Grenoble INP
Esisar
4ème année
CS410 – CS422
2013 – 2014

Projet Compilation

et Génie Logiciel

Spécifications

Table des matières

~/ProjetCompil/Global/Doc :

Exemples.cas 1
Lexicographie.txt 2
Syntaxe.txt 4
Context.txt 6
ArbreAbstrait.txt 9
ArbreEnrichi.txt 12
MachineAbstraite.txt 15

```
--                          Deux exemples de programmes JCas

-- Fonction factorielle
program
   n, fact : integer ;
begin
   write("Entrer un entier : ") ;
   -- Lecture d'un entier
   read(n) ;
   -- Calcul de la factorielle
   fact := 1 ;
   while n >= 1 do
      fact := fact * n ;
      n := n - 1 ;
   end ;
   -- Affichage du resultat
   write("factorielle = ", fact) ;
   new_line ;
end.



-- Recherche dichotomique dans un tableau trie
program
   T : array [1 .. 20] of integer ;      -- Tableau d'entiers
   i : 1 .. 20 ;                         -- Indice de tableau
   min, max, milieu : 1 .. 20 ;          -- Indices de tableau
   val : integer ;                       -- Valeur cherchee
   trouve : boolean ;                    -- vrai lorsque la valeur est trouvee

begin
   -- Initialisation du tableau : pour tout i, T[i] = 2 * i
   for i := 1 to 20 do
      T[i] := 2 * i ;
   end ;

   -- Lecture de la valeur a rechercher
   write("Valeur a chercher dans le tableau :") ;
   read(val) ;

   -- Recherche dichotomique
   min := 1 ;
   max := 20 ;
   trouve := false ;
   while (min <= max) and not trouve do
      milieu := (min + max) div 2 ;
      if T[milieu] = val then
         trouve := true ;
      else
         if T[milieu] < val then
            min := milieu + 1 ;
         else
            max := milieu - 1 ;
         end ;
      end ;
   end ;

   -- Affichage du resultat
   if trouve then
      write("Trouve : T[", milieu, "] = ", T[milieu]) ;
   else
      write("Valeur ", val, " non trouvee") ;
   end ;
   new_line ;
end.
```

Lexicographie du langage JCas

1) Conventions

- Les éléments entre quotes (comme ’0’, ’.’) désignent les caractères ou chaînes correspondantes ; ce sont des terminaux du langage JCas.
- Les noms notés en majuscules (comme LETTRE, REEL) désignent des langages.
- Les opérateurs sur les langages utilisés sont les notations habituelles d’expressions régulieres.
- ’’ désigne la chaîne vide.
- On appelle "caractère imprimable" tout caractère dont le code ASCII (en octal) est dans l’intervalle [\040−\176].
  Le caractere ’ ’ a le code octal \040.
  Le caractere ’!’ a le code octal \041.
  Le caractere ’"’ a le code octal \042.

2) Identificateurs

LETTRE = {’a’,...,’z’,’A’,...,’Z’}

CHIFFRE = {’0’,...,’9’}

Les identificateurs sont définis par :

IDF = LETTRE (LETTRE + CHIFFRE + ’_’)*

Exception : les mots réservés ne sont pas des identificateurs.
Les majuscules ne sont pas pertinentes pour distinguer des identificateurs.

3) Constantes numériques

NUM = CHIFFRE CHIFFRE*

SIGNE = {’+’, ’−’, ’’}.

EXP (exposants) = ’E’ SIGNE NUM + ’e’ SIGNE NUM

DEC (décimaux positifs) = NUM ’.’ NUM

Les constantes numériques sont définies par :

- Constantes entières : INT = NUM
- Constantes réelles : REEL = DEC + DEC EXP

Exemples : ’12.0E+4’ ’0.5E2’ ’12.3’ ’12e3’
Contre−exemples : ’3.’ ’3E1.2’ ’12E 0’

4) Chaînes de caractères

CHAINE_CAR est l’ensemble de tous les caractères imprimables, a l’exception du caractère ’"’.

Les chaînes de caractères sont définies par :

CHAINE = ’"’ (CHAINE_CAR + ’""’)* ’"’

Exemple : ’"asd""ef "’ ; ’"’
Contre−exemples : ’"asdasf"v"’

5) Commentaires

Un commentaire est une suite de caractères imprimables et de tabulations qui commence par deux tirets adjacents et s'étend jusqu'à la fin de ligne.

6) Mots réservés

Les mots suivants (dits mots réservés) ne sont pas des identificateurs :

and	array	begin	div
do	downto	else	end
for	if	mod	new_line
not	null	of	or
program	read	then	to
while	write		

On peut mettre en majuscules tout ou partie d'un mot réservé.
Par exemple, 'ElsE' est équivalent a 'else'.

7) Symboles spéciaux

'<' '>' '=' '+' '–' '*' '/' '.' '[' ']' ',' ':' '(' ')' ';'

'..' ':=' '/=' '>=' '<='

8) Séparateurs

Espaces, fin de lignes, tabulations et commentaires sont des séparateurs.

Par exemple, ': =' n'est pas une affectation, mais la suite des deux symboles spéciaux ':' et '='.

Syntaxe hors-contexe du langage JCas

```
/* Terminaux (tokens retournés par l'analyseur lexical. */
terminal String IDF;          // Identificateur
terminal Integer CONST_ENT;   // Constante entière
terminal Float CONST_REEL;    // Constante réelle
terminal String CONST_CHAINE; // Constante chaîne

/* Mots réservés */
terminal   AND,         ARRAY,       BEGIN;
terminal   DIV,         DO,          DOWNTO;
terminal   ELSE,        END,         FOR;
terminal   IF,          MOD,         NEW_LINE;
terminal   NOT,         NULL,        OF;
terminal   OR,          PROGRAM,     READ;
terminal   THEN,        TO,          WHILE;
terminal   WRITE;

/* Opérateurs */
terminal INF, SUP, EGAL, DIFF, // "<", ">", "=", "/="
         INF_EGAL, SUP_EGAL,   // "<=", ">="
         PLUS, MOINS,          // "+", "-"
         MULT, DIV_REEL;       // "*", "/"

/* Autres terminaux */
terminal PAR_OUVR, PAR_FERM; // "(", ")"
terminal DOUBLE_POINT;       // ".."
terminal DEUX_POINTS;        // ":"
terminal VIRGULE;            // ","
terminal POINT_VIRGULE;      // ";"
terminal CROCH_OUVR;         // "["
terminal CROCH_FERM;         // "]"
terminal AFFECT;             // ":="
terminal POINT;              // "."


/*
   On définit les priorités des opérateurs
   Les priorités vont dans l'ordre croissant
   On déclare egalement l'associativité des opérateurs
   (left, right, ou nonassoc)
*/

precedence nonassoc INF, SUP, EGAL, DIFF, INF_EGAL, SUP_EGAL;
precedence left PLUS, MOINS, OR;
precedence left MULT, DIV_REEL, DIV, MOD, AND;
precedence nonassoc NOT;


/* Grammaire du langage JCas */

program ::= PROGRAM liste_decl BEGIN liste_inst END POINT
   ;

liste_decl ::= liste_decl decl POINT_VIRGULE
   |  // epsilon
   ;

decl ::= liste_idf DEUX_POINTS type
   ;

liste_idf ::= liste_idf VIRGULE idf
   |  idf
;

type ::= idf
   |  type_intervalle
   |  ARRAY CROCH_OUVR type_intervalle CROCH_FERM OF type
```

```
    ;

type_intervalle ::= constante DOUBLE_POINT constante
    ;

constante ::= PLUS const
    |  MOINS const
    |  const
    ;

const ::= CONST_ENT
    |  idf
    ;

idf ::= IDF
    ;

liste_inst ::= liste_inst inst POINT_VIRGULE
    |  inst POINT_VIRGULE
    ;

inst ::= NULL
    |  place AFFECT exp
    |  FOR pas DO liste_inst END
    |  WHILE exp DO liste_inst END
    |  IF exp THEN liste_inst END
    |  IF exp THEN liste_inst ELSE liste_inst END
    |  WRITE PAR_OUVR liste_exp PAR_FERM
    |  READ PAR_OUVR place PAR_FERM
    |  NEW_LINE
    ;

pas ::= idf AFFECT exp TO exp
    |  idf AFFECT exp DOWNTO exp
    ;

liste_exp ::= liste_exp VIRGULE exp
    |  exp
    ;

exp ::= facteur
    |  exp AND exp
    |  exp OR exp
    |  exp EGAL exp
    |  exp INF_EGAL exp
    |  exp SUP_EGAL exp
    |  exp DIFF exp
    |  exp INF exp
    |  exp SUP exp
    |  exp PLUS exp
    |  exp MOINS exp
    |  exp MULT exp
    |  exp DIV_REEL exp
    |  exp MOD exp
    |  exp DIV exp
    |  PLUS facteur
    |  MOINS facteur
    |  NOT facteur
    ;

facteur ::= CONST_ENT
    |  CONST_REEL
    |  CONST_CHAINE
    |  place
    |  PAR_OUVR exp PAR_FERM
    ;

place ::= idf
    |  place CROCH_OUVR exp CROCH_FERM
    ;
```

Contraintes contextuelles du langage JCas

1) Les types du langage JCas et leur représentation

Les types du langage JCas sont représentés par des objets de la classe Type.
(cf. Type.java).

```
TYPE            -> Type.Real
                |  Type.Boolean
                |  Type.String
                |  INTERVALLE
                |  Type.Array(INTERVALLE, TYPE)

INTERVALLE      -> Type.Interval(int, int)
```

Le type 'Type.Interval(i1, i2)' représente l'intervalle (fermé) des entiers compris entre i1 et i2.

Exemples :
- Le type JCas '1 .. max_int' est représenté par le type 'Type.Interval(1, valmax)'.
- Le type JCas 'integer' est représenté par 'Type.Integer', qui a pour valeur 'Type.Interval(−valmax, valmax)', où valmax = java.lang.Integer.MAX_VALUE.

Notation : dans la suite, on notera 'Type.Interval' un type Type.Interval(i1, i2), où i1 et i2 sont des entiers quelconques.

Le type 'Type.Boolean' represente les booléens.

Le type 'Type.Real' represente les réels.

Le type 'Type.Array(E1, E2)' représente le type tableau dont les indices appartiennent au type E1 (qui doit être un type Type.Interval), et les éléments appartiennent au type E2.

Le type 'Type.String' représente le type des chaînes de caractères.
Remarque : dans le langage JCas, les seules chaînes de caractères manipulées sont des littéraux (comme dans l'instruction 'write("ok")'), ce qui explique qu'il n'existe pas d'identificateur pour ce type en JCas.

2) Identificateurs prédéfinis

Les identificateurs suivants sont prédéfinis dans le langage JCas :

```
integer, boolean, real (identificateurs de type)
true, false, max_int   (identificateurs de constante)
```

3) Règles de visibilité

Les identificateurs prédéfinis sont considérés comme des identificateurs déjà visibles avant l'analyse du programme. Ils ne peuvent pas être redéfinis. Les identificateurs déclarés dans le programme ne peuvent pas être redéclares. Les identificateurs qui apparaissent dans le programme et qui ne sont pas des identificateurs prédéfinis doivent avoir été déclarés.

Les identificateurs sont de différentes natures :
- identificateur de constante (entière, booléenne, réelle ou chaîne),
- identificateur de type,
- identificateur de variable.

On ne peut déclarer dans le programme que des identificateurs de variables.

Vérifications à effectuer (avec les notations de l'arbre abstrait)

Règle de l'arbre	Vérification à effectuer
TYPE −> IDENT	IDENT est un identificateur de type
EXP_CONST −> IDENT	IDENT est un identificateur de constante
EXP −> IDENT	IDENT est un identif. de constante ou de variable
PLACE −> IDENT	IDENT est un identificateur de variable

4) Profil des opérateurs

not :	Type.Boolean −> Type.Boolean
and, or :	Type.Boolean, Type.Boolean −> Type.Boolean
=, <, >, /= <=, >= :	Type.Interval, Type.Interval −> Type.Boolean Type.Interval, Type.Real −> Type.Boolean Type.Real, Type.Interval −> Type.Boolean Type.Real, Type.Real −> Type.Boolean
+, − :	Type.Interval −> Type.Integer Type.Real −> Type.Real
+, −, * :	Type.Interval, Type.Interval −> Type.Integer Type.Interval, Type.Real −> Type.Real Type.Real, Type.Interval −> Type.Real Type.Real, Type.Real −> Type.Real
div, mod :	Type.Interval, Type.Interval −> Type.Integer
/ :	Type.Interval, Type.Interval −> Type.Real Type.Interval, Type.Real −> Type.Real Type.Real, Type.Interval −> Type.Real Type.Real, Type.Real −> Type.Real
[] :	Array(Type.Interval, <type>), Type.Interval −> <type>

5) Vérifications de types

− Intervalles : <exp_const1> .. <exp_const2>
Il faut vérifier que <exp_const1> et <exp_const2> sont de type Type.Interval.

− Affectations : <place> := <expression>
Le type de la place et le type de l'expression doivent etre compatibles pour l'affectation, c'est−à−dire :

. <place> et <expression> de type Type.Interval (pas forcément avec les mêmes bornes) ;

. <place> et <expression> de type Type.Real ;

. <place> et <expression> de type Type.Boolean ;

. <place> de type Type.Real et <expression> de type Type.Interval ;

. <place> et <expression> de type Array, les types des indices étant identiques (plus précisement, de type 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 Type.Boolean.

− Instruction for : la variable de contrôle, ainsi que les deux expressions

doivent être de type Type.Interval.

- Instruction read : la place doit etre de type Type.Interval ou Type.Real.
- Instruction write : les expressions doivent être de type Type.Interval, Type.Real ou Type.String.
- Les places et expressions doivent être bien typées vis-à-vis des déclarations et des profils des opérateurs.

Syntaxe abstraite pour le langage JCas

La syntaxe abstraite du langage JCas est décrite à l'aide d'une grammaire d'arbres.

L'axiome de la grammaire est le non-terminal PROGRAMME.

Les règles de la grammaire sont de la forme :

C --> A1 | A2 | ... | An (avec n >= 1)

où :
- C est le non-terminal correspondant à une classe d'arbres (toujours noté en majuscules) ;
- les Ai sont des arbres.

Chaque Ai est noté :

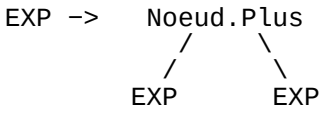
nd(C1, C2, ..., Cp) (avec p >= 0)

où :
- nd est un nom de noeud d'arité p ;
- les Ci sont des classes d'arbres (non-terminaux de la grammaire d'arbres).

Par exemple, la règle

EXP -> Noeud.Plus(EXP, EXP)

peut être représentée graphiquement par :



Tous les noeuds de l'arbre possèdent un attribut numligne (numéro de ligne du texte correspondant dans le fichier source).

Certaines feuilles de l'arbre possèdent un attribut supplémentaire :

Noeud.Ident : attribut Chaine ;
Noeud.Entier : attribut Entier ;
Noeud.Reel : attribut Reel ;
Noeud.Chaine : attribut Chaine.

D'autres attributs seront définis ultérieurement (pour la passe de vérification contextuelle).

Un noeud de l'arbre n'a pas de correspondance dans la syntaxe concrete : c'est le noeud Noeud.Conversion (qui sera utilisé pour dénoter la conversion d'une expression entière en réel).

Exemple :

```
 1  -- Exemple de programme JCas
 2  program
 3     T : array[1 .. 5] of array[1 .. 10] of integer ;
 4     i : 1 .. 5 ;
 5  begin
 6     for i := 1 to 5 do
 7        T[i][1] := 0 ;
 8     end ;
 9     write("ok") ;
10     new_line ;
11  end.
```

Arbre (presenté sous forme préfixée avec indentation selon la profondeur)

```
Noeud.Programme                                          -- ligne : 2
. Noeud.ListeDecl                                        -- ligne : 4
. . Noeud.ListeDecl                                      -- ligne : 3
. . . Noeud.Vide                                         -- ligne : 3
. . . Noeud.Decl                                         -- ligne : 3
. . . . Noeud.ListeIdent                                 -- ligne : 3
. . . . . Noeud.Vide                                     -- ligne : 3
. . . . . Noeud.Ident "t"                                -- ligne : 3
. . . . Noeud.Tableau                                    -- ligne : 3
. . . . . Noeud.Intervalle                               -- ligne : 3
. . . . . . Noeud.Entier 1                               -- ligne : 3
. . . . . . Noeud.Entier 5                               -- ligne : 3
. . . . . Noeud.Tableau                                  -- ligne : 3
. . . . . . Noeud.Intervalle                             -- ligne : 3
. . . . . . . Noeud.Entier 1                             -- ligne : 3
. . . . . . . Noeud.Entier 10                            -- ligne : 3
. . . . . . Noeud.Ident "integer"                        -- ligne : 3
. . Noeud.Decl                                           -- ligne : 4
. . . Noeud.ListeIdent                                   -- ligne : 4
. . . . Noeud.Vide                                       -- ligne : 4
. . . . Noeud.Ident "i"                                  -- ligne : 4
. . . Noeud.Intervalle                                   -- ligne : 4
. . . . Noeud.Entier 1                                   -- ligne : 4
. . . . Noeud.Entier 5                                   -- ligne : 4
. Noeud.ListeInst                                        -- ligne : 10
. . Noeud.ListeInst                                      -- ligne : 9
. . . Noeud.ListeInst                                    -- ligne : 6
. . . . Noeud.Vide                                       -- ligne : 9
. . . . Noeud.Pour                                       -- ligne : 6
. . . . . Noeud.Increment                                -- ligne : 6
. . . . . . Noeud.Ident "i"                              -- ligne : 6
. . . . . . Noeud.Entier 1                               -- ligne : 6
. . . . . . Noeud.Entier 5                               -- ligne : 6
. . . . . Noeud.ListeInst                                -- ligne : 7
. . . . . . Noeud.Vide                                   -- ligne : 8
. . . . . . Noeud.Affect                                 -- ligne : 7
. . . . . . . Noeud.Index                                -- ligne : 7
. . . . . . . . Noeud.Index                              -- ligne : 7
. . . . . . . . . Noeud.Ident "t"                        -- ligne : 7
. . . . . . . . . Noeud.Ident "i"                        -- ligne : 7
. . . . . . . . Noeud.Entier 1                           -- ligne : 7
. . . . . . . . Noeud.Entier 0                           -- ligne : 7
. . . Noeud.Ecriture                                     -- ligne : 9
. . . . Noeud.ListeExp                                   -- ligne : 9
. . . . . Noeud.Vide                                     -- ligne : 9
. . . . . Noeud.Chaine "ok"                              -- ligne : 9
. . Noeud.Ligne                                          -- ligne : 10
```

```
Grammaire d'arbres définissant la syntaxe abstraite du langage JCas
-------------------------------------------------------------------

PROGRAMME       -> Noeud.Programme(LISTE_DECL, LISTE_INST)

LISTE_DECL      -> Noeud.ListeDecl(LISTE_DECL, DECL)
                |  Noeud.Vide

DECL            -> Noeud.Decl(LISTE_IDENT, TYPE)

LISTE_IDENT     -> Noeud.ListeIdent(LISTE_IDENT, IDENT)
                |  Noeud.Vide

IDENT           -> Noeud.Ident                  -- attribut de type Chaine

TYPE            -> IDENT
                |  TYPE_INTERVALLE
                |  Noeud.Tableau(TYPE_INTERVALLE, TYPE)

TYPE_INTERVALLE -> Noeud.Intervalle(EXP_CONST, EXP_CONST)

EXP_CONST       -> IDENT
                |  Noeud.Entier                 -- attribut de type Entier
                |  Noeud.PlusUnaire(EXP_CONST)
                |  Noeud.MoinsUnaire(EXP_CONST)

LISTE_INST      -> Noeud.Vide
                |  Noeud.ListeInst(LISTE_INST, INST)

INST            -> Noeud.Nop
                |  Noeud.Affect(PLACE, EXP)
                |  Noeud.Pour(PAS, LISTE_INST)
                |  Noeud.TantQue(EXP, LISTE_INST)
                |  Noeud.Si(EXP, LISTE_INST, LISTE_INST)
                |  Noeud.Lecture(PLACE)
                |  Noeud.Ecriture(LISTE_EXP)
                |  Noeud.Ligne

PAS             -> Noeud.Increment(IDENT, EXP, EXP)
                |  Noeud.Decrement(IDENT, EXP, EXP)

PLACE           -> IDENT
                |  Noeud.Index(PLACE, EXP)

LISTE_EXP       -> Noeud.Vide
                |  Noeud.ListeExp(LISTE_EXP, EXP)

EXP             -> Noeud.Et(EXP, EXP)
                |  Noeud.Ou(EXP, EXP)
                |  Noeud.Egal(EXP, EXP)
                |  Noeud.InfEgal(EXP, EXP)
                |  Noeud.SupEgal(EXP, EXP)
                |  Noeud.NonEgal(EXP, EXP)
                |  Noeud.Inf(EXP, EXP)
                |  Noeud.Sup(EXP, EXP)
                |  Noeud.Plus(EXP, EXP)
                |  Noeud.Moins(EXP, EXP)
                |  Noeud.Mult(EXP, EXP)
                |  Noeud.DivReel(EXP, EXP)
                |  Noeud.Reste(EXP, EXP)
                |  Noeud.Quotient(EXP, EXP)
                |  Noeud.Index(PLACE, EXP)
                |  Noeud.PlusUnaire(EXP)
                |  Noeud.MoinsUnaire(EXP)
                |  Noeud.Non(EXP)
                |  Noeud.Conversion(EXP)
                |  Noeud.Entier                 -- attribut de type Entier
                |  Noeud.Reel                   -- attribut de type Reel
                |  Noeud.Chaine                 -- attribut de type Chaine
                |  IDENT
```

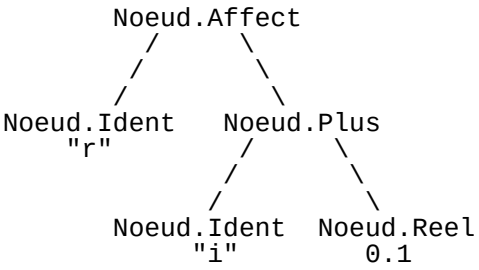
Enrichissement et décoration de l'arbre en passe 2

1) Enrichissement de l'arbre : ajouts de Noeud.Conversion

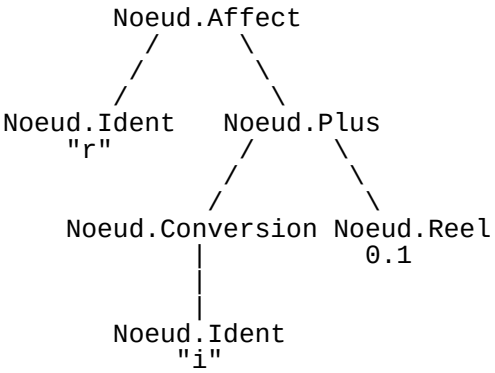
Ce noeud indique qu'il y a une conversion du type entier au type réel dans un programme JCas.

Soit le contexte de déclarations r : real ; i : integer.

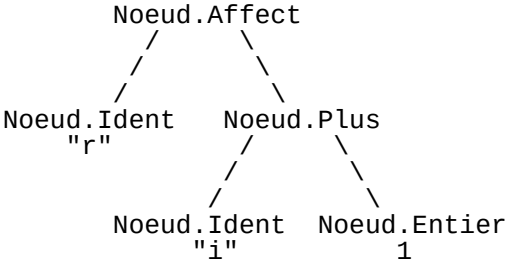
L'instruction r := i + 0.1 est représentée par l'arbre primitif :



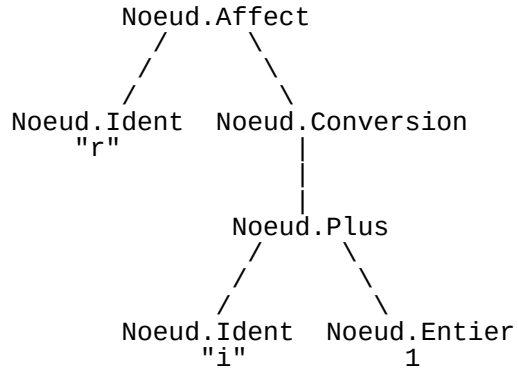
Après enrichissement, on obtient l'arbre :



L'instruction r := i + 1 est représentée par l'arbre primitif :



Après enrichissement, on obtient l'arbre :



2) Décoration de l'arbre

Un décor est un triplet
(Defn defn, Type type, int infoCode)

. 'defn' n'est rattaché qu'aux noeuds 'Noeud.Ident'.

. 'type' n'est rattaché qu'aux noeuds 'Noeud.Affect', 'Noeud.Conversion', et à tous les noeuds obtenus par la règle EXP de la grammaire d'arbres (voir ArbreAbstrait.txt).

Remarque :
'type' sera rattaché aux noeuds 'Noeud.Ident' correspondant à un identificateur apparaissant dans les instructions.

. 'infoCode' peut etre rattaché à tous les noeuds de l'arbre.

Remarque :
'infoCode' servira uniquement dans la passe de génération de code (par exemple, nombre de registres nécessaires à l'évaluation d'une expression)

Voir les spécifications détaillees dans Decor.java, Defn.java et Type.java.

On utilise une "sémantique de partage" : les defns et les types sont partagés autant que possible.
Par exemple, les noeuds 'Noeud.Ident' correspondant à la même définition ont tous la même defn et le même type.

3) Exemple complet

```
1  -- Exemple de programme JCas
2  program
3     T : array[1 .. 5] of real ;
4  begin
5     T[1] := 0 ;
6  end.
```

Arbre abstrait enrichi et décoré correspondant :

(Remarque : le champ 'Taille' des types sera mis a jour lors de
la passe 3.)

```
Noeud.Programme                                                  -- ligne : 2
. Noeud.ListeDecl                                                -- ligne : 3
. . Noeud.Vide                                                   -- ligne : 3
. . Noeud.Decl                                                   -- ligne : 3
. . . Noeud.ListeIdent                                           -- ligne : 3
. . . . Noeud.Vide                                               -- ligne : 3
. . . . Noeud.Ident "t"                                          -- ligne : 3
. . . . -- defn :
. . . . --    (NatureDefn.Var, NatureType.Array(NatureType.Interval(1, 5),
NatureType.Real))
. . . Noeud.Tableau                                              -- ligne : 3
. . . . Noeud.Intervalle                                         -- ligne : 3
. . . . . Noeud.Entier 1                                         -- ligne : 3
. . . . . Noeud.Entier 5                                         -- ligne : 3
. . . . Noeud.Ident "real"                                       -- ligne : 3
. . . . -- defn :
. . . . --    (NatureDefn.Type, NatureType.Real)
. Noeud.ListeInst                                                -- ligne : 5
. . Noeud.Vide                                                   -- ligne : 6
. . Noeud.Affect                                                 -- ligne : 5
. . -- type :
. . --    NatureType.Real
. . . Noeud.Index                                                -- ligne : 5
. . . -- type :
. . . --    NatureType.Real
. . . . Noeud.Ident "t"                                          -- ligne : 5
. . . . -- defn :
. . . . --    (NatureDefn.Var, NatureType.Array(NatureType.Interval(1, 5),
NatureType.Real))
. . . . -- type :
. . . . --    NatureType.Array(NatureType.Interval(1, 5), NatureType.Real)
. . . . Noeud.Entier 1                                           -- ligne : 5
. . . . -- type :
. . . . --    Type.Integer
. . . Noeud.Conversion                                           -- ligne : 5
. . . -- type :
. . . --    NatureType.Real
. . . . Noeud.Entier 0                                           -- ligne : 5
. . . . -- type :
. . . . --    Type.Integer
```

Définition de la Machine Abstraite et de son langage d'assemblage

Les types des valeurs manipulées sont les entiers, les flottants, les adresses.

La "mémoire physique" (sous ce terme sont englobés registres, caches, RAM...) de la machine est logiquement partagée en 3 zones :

- La zone registres. Elle est constituée des registres banalisés R0 .. R15. Ils peuvent contenir des valeurs de tout type, et peuvent etre lus ou modifiés.
- La zone code. Elle contient les instructions du programme. A cette zone est associé un registre spécialisé, PC (compteur ordinal), qui contient les adresses successives des instructions à exécuter. PC ne peut etre ni lu ni modifié explicitement. PC+1 (resp. PC−1) est l'adresse de l'instruction suivant (resp. précédant) celle d'adresse PC.
- La zone pile, qui comprend N mots (N n'est pas fixé a priori). Ils peuvent contenir des valeurs de tout type, et peuvent etre lus ou modifiés. A chaque mot est associée une adresse. A cette zone sont associés trois registres spécialisés, qui ne peuvent contenir que des adresses.
  GB (base globale) : contient à tout instant l'adresse précédant celle du premier mot de la pile.
  LB (base locale).
  SP (pointeur de pile).
  Les adresses des mots de la pile sont comprises entre GB+1 et GB+N.

Les éléments de mémorisation (registres banalisés et mots mémoires) sont "typés" dynamiquement. Initialement, tout est "indéfini" ; lors d'une modification d'un élément, le type de données est aussi mémorise. Lors d'une opération, il y a vérification de compatibilité de type.

Avant l'exécution de la première instruction :
- Le contenu de la pile ainsi que des registres R0 .. Rn est indéfini.
- GB = LB = SP sont initialisés à la même valeur (par le chargeur).
- PC est initialisé (par le chargeur) à l'adresse de la première instruction à exécuter dans la zone code.

On dispose de 4 modes d'adressage :
* registre direct : Rm (m dans 0 .. n)
* registre indirect avec déplacement : d(XX), ou d est entier et XX = Rm (qui doit contenir une adresse), GB ou LB.
  d(XX) désigne l'adresse (contenu de XX)+d.
* registre indirect avec déplacement et index : d(XX, Rm), où d est entier, XX = GB, LB ou Rp (qui doit contenir une adresse), et Rm doit contenir un entier.
  d(XX,Rm) désigne l'adresse (contenu de XX)+(contenu de Rm)+d.
  Pour les deux modes d'adressage indirect, le déplacement d est un entier en notation Cas, éventuellement précédé d'un signe + ou −.
* immédiat : #d, ou d est entier ou réel, en notation Cas, éventuellement précédé d'un signe + ou −.

Les instructions (dénotées InstructionMA dans la grammaire ci−dessous) sont de plusieurs catégories :
- transfert de données
- opérations arithmétiques
- contrôle
- entrées−sorties
- divers

Les instructions ont 0, 1 ou 2 opérandes (source puis destination).
Avant l'exécution de chaque instruction, PC est incrémente de 1.

Dans ce qui suit :

dadr	==	d(XX)	d(XX, Rm)		(désignation d'adresse)
dval	== Rm	d(XX)	d(XX, Rm)	#d	(désignation de valeur)

Les notations d(XX) et d(XX, Rm) ne sont autorisées pour des dval que si l'adresse désignée est une adresse d'un mot de la pile.

```
  +-------------------------------------------------------------------+
  | C[XX]    ==  contenu du registre XX (XX = Rm, SP, LB, GB)         |
  | C[@]     ==  contenu du mot d'adresse @ dans la pile              |
  |              (@ doit être dans C[GB]+1 .. C[GB]+N)                |
  +-------------------------------------------------------------------+
  +--------------------------------+ +--------------------------------+
  |         ADRESSES               | |          VALEURS               |
  | A[d(XX)]    = C[XX] + d        | | V[XX]       = C[XX]            |
  | A[d(XX,Rm)] = C[XX] + C[Rm] + d | | V[dadr]     = C[A[dadr]]       |
  +--------------------------------+ | V[#d]       = d                |
                                     +--------------------------------+
  V[dadr] n'a de sens que si A[dadr] est dans l'intervalle C[GB]+1 .. C[GB]+N,
  c'est-à-dire si l'adresse associée à dadr est l'adresse d'un mot de la pile.

  L <- Val est une affectation : la valeur Val est rangée dans L.
  L est soit un registre, soit l'adresse d'un mot de la pile.
  Dans ce dernier cas, Val est rangée dans le mot d'adresse L.

Les codes condition cc sont :
   EQ (égal)                      NE (différent)
   GT (strictement supérieur)     LT (strictement inférieur)
   GE (supérieur ou égal)         LE (inférieur ou égal)
   OV (débordement)
Ils sont positionnés à vrai ou faux par certaines instructions.

Les codes de comparaison EQ, NE, GT, LT, GE, LE sont toujours positionnés
simultanément, et leurs valeurs satisfont toujours les axiomes :
  NE == non EQ
  LT == NE et non GT    GT == NE et non LT
  LE == LT ou EQ        GE == GT ou EQ
La valeur initiale des codes condition est indéterminée, mais elle satisfait
les axiomes.

Pour indiquer qu'une instruction positionne les codes condition,
on écrit "CC :" suivi de CP et/ou OV (CP pour ComParaison).
En général, la valeur des codes de comparaison est relative à la comparaison
à #0 (pour les entiers) ou #0.0 (pour les flottants), du résultat de
l'opération pour les instructions arithmétiques, ou de la valeur de la source
pour les autres.
Lorsqu'une instruction positionne OV à vrai, son effet est indéterminé.

- transfert de données
   LOAD dval, Rm   : Rm <- V[dval]                        CC : OV faux, CP
   STORE Rm, dadr  : A[dadr] <- V[Rm]                     CC : OV faux, CP
   LEA dadr, Rm    : Rm <- A[dadr]
   PEA dadr        : SP <- V[SP] + 1 ; V[SP] <- A[dadr]
   PUSH Rm         : SP <- V[SP] + 1 ; V[SP] <- V[Rm]     CC : OV faux, CP
   POP Rm          : Rm <- V[V[SP]] ;  SP <- V[SP] - 1    CC : OV faux, CP

- opérations arithmétiques (soit entre entiers, soit entre flottants)
   ADD dval, Rm    : Rm <- V[Rm] + V[dval]             CC : CP, OV
   SUB dval, Rm    : Rm <- V[Rm] - V[dval]             CC : CP, OV
   OPP dval, Rm    : Rm <- - V[dval]                   CC : OV faux, CP
   MUL dval, Rm    : Rm <- V[Rm] * V[dval]             CC : CP, OV
   CMP dval, Rm    : mise a jour des codes condition   CC : OV faux, CP
                     selon V[Rm] - V[dval] (ex. GT := V[Rm] - V[dval] > 0)

- opérations arithmétiques spécifiques aux entiers
   DIV dval, Rm    : Rm <- V[Rm] div V[dval]    CC : CP, OV vrai ssi V[dval] = 0
   MOD dval, Rm    : Rm <- V[Rm] mod V[dval]    CC : CP, OV vrai ssi V[dval] = 0
   FLOAT dval, Rm  : conversion entier->reel    CC : OV vrai ssi V[dval] non
                     Rm <- IntToFloat(V[dval])       codable sur un flottant
   Scc Rm          : si (cc = vrai) alors Rm <- 1 sinon Rm <- 0

- opérations arithmétiques spécifiques aux flottants
   DIV dval, Rm    : Rm <- V[Rm] / V[dval]      CC : CP, OV vrai ssi débordement
   INT dval, Rm    : conversion reel->entier    CC : OV vrai ssi V[dval] non
                                                     codable sur un entier
                     Rm <- Signe(V[dval]) * PartieEntière(ValAbsolue(V[dval]))

- contrôle
```

```
   BRA etiq        : branchement inconditionnel
                          PC <- (@ de l'instruction qui suit etiq)
   Bcc etiq        : branchement conditionnel
                     si (cc = vrai) alors
                        PC <- (@ de l'instruction qui suit etiq)
   BSR etiq        : SP <- V[SP] + 2 ; V[SP]-1 <- V[PC] ; V[SP] <- V[LB] ;
                     LB <- V[SP] ; PC <- (@ de l'instruction qui suit etiq)
   RTS             : PC <- C[V[LB]-1] ; SP <- V[LB]-2 ; LB <- C[V[LB]] ;

- entrées-sorties
   RINT            : R1 <- entier lu         CC : CP, OV vrai ssi débordement
                                                  ou erreur de syntaxe
   RFLOAT          : R1 <- flottant lu       CC : CP, OV (cf RINT)
   WINT            : écriture de l'entier V[R1]
   WFLOAT          : écriture du flottant V[R1]
   WSTR "..."      : écriture de la chaîne (meme notation qu'en langage Cas)
   WNL             : écriture newline

- divers
   ADDSP #d        : SP <- V[SP] + d
                     d doit être un entier naturel, avec ou sans le signe +.
   SUBSP #d        : SP <- V[SP] - d
                     d doit être un entier naturel, avec ou sans le signe +.
   TSTO #d         : test débordement pile.                    CC : OV
                     si V[SP] + d > V[GB] + N alors OV := vrai
                     d doit être un entier naturel, avec ou sans le signe '+'.
   HALT            : arrêt du programme


Temps d'exécution des instructions (en nombre de cycles internes) :

LOAD     2                          INT      4
STORE    2                          BRA      5
LEA      0                          Bcc      5 (cc vrai) 4 (cc faux)
PEA      4                          BSR      9
PUSH     4                          RTS      8
POP      2                          RINT    16
ADD      2                          RFLOAT  16
SUB      2                          WINT    16
OPP      2                          WFLOAT  16
MUL     20                          WSTR    16
CMP      2                          WNL     14
DIV     79 (entiers)                ADDSP    4
DIV     40 (reels)                  SUBSP    4
MOD     79                          TSTO     4
FLOAT    4                          HALT     1
Scc      3 (cc vrai) 2 (cc faux)


Il faut ajouter le cas écheant le temps d'accès aux opérandes :

Modes d'adressage  Temps
       Rm            0
     d(XX)           4
     d(XX,Rm)        5
       #d            2
     "..."           2 * longueur de la chaîne

Notes sur le langage d'assemblage :
  - l'espace et la tabulation sont des séparateurs.
  - on peut insérer des lignes blanches où on veut.
  - les commentaires sont constitués du caractère ';' et du reste de la ligne
    (caractères imprimables et tabulations).
  - une étiquette est positionnée en faisant suivre son nom de ':'.
  - on place une instruction par ligne, éventuellement suivie de commentaires.
Les codes opération et les noms des registres peuvent être en majuscules ou
minuscules.
Il est d'usage d'indenter les instructions par rapport aux étiquettes.

La syntaxe est la suivante (notation Ayacc) :
```

```
  Programme   :
              |  Programme Ligne
              ;
  Ligne       : ListeEtiq Instruction '\n'
              ;
  ListeEtiq   :
              | ListeEtiq etiq ':'
              ;
  Instruction :
              | InstructionMA       -- cf instructions ci-dessus
              ;
```

La lexicographie des étiquettes et des commentaires est la suivante (notation Aflex)

```
  LETTRE         [a-zA-Z]
  CHIFFRE        [0-9]
  ETIQUETTE      {LETTRE}({LETTRE}|{CHIFFRE}|_|".")*
  COMM_CAR       [\t\040-\176] -- caracteres imprimables et tabulation
  COMMENTAIRE    ";"{COMM_CAR}*
```

Exemples d'etiquettes: Ceci_Est_1_etiquette.0 En.Voici_42._autres_

On ne distingue pas majuscules et minuscules. Une étiquette ne doit pas avoir un nom de code-opération ou de registre.

Exemple : la factorielle itérative

```
; ========================================================================
     ADDSP #2
; -------------------------- Writeln ligne 5 --------------------------
     WSTR "Entrer un entier : "
     RINT
     STORE R1, 1(GB)
; ------------------------ Affectation, ligne 9 ------------------------
     LOAD #1, R0
     STORE R0, 2(GB)
; --------------------- Boucle tant que, ligne 10 ----------------------
     BRA etiq.2
etiq.1 :
; ----------------------- Affectation, ligne 11 ------------------------
     LOAD 2(GB), R0
     MUL 1(GB), R0
     BOV erreur_debordement
     STORE R0, 2(GB)
; ----------------------- Affectation, ligne 12 ------------------------
     LOAD 1(GB), R0
     SUB #1, R0
     STORE R0, 1(GB)
etiq.2 :
     LOAD 1(GB), R0
     CMP #1, R0
     BGE etiq.1
; -------------------------- Writeln ligne 15 -------------------------
     WSTR "factorielle = "
     LOAD 2(GB), R1
     WINT
; -------------------------- New_Line ligne 16 -------------------------
     WNL
     HALT
; ------------------------ Erreurs a l'execution ----------------------
erreur_debordement :
     WSTR "Erreur a l'execution : debordement arithmetique"
     WNL
     HALT
; ========================================================================
```