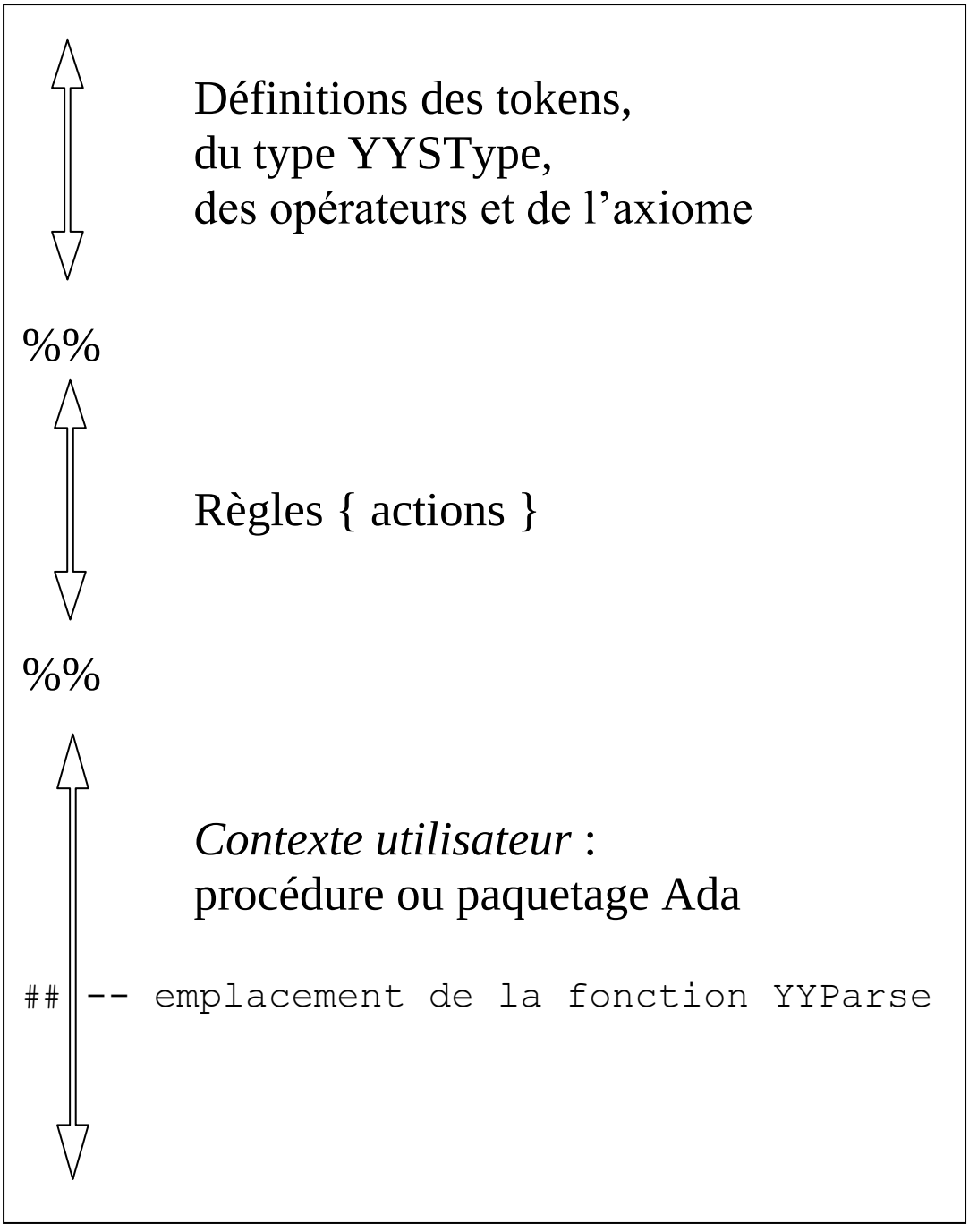


- `syntax.adb` : fichier principal, qui contient la procédure `YYParse` et le code Ada correspondant aux actions à effectuer.
  Dans le projet, ce fichier contient le corps du paquetage `Syntax`.

- `syntax_tokens.ads` : contient les déclarations des types `Token` et `YYSType`, et des variables `YYLVal` et `YYVal`.

- `syntax_goto.ads` et `syntax_shift_reduce.ads` contiennent des tables utilisées par `YYParse`.

b) Structure du fichier syntax.y



c) *Principe de l'analyse syntaxique*

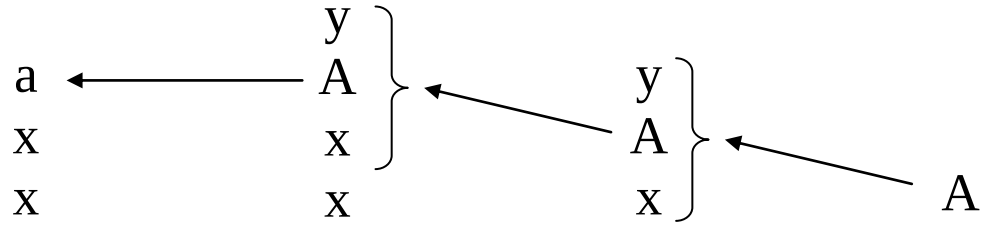
Analyse ascendante
On reconnait des parties droites de règles et on essaie de remonter vers l'axiome de la grammaire.

Exemple 1.
Grammaire pour le langage $x^n\, a\, y^n$.

```
A : 'x' A 'y'   { Put_Line("A -> x A y") ; }
  | 'a'         { Put_Line("A -> a") ;     }
```

On considère la chaîne d'entrée : `x x a y y`

On a des « empilements » et des « réductions ».
(*shift/reduce* en anglais)



Les flèches correspondent à des « réductions ».

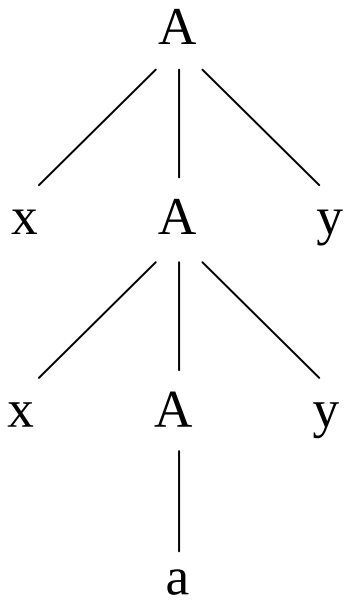
Lorsqu'on réduit, l'action correspondante est effectuée.

On affiche donc :

```
A -> a
A -> x A y
A -> x A y
```

La chaîne est reconnue si, après avoir empilé tous les caractères et effectué toutes les réductions, la pile ne contient que l'axiome de la grammaire.

Arbre de dérivation

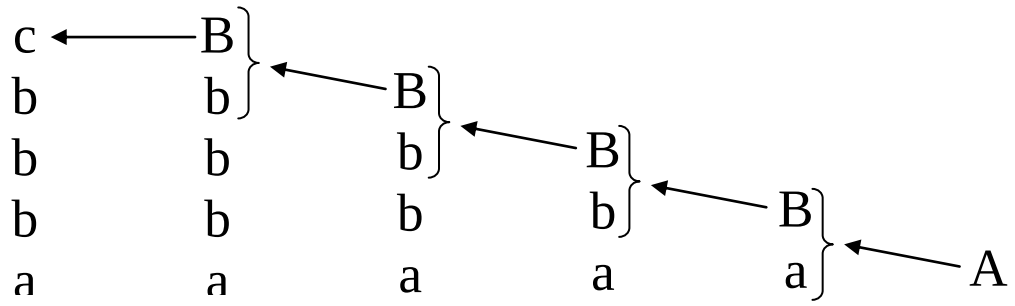


Analyse ascendante : on effectue d'abord les règles associées aux feuilles de l'arbre de dérivation, puis on remonte vers l'axiome.

Exemple 2.
Soit le langage $a\ b^*\ c$, engendré par la grammaire :

```
A : 'a' B
B : 'b' B | 'c'
```

On considère la chaîne d'entrée : `a b b b c`



On doit empiler toute la chaîne d'entrée avant de commencer à réduire.

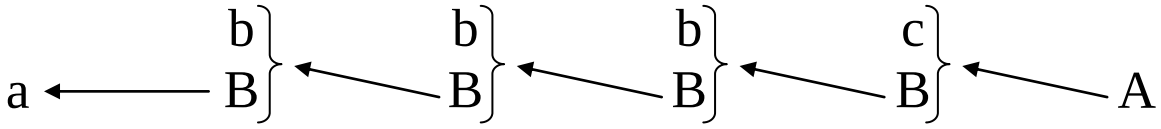
Problème
La pile utilisée par Ayacc a une taille limitée ; donc si la chaîne d'entrée est très longue, la pile déborde.

La grammaire est récursive à droite.

On peut reconnaître le même langage avec une grammaire récursive à gauche :

```
A : B 'c'
B : B 'b' | 'a'
```

On reprend la même chaîne d'entrée : `a b b b c`



Les réductions ont lieu au fur et à mesure des empilements.

Conclusion

En analyse ascendante, on utilise de préférence des règles récursives à gauche (pour des raisons de place mémoire).

Par exemple, dans le projet, on a la règle :

```
liste_inst : liste_inst inst ';'
           | inst ';'
```

Rappel : en analyse *descendante*, on utilise des règles récursives à droite. En effet lorsqu'une règle est récursive à gauche, la grammaire n'est pas LL(1).

d) Attributs

A chaque terminal et non terminal de la grammaire est associé un attribut synthétisé (c'est-à-dire un attribut transmis du fils vers le père dans l'arbre de dérivation).

Pour les terminaux, l'attribut est la valeur `YYLVal` mise à jour dans lexico.l :

```
(Lex_Autre, Num_Ligne_Autre)
(Lex_Idf, Num_Ligne_Idf, Val_Idf)
...
```

Pour les non-terminaux, l'attribut est de la forme :

```
(NT, Val_Arbre).
```

Ces valeurs permettent de construire l'arbre abstrait associé au programme.

Considérons par exemple la règle :

```
program : PROGRAM_Lex liste_decl BEGIN_Lex
   $$          $1          $2        $3
          liste_inst END_Lex
               $4       $5

    $1 ≡ (Lex_Autre, No_Ligne1)
    $2 ≡ (NT, A2)      A2 == $2.Val_Arbre
    $3 ≡ (Lex_Autre, No_Ligne2)
    $4 ≡ (NT, A4)      A4 == $4.Val_Arbre
    $5 ≡ (Lex_Autre, No_Ligne3)
```

Dans l'action associée à cette règle, on calcule la valeur de `$$` en fonction des `$i`:

```
program : PROGRAM_Lex liste_decl BEGIN_Lex
          liste_inst END_Lex
    { $$ := (NT, Creation2(Noeud_Programme,
                    $2.Val_Arbre,
                    $4.Val_Arbre,
                    $1.Num_Ligne_Autre)) ;
    }

idf : IDF_Lex
    { $$ := (NT, Creation_Ident($1.Val_Idf,
                    $1.Num_Ligne_Idf)) ;
    }

const : CONSTENT_Lex
    { $$ := (NT, Creation_Entier($1.Val_Entier,
                    $1.Num_Ligne_Entier)) ;
    }

exp : exp '+' exp
    { $$ := (NT, Creation2(Noeud_Plus,
                    $1.Val_Arbre,
                    $3.Val_Arbre,
                    $2.Num_Ligne_Autre)) ;
    }


exp : facteur
    { $$ := $1 ;
    }
```

Dans syntax_tokens.ads sont définies deux variables globales :

```
   YYLVal, YYVal : YYSType ;
```

`YYLVal` correspond à la valeur de l'attribut des terminaux (`$i` pour les `XXX_Lex`)
`YYVal` correspond à l'attribut de l'axiome de la grammaire (`$$` de program).

Dans syntax.y, on trouve la procédure principale de l'analyseur syntaxique :

```
procedure Analyser_Construire_Arbre
   (A : out Arbre) is
begin
   YYParse ;
   A := YYVal.Val_Arbre ;
end ;
```

⇒ *Travail à effectuer : compléter syntax.y*

Programme de test fourni : test_synt, page C-27.

III. Passe II : vérifications contextuelles

Buts de la passe 2 :

- vérifier qu'un programme Cas est contextuellement correct ;
- enrichir et décorer l'arbre abstrait pour préparer la passe 3.

1. Contraintes contextuelles

Les contraintes contextuelles du langage Cas sont définies dans Context.txt page B-6.

a) Les types du langage Cas

Les types du langage Cas sont les suivants : intervalle d'entiers, réel, booléen, string et tableau.

- intervalle d'entiers
  Exemple : `1..10` représente l'intervalle des entiers de 1 à 10, noté *interval(1,10)*.
  `max_int` est une constante qui représente l'entier maximal du langage Cas, de valeur *valmax*.
  Le type `integer` représente
  interval(-valmax, valmax).

- réel, qui correspond à un sous-ensemble de $\mathbb{R}$.

- booléen, qui correspond à l'ensemble {*vrai*, *faux*}
  On a deux constantes de type booléen : `true` (valeur *vrai*) et `false` (valeur *faux*).

- string. Pas de syntaxe dans le langage Cas. On ne peut donc pas déclarer de variable de type string. On a uniquement des littéraux de type string comme dans l'instruction :

```
write("ok") ;
```

- tableau
  Syntaxe : `array[` *type_intervalle* `]` of *type*
  Exemples :

```
array[1..10] of integer
array[1..10] of array[1..5] of boolean
```

Grammaire de types du langage Cas

EXP_TYPE → INTERVALLE
| real
| boolean
| string
| array(INTERVALLE, EXP_TYPE)

INTERVALLE → interval(entier, entier)

Equivalence de types
équivalence *structurelle* (≠équivalence de nom).

Exemple :

```
v1 : array[1..10] of integer ;
m  : array[1..5] of array[1..10] of integer;
v2 : array[1..10] of integer ;
```

`m[1]`, `m[2]`, `...` `v1` `et` `v2` sont de même type.

```
v1 := v2 ;    -- ok
m[1] := v1 ; -- ok
m := v1 ;     -- interdit
```

b) Règles de visibilité

Les règles de visibilités du langage Cas sont les suivantes :

- On ne peut pas re-déclarer un identificateur déjà déclaré.
- Tout identificateur apparaissant dans un programme Cas doit être déclaré, sauf les identificateurs prédéfinis.
- Les identificateurs prédéfinis ne peuvent pas être redéfinis.

Les identificateurs d'un programme Cas sont de différentes natures :

- identificateurs de constantes (de type intervalle, booléen, réel ou chaîne) ;
- identificateurs de type ;
- identificateurs de variable.

Nature = {const, type, var}.

Seuls des identificateurs de variables peuvent être déclarés dans un programme Cas. Les seuls identificateurs de constante et de type sont donc des identificateurs prédéfinis.

La nature des identificateurs doit être vérifiée.

L’environnement associe à chaque identificateur une définition.

Au début de l’analyse du programme, l’environnement contient uniquement les identificateurs prédéfinis.

Environnement prédéfini :

″boolean″	→	(type, *boolean*)
″false″	→	(const(*faux*), *boolean*)
″true″	→	(const(*vrai*), *boolean*)
″integer″	→	(type, *integer*)
″max_int″	→	(const(*valmax*), *integer*)
″real″	→	(type, *real*)

c) Profils de opérateurs

integer : *type interval(-valmax, valmax)*
interval : un type intervalle quelconque *interval(a,b).*

`not` :	boolean → boolean
`and, or` :	boolean, boolean → boolean
`=, <, >, /=, <=, >=` :	interval, interval → boolean interval, real → boolean real, interval → boolean real, real → boolean
`+, -` :	interval → integer real → real
`+, - , *` :	interval, interval → integer interval, real → real real, interval → real real, real → real
`div, mod` :	interval, interval → integer
`/` :	interval, interval → real interval, real → real real, interval → real real, real → real
`[ ]` (indexation)	array(interval, type) → type

d) Vérifications de type

- Intervalles *exp_const1* . . *exp_const2*
  exp_const1 et *exp_const2* doivent être de type interval.
- Affectations *place* := *expression*
  Le type de *place* et le type de *expression* doivent être compatibles pour l'affectation, c'est-à-dire :
  - *place* et *expression* de type interval (pas forcément avec les mêmes bornes) ;
  - *place* et *expression* de type real ;
  - *place* et *expression* de type boolean ;
  - *place* de type real et *expression* de type interval ;
  - *place* et *expression* de type array, les types des indices étant identiques (de type interval, avec les mêmes bornes), et les types des éléments compatibles pour l'affectation.
- Instructions `if` et `while` : la condition doit être de type boolean.
- Instruction `for` : la variable de contrôle, ainsi que les deux expressions doivent être de type interval.
- Instruction `read` : la place doit être de type interval ou real.
- Instruction `write` : les expressions doivent être de type interval, real ou string.
- Les places et expressions doivent être bien typées vis-à-vis des déclarations et des profils des opérateurs.

2. Enrichissement et décoration de l'arbre abstrait

L'enrichissement et la décoration de l'arbre abstrait a pour but de préparer la passe 3 (génération de code). cf. ArbreEnrichi.txt page B-14.

a) Ajouts de Noeud_Conversion

Le langage Cas autorise l'ajout d'un entier et d'un réel, ou l'affectation d'un entier à un réel.

On ne peut pas réaliser cela directement en assembleur : il faut commencer par convertir l'entier en réel.

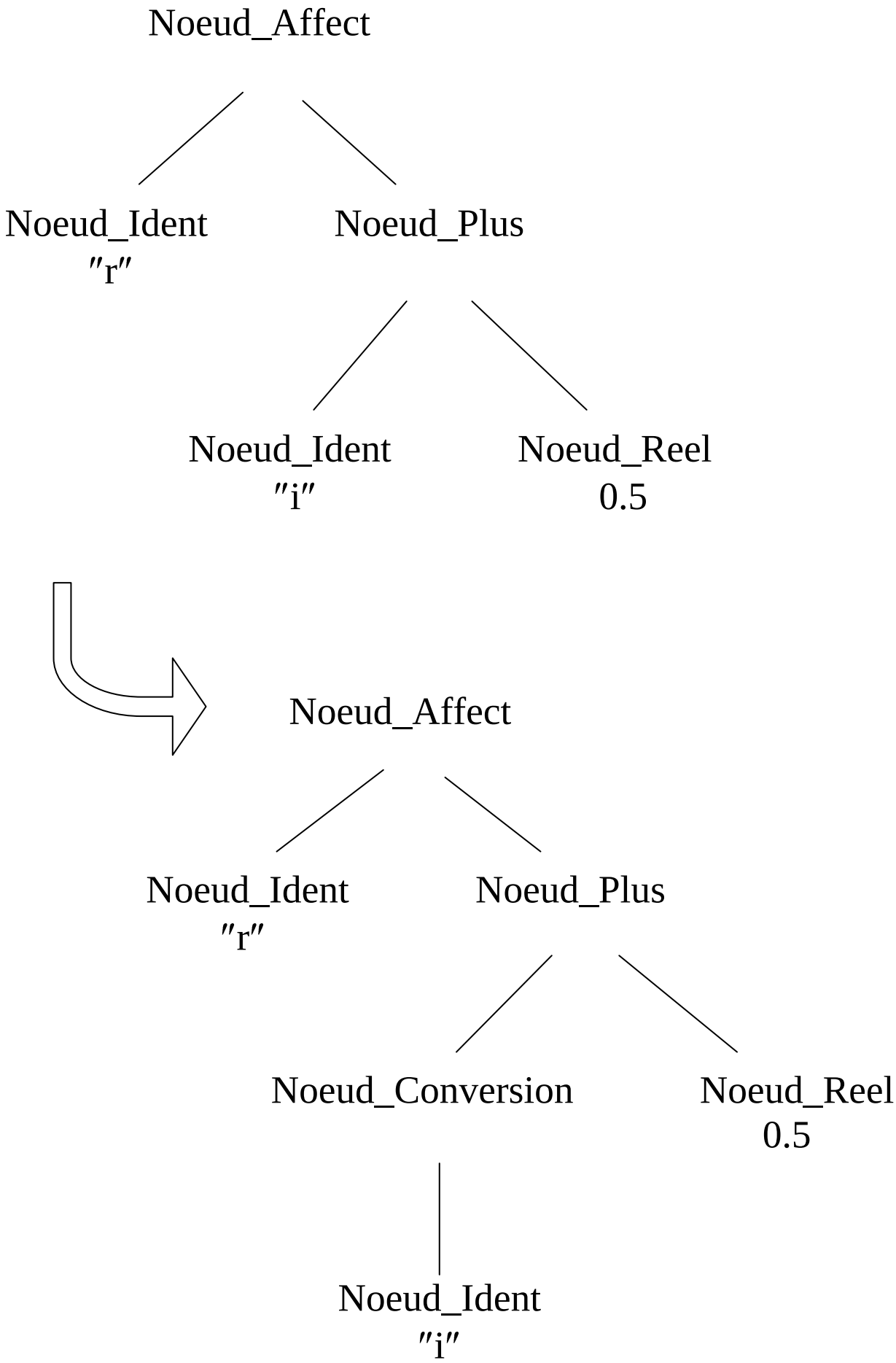
Un Noeud_Conversion représente la conversion d'un entier vers un réel (et non l'inverse !)

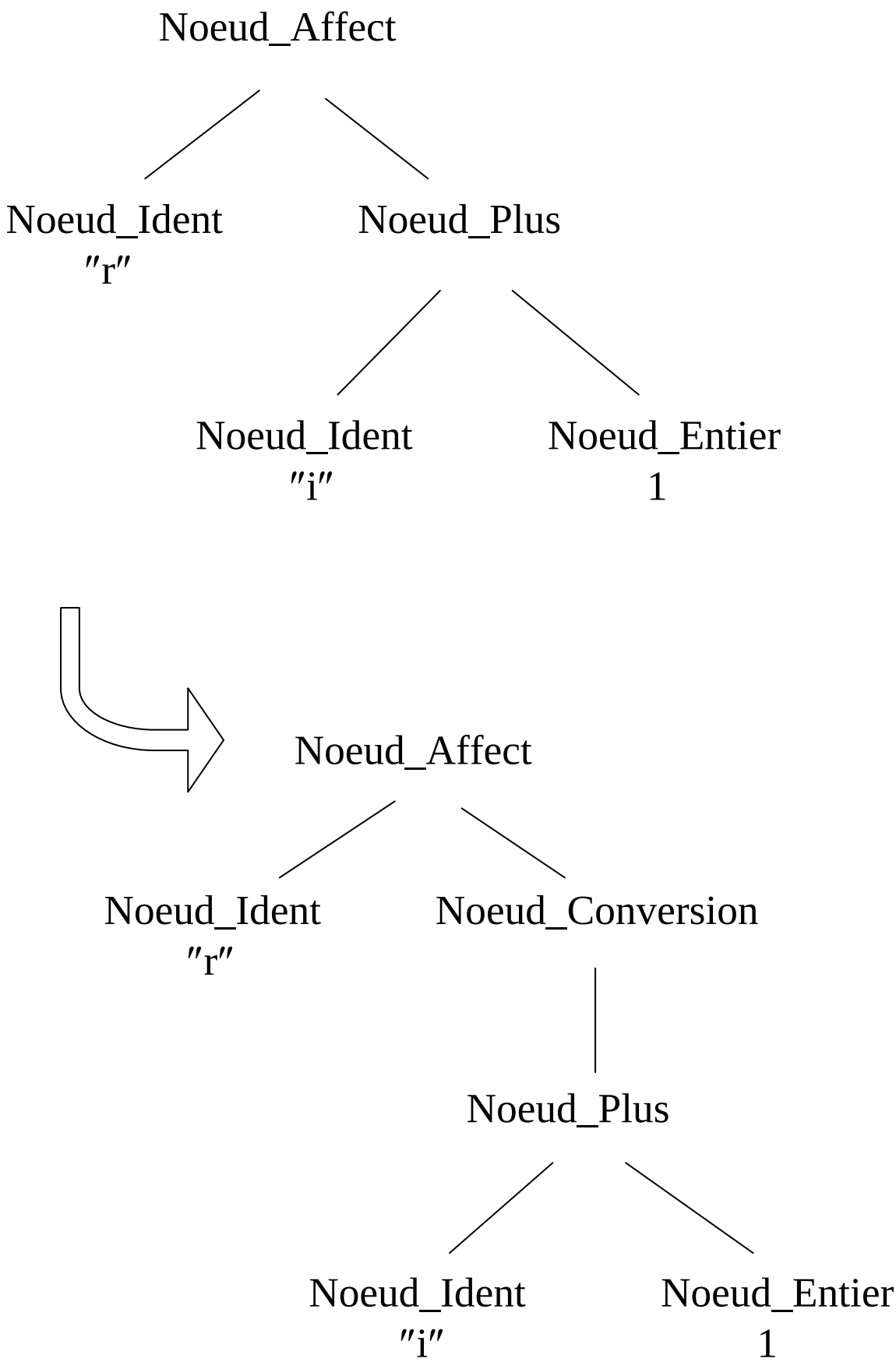
Exemples :

```
r : real ;
i : integer ;

r := i + 0.5 ;
r := i + 1 ;
```

Pour ajouter les Noeud_Conversion, on utilise les procédures `Changer_Fils1` et `Changer_Fils2` du paquetage Arbre.


Noeud_Affect
Noeud_Ident
"r"
Noeud_Plus
Noeud_Ident
"i"
Noeud_Reel
0.5
Noeud_Affect
Noeud_Ident
"r"
Noeud_Plus
Noeud_Conversion
Noeud_Reel
0.5
Noeud_Ident
"i"


Noeud_Affect
Noeud_Ident
"r"
Noeud_Plus
Noeud_Ident
"i"
Noeud_Entier
1
Noeud_Affect
Noeud_Ident
"r"
Noeud_Conversion
Noeud_Plus
Noeud_Ident
"i"
Noeud_Entier
1

b) Décoration de l'arbre abstrait

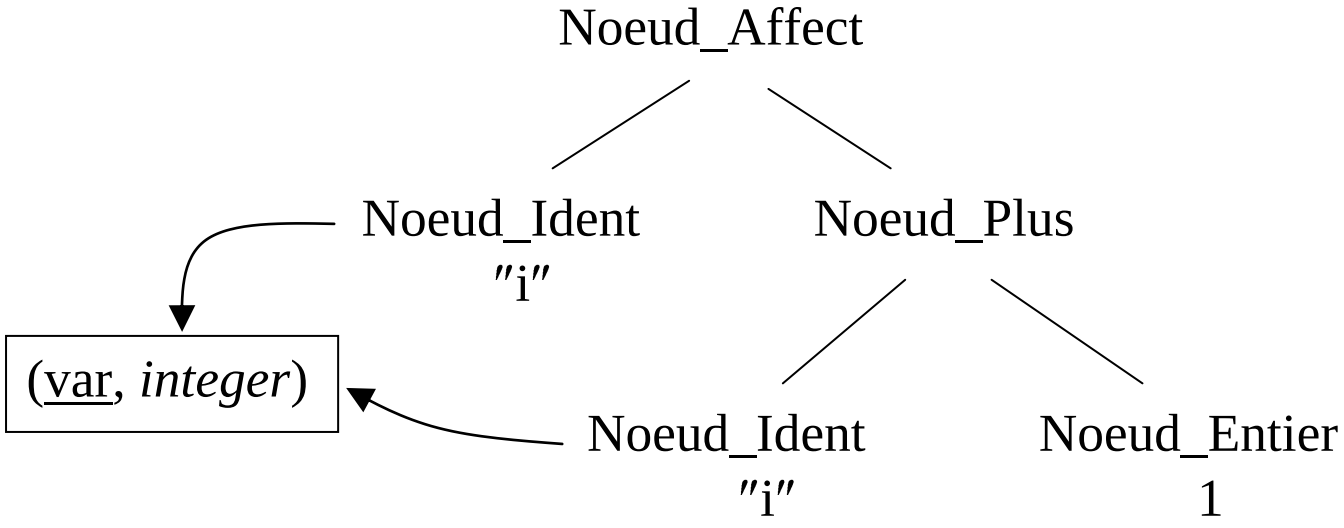
A chaque nœud de l'arbre est associé un décor. Un décor est un triplet :

```
(Def: Defn, Type: Exp_Type, Info_Code: Integer)
```

- `Def` est associé aux Noeud_Ident.
- `Type` est associé aux Noeud_Affect, Noeud_Conversion et à tous les nœuds qui dérivent de EXP dans la grammaire d'arbres.
- `Info_Code` servira en passe 3 pour calculer le nombre de registres nécessaires pour évaluer une expression.

Sémantique de partage : les Defn et Exp_Type sont partagées.

Par exemple, tous les Noeud_Ident correspondant au même identificateur sont décorés avec la même Defn.



3. Mise en œuvre de la passe 2

a) Paquetages fournis

- `Exp_Types` : paquetage permettant de manipuler des types du langage Cas ;
- `Defns` : paquetage permettant de manipuler des définitions.
  Remarque : une `Defn` est un triplet
  `(String, Nature_Defn, Exp_Type).`
- `Decors` : paquetage permettant de manipuler des décors.

b) Implémentation de la passe 2

La passe 2 est un parcours de l'arbre abstrait du programme. Lors de ce parcours :

- on vérifie que le programme Cas est contextuellement correct ;
- on ajoute des Noeud_Conversion ;
- on décore les différents nœuds de l'arbre.

Pour implémenter ce parcours d'arbre, on suit exactement la grammaire d'arbres. On écrit (au minimum) une procédure par non-terminal de la grammaire d'arbres.

```
procedure Verif_PROGRAMME(A : in Arbre) ;
procedure Verif_LISTE_INST(A : in Arbre) ;
procedure Verif_DECL(A : in Arbre) ;
...
```

Pour les identificateurs, il faut distinguer les déclarations et les utilisations d'identificateurs.

```
procedure Verif_IDENT_Decl(A : in Arbre; ...) ;
procedure Verif_IDENT_Util(A : in Arbre; ...) ;
```

On peut également définir d'autres procédures pour les différents nœuds de l'arbre.

Exemple :

```
procedure Verif_LISTE_INST(A : in Arbre) is
begin
   case Acces_Noeud(A) is
      when Noeud_Vide =>
         null ;
      when Noeud_Liste_Inst =>
         Verif_LISTE_INST(Acces_Fils1(A)) ;
         Verif_INST(Accces_Fils2(A)) ;
      when others =>
         Erreur_Interne(
"Arbre incorrect dans Verif_LISTE_INST") ;
   end case ;
end ;

procedure Verif_INST(A : in Arbre) is
begin
   case Acces_Noeud(A : in Arbre) is
      when Noeud_Nop => null ;
      when Noeud_Affect => Verif_Affect(A) ;
      when Noeud_Pour => Verif_Pour(A) ;
      ...
      when others =>
         Erreur_Interne(
"Arbre incorrect dans Verif_INST") ;
   end case ;
end ;
```

Pour cette étape, il est important de

- bien décomposer les problèmes en écrivant des procédures **courtes** ;
- **factoriser** les éléments communs (pas de copié-collé !) ;
- regrouper les procédures dans plusieurs paquetages (ex : déclarations, instructions, expressions).
  Dans les *spécifications* des paquetages, on déclare uniquement *ce qui est utilisé par d'autres paquetages* (types, fonctions, procédures)
- **compiler et tester au fur et à mesure** ;
- **conserver et documenter** tous les fichiers de test :
  - tests de non régression ; scripts permettant d'enchaîner les tests ;
  - commentaire indiquant le résultat du test (passe, erreur contextuelle ligne *n*)

Exercice Ecrire la procédure de construction de l'environnement prédéfini.

A faire

- `Erreur` : paquetage définissant une procédure qui affiche un message d'erreur.
- `Environ` : paquetage qui définit l'environnement.
- `Règles_Typage` : paquetage définissant des prédicats indiquant si deux types sont compatibles (pour une affectation, pour un opérateur binaire, pour un opérateur unaire).
- `Verif` : paquetage principal de la vérification contextuelle.
- Autres paquetages de vérifications.

A Rendre

- Programmes
- Jeux de tests
- Documentation décrivant :
  - les messages d'erreurs,
  - l'architecture de la passe 2,
  - compte rendu sur l'utilisation de Gcov (tests ajoutés, couverture obtenue).

Temps : deux fois plus que pour la passe 1.
Passe 1 : 200 lignes pour lexico.l
500 lignes pour syntax.y
Passe 2 : 1500 lignes