

Prolog et Traitement Automatique des Langues

Éric de la Clergerie
Eric.De_La_Clergerie@inria.fr

ALPAGE – INRIA
http://alpage.inria.fr

Cours M2 LI 2008

Quinzième partie XV

Méta-Grammaires

Principe

Introduite par **Candito**

- Description de grammaires comme des ensembles (**classes**) de contraintes à satisfaire
- Ajout de la notion d'**héritage** et de **croisement** de classe pour factoriser l'information
⇒ modularité
- Satisfaction partielle des contraintes hors-ligne pour construire les structures d'une grammaire TAG (ou LFG, ...)
⇒ éviter les risques d'explosion combinatoire liés aux contraintes

Les Méta-Grammaires par l'exemple

Definition (Méta-Grammaire)

Description modulaire par **classes** regroupant des **contraintes**, avec **héritage**

```
class collect_real_subject_canonical {
  <: collect_real_subject;
  $arg.extracted = value(~ cleft);
  S >> VSubj; V >> psubj;
  VSubj < V; VMod < psubj;
  node psubj: [cat:N2, id:subject,
               top:[wh:-, sat:+]];
  - psubj::agreement; psubj = psubj::N;
  psubj =>
  node(Infl).bot.inv = value(+),
  $arg.extracted = value(-),
  $arg.real = value(N2),
  desc.extraction = value(~-),
  node(V).top.mode = value(~ inf | imp | ...);
  ~psubj=> node(Infl).bot.inv = value(~+);
}
```

- Héritage (<:)
- Contraintes
 - ▶ dominance (>> et >>>+)
 - ▶ précédence (<)
 - ▶ égalité (=)
 - ▶ Décorations (FS)
 - ★ noeuds
 - ★ classe
 - ▶ Éq. entre chemins (.)
 - ★ noeuds (**node** psubj)
 - ★ classe (**desc**)
 - ★ variable (**\$arg**)
- Ressources + / Besoins -
 - ▶ Espace de noms (::)
- Gardes (=>)

Format DyALog (à la Prolog) : comme entrée pour **MGC**OMP

```
class('adj_on_noun').
super('adj_on_noun', 'adj_as_modifier').
equation('adj_on_noun', desc:ht:arg0:real, value('N2')).
node('adj_on_noun', 'Root').
nodefeature('adj_on_noun', 'Root', [cat: 'N']).
```

Exemple de garde :

```
guard('verb_extraction_relative',
      'prel_object', +, [
        and([node('XGroup') : extracted : real = value(prel),
             node('XGroup') : extracted : kind = value(obj),
             node('XGroup') : extracted : pcas = value(-)])
      ]).
```

Contraintes sur les structures de traits

- valeur atomique, multiple et négation : `value(~ncadj|np|)`
- Affectation sur un noeud : `node N: [cat: nc, top: $top ^ [gender:masc]]`
- Affectation sur la classe : `desc([ht: [cat: nc]])`
- Équations entre chemins :
 - ▶ ancré sur un noeud `node(N).top.gender=value(masc)`
 - ▶ ancré sur la classe `desc.ht.cat=node(N).cat`
 - ▶ ancré sur une variable `$top.number = value(sing)`
- Macros sur les valeurs et chemins

```
%% Macro for value
template @emptyarg_fs = [pcas: -, kind:--, real:--, extracted:--]
%% Macro for pathes
path @arg0 = .ht.arg0
node(N).@real0 = value(@emptyarg_fs);
```

Note : Pas de vérifications de cohérences sur les structures (valeurs admissibles pour un trait).

mais possibilité d'écrire : `node(N).top.cat = value(@cat^nc|np)`

- Dominance immédiate : `Father >> Son`
- Dominance strict : `Ancestor >>> Descendant`
- Précédence : `Left < Right`
- Aliasing : `Alias1 = Alias2`

Note : pas de dominance reflexive ni de précédence immédiate

Structure des noeuds

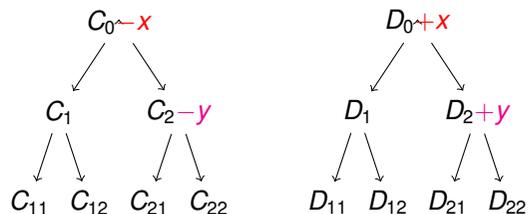
Attributs standards sur un noeud :

- **type** : dans `std`, `subst`, `foot`, `anchor`, `coanchor`, `lex`, `sequence`, `alternative`
- **id** : identification du noeud
- **cat** : catégorie syntaxique
- **lex** : forme (pour les noeuds de type lex)
- **top** et **bot** : arguments TAG
- **adj** : dans `yes`, `no`, `strict`
- **optional** : optionalité + -
- **star** : répétition * -

Noeud alternative :

```
node XGroup : [type: alternative];
XGroup >> pri_subj;
XGroup >> pri_obj;
pri_subj < pri_obj;
```

- par héritage (multiple) `<: super_class`
- par croisement de classes sur ressource `+ r` et `- r`



Identification des noeuds

Dans **FRMG** :

- identification explicite sur le nom des noeuds
- identification explicite par aliasing `A=B`
- identification implicitement sur les pères `father(N)`

```
class agreement { % Generic class to add agreement equations
+ agreement;
father(N).bot.number = node(N).top.number;
father(N).bot.gender = node(N).top.gender;
father(N).bot.person = node(N).top.person;
}
class superlative_as_adj_mod {
<: superlative_as_mod;
node(Foot).cat = value(adj);
- det::agreement; det = det::N;
- adj::agreement; Foot = adj::agreement;
}
```

Pas d'identification implicite via des couleurs comme dans XMG
 Reflexions sur l'expression de condition d'extensionnalité

```
extensional [cat: v];
```

Les conflits de noms (lors de l'import multiple d'une même classe) sont évités grâce aux **espaces de noms** sur

- les ressources `- foo::r`
- les noeuds `foo::N`
- les variables `$foo::bar`

Permet une écriture proche d'appels de fonction avec liaisons de noms

```
- foo::node_agreement; A=foo::N1; B=foo::N2;
%% close of: foo::node_agreement( N1 => A, N2 => B );
```

Mais les liaisons peuvent être faites ultérieurement

Proche de XMG sauf que les noms sont globaux par défaut dans **FRMG**

Gardes

- Possibilité de poser des **gardes** ou conditions sur la présence ou absence d'un noeud.

```
Subj => node(v).top.mode = value(~imperative);
~ Subj => node(v).top.mode = value(imperative);
```

- Les gardes sont accumulées par conjonction :
`N => G1` et `N => G2` équivalent à `N => G1,G2`
- Les conditions sont des disjonctions de conjonctions de chemins

```
~ Subj =>
node(v).top.mode = value(imperative)
| node(S).top.inv = value(+),
node(v).top.mode = value(imperative)
;
```

Gardes (suite)

Les gardes sont très puissantes mais fastidieuses
 ⇒ notation plus pratique

- Exclusion entre les noeuds A et B

```
A => desc.dummy.A_or_B = value(a);
~ A => desc.dummy.A_or_B = value(~ a);
B => desc.dummy.A_or_B = value(b);
~ B => desc.dummy.A_or_B = value(~ b);
```

- Unicité (un seul arg verbal extrait)

```
XGroup =>
  $a0.extr = value(+), $a1.extr=value(-), $a2.extr=value(-)
| $a0.extr = value(-), $a1.extr=value(+), $a2.extr=value(-)
| $a0.extr = value(-), $a1.extr=value(-), $a2.extr=value(+);
~ XGroup =>
  $a0.extr = value(-), $a1.extr=value(-), $a2.extr=value(-);
```

Croisements

Croisements guidés par les ressources, ordonnées par *profondeur croissante*

- r_1 dépend de r_2 si il existe $C[+r_1, -r_2]$
- $p(r) = 0$ si r est non dépendante
 $p(r) = 1 + \max p(r_i)$ si r dépend des r_i
- erreur si r dépend de elle-même (directement ou indirectement)

Accumulation des contraintes :

$$C^-[-N::R \cup \mathcal{K}^-] \oplus C^+[+R \cup \mathcal{K}^+] = (C^- \oplus N::C^+)[=N::R \cup \mathcal{K}^- \cup N::\mathcal{K}^+]$$

Croisement impossible si une même ressource (avec espace de noms et éventuellement neutralisée) présente dans les deux classes croisées :

$$\exists r : \text{res}, \exists k_1, k_2 \in \{+, -, =\}, \\ k_1 r \in C_1 \wedge k_2 r \in C_2 \wedge (k_1 = k_2 \vee k_1 = (=) \vee k_2 = (=))$$

Compiler la méta-grammaire

Compilateur **MGCOMP**, développé avec **DYALOG**

Étape 1 : Classes terminales

Héritage des contraintes par les classes terminales (+ vérif contraintes)

Étape 2 : Classes neutres

- Croisement des classes terminales pour neutraliser ressources & besoins
 - ▶ $C_1[-R \cup \mathcal{K}_1] \times C_2[+R \cup \mathcal{K}_2] = (C_1 \times C_2)[=R \cup \mathcal{K}_1 \cup \mathcal{K}_2]$
 - ▶ (Espace de nom) ⇒ import classe productrice avec renommage
 $C_1[-N::R \cup \mathcal{K}_1] \times C_2[+R \cup \mathcal{K}_2] = (C_1 \times N::C_2)[=N::R \cup \mathcal{K}_1 \cup N::\mathcal{K}_2]$
- Réduction des gardes (quand possible)
- Vérification des contraintes

Étape 3 : Arbres TAG/TIG

Utilisation des contraintes des classes neutres pour construire les arbres

Vérification des contraintes

Problème des méta-grammaires :

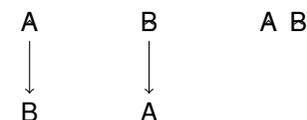
- 1 interpréter ce qui n'est pas dit : possible ou interdit ?
 - 2 interpréter par rapport à une axiomatique :
 arbre (TAG), forêt (MC-TAG), graphe (shared MC-TAG)
- Liens entre les noeuds A et B dans une classe non neutre
 ⇒ noeuds potentiellement aliasables

$$A = B \vee A < B \vee B < A \vee A \triangleright^+ B \vee B \triangleright^+ A$$

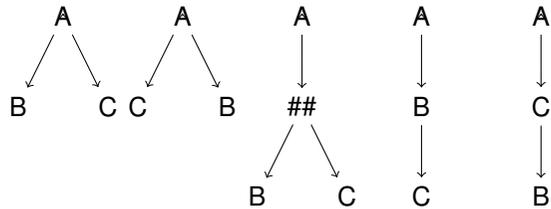
- Liens entre les noeuds A et B dans une classe neutre
 ⇒ noeuds distincts

$$A < B \vee B < A \vee A \triangleright^+ B \vee B \triangleright^+ A$$

Modèle minimal pour 2 noeuds sans contraintes :



Soit $A \gg B$ et $A \gg C$



Fermetures transitives

La vérification des contraintes essentiellement effectuée par des calculs de fermeture transitive, exploitant les propriétés de tabulation (faible) de **DyALog**

```
:-light_tabular precedes_closure/3.
:-mode(precedes_closure/3,+(+,+,-)).

%% Précédence dans la classe
precedes_closure(Class,N1,N3) :-
    precedes(Class,N1,N2),
    ( N3=N2 ; precedes_closure(Class,N2,N3) ).

%% Héritage des précédences des classes parentes
precedes_closure(Class,N1,N3) :-
    xsuper(Class,Super,NS),
    deep_module_unshift(N1,NS,_N1),
    %% Réutilisation de la fermeture sur Super
    '$answers'(precedes_closure(Super,_N1,_N2)),
    deep_module_shift(_N2,NS,N2),
    ( N3=N2 ; precedes_closure(Class,N2,N3) ).
```

Hiérarchisation des types de contraintes :

- 1 Introduction des variables par analyse des équations
- 2 Aliasing entre noeuds
- 3 Gestion des noeuds anonymes
- 4 Dominance entre noeuds
- 5 Précédence entre noeuds (dépend de la dominance)
- 6 Décorations
- 7 Réduction des gardes

Fermetures transitives

$$\begin{aligned}
 \mathcal{K} \cup (A = B) &|- \mathcal{K}[A \mapsto B] \\
 \mathcal{K} &|- \mathcal{K} \cup (A \triangleright^+ B) & A \triangleright B \in \mathcal{K} \\
 \mathcal{K} &|- \mathcal{K} \cup (A \triangleright^+ C) & \begin{cases} A \triangleright^+ B \in \mathcal{K} \\ B \triangleright^+ C \in \mathcal{K} \end{cases} \\
 \mathcal{K} &|- \mathcal{K} \cup (A < C) & \begin{cases} A < B \in \mathcal{K} \\ B < C \in \mathcal{K} \end{cases} \\
 \mathcal{K} \cup (A < B) \cup (B \triangleright^+ C) &|- \mathcal{K} \cup (A < B) \cup (B \triangleright^+ C) \cup (A < C) \\
 \mathcal{K} \cup (A < B) \cup (A \triangleright^+ C) &|- \mathcal{K} \cup (A < B) \cup (A \triangleright^+ C) \cup (C < B) \\
 \mathcal{K} \cup (A < A) &|- \text{fail} \\
 \mathcal{K} \cup (A \triangleright^+ A) &|- \text{fail} \\
 \mathcal{K} \cup (AR_1 B) \cup (AR_2 B) &|- \text{fail} & \begin{cases} \mathcal{R}_1 \in \{<, >\} \\ \mathcal{R}_2 \in \{\triangleright^+, \triangleleft^+\} \end{cases}
 \end{aligned}$$

Exemple

```

precedes_closure ( Class ,N1 ,N3 ,K) :-
  precedes ( Class ,N1 ,N2) ,
  (   N3=N2, K=strict ,
      true
    ; precedes_closure ( Class ,N2 ,N3 ,K) ,
      true
    ; '$answers' (dominates_closure (Class ,N2 ,N3 ,_)) , K= strict ,
      true
    ; '$answers' (dominates_closure (Class ,N3 ,N2 ,_)) , K = dom ,
      true
  ) .

```

Réduction des gardes

$$\begin{array}{c}
 \frac{}{t = t \rightarrow \text{true}} \\
 \frac{}{t = r \rightarrow x = r} \text{mgu}(t, r) \neq \text{Id} \\
 \frac{}{t = r \rightarrow \text{fail}} \neg \text{mgu}(t, r) \\
 \frac{\alpha \rightarrow \text{true} \quad \beta \rightarrow \beta'}{\alpha \vee \beta \rightarrow \text{true}} \\
 \frac{\alpha \rightarrow \text{fail} \quad \beta \rightarrow \beta'}{\alpha \vee \beta \rightarrow \beta'} \\
 \frac{\alpha \rightarrow \alpha' \quad \beta \rightarrow \beta'}{\alpha \vee \beta \rightarrow \alpha' \vee \beta'} \{\alpha', \beta'\} \cap \{\text{true}, \text{fail}\} = \emptyset \\
 \frac{\alpha \rightarrow \text{true} \quad \beta \rightarrow \beta'}{\alpha \wedge \beta \rightarrow \beta'} \\
 \frac{\alpha \rightarrow \text{fail} \quad \beta \rightarrow \beta'}{\alpha \vee \beta \rightarrow \text{fail}} \\
 \frac{\alpha \rightarrow \alpha' \quad \beta \rightarrow \beta'}{\alpha \vee \beta \rightarrow \alpha' \vee \beta'} \{\alpha', \beta'\} \cap \{\text{true}, \text{fail}\} = \emptyset
 \end{array}$$

Réduction des gardes

- Éliminer les portions de gardes trivialement vraies ou fausses
- Appliquer les gardes (positives ou négatives) quand aucune alternative disponible
 - ▶ Un noeud gardé A devient nécessairement présent si sa garde négative G_A^- se réduit à fail.
Application de G_A^+ si simple conjonction
 - ▶ un noeud gardé A devient nécessairement absent si sa garde positive G_A^+ se réduit à fail.
Application de G_A^- si simple conjonction
- Répéter jusqu'à obtention d'un point fixe

Emission des arbres minimaux

Utilisation de l'opérateur d'entrelacement en s'appuyant :

- notion de père potentiel :
 A pp B si B dominé strictement par un ancêtre de A et B sans père
- notion de précedence conditionnelle : $A < B$ quand A et B frères

Principe de l'algorithme :

Étant donné un noeud A et une liste L de fils potentiels de A non encore positionnés par ses ancêtres :

- partitionner $L \cup \text{Descendants}(A)$ en $[C_1, \dots, C_n]$
- chaque C_i de la forme $[B_{i0}, B_{i1}, \dots, B_{ini}]$ avec B_{ij} fils potentiels de B_{i0} ($j > 1$)
- $C_i < C_j \Rightarrow i < j$
- transformation de la partition en une formule avec les opérateurs séquence , et entrelacement ## sur $\text{tree}(C_i)$.

$$C_i = \langle A_i, \mathcal{A}_i \rangle \Rightarrow \forall B \in \mathcal{A}_i, \neg(A_i < B) \wedge \neg(B < A_i) \wedge \neg(B \triangleright^+ A_i)$$

$$F_1 < F_2 \in F \Rightarrow \forall C_i = \langle A_i, \mathcal{A}_i \rangle \in F_1, \forall C_j = \langle A_j, \mathcal{A}_j \rangle \in F_2, \bigwedge \left\{ \begin{array}{l} A_i < A_j \\ \forall B_i \in C_i, \forall B_j \in C_j, \neg(B_i < B_j) \end{array} \right.$$

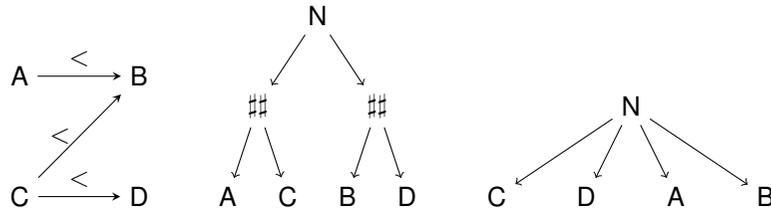
$$F_1 \#\# F_2 \in F \Rightarrow \forall C_i \in F_1, \forall C_j \in F_2, \forall B_i \in C_i, \forall B_j \in C_j, \bigwedge \{ \neg(B_i < B_j) \neg(B_j < B_i) \}$$

Quelques cas particuliers :

- Pas toujours possible de transformer un ordre partiel sur des fils en une seule formule séquence entrelacement.
cas de : $A < B \wedge C < B \wedge C < D$ donne

$$A < D \Rightarrow (A \#\# C) < (B \#\# D)$$

$$D < A \Rightarrow C < D < A < B$$

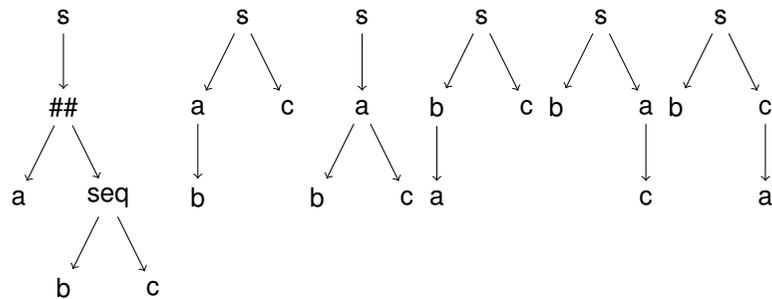


Question : garder entrelacement ou utiliser directement l'ordre partiel

- Pas d'entrelacements au-dessus d'un pied, pour faciliter l'obtention d'arbre TIG.
- Prise en compte d'information de rang (premier ou dernier d'une fratrie)

Exemple

```
class bar {
  s >>> a; s >>> b; s >>> c; b < c;
}
```



Seizième partie XVI

Grammaires de Concaténation d'intervalles

RCG : short presentation

Range Concatenation Grammars (RCG) [Boullier] :
Constraints on intervals on the input string.
For language $a^n b^n c^n$

```
S(X @ Y @ Z) → A(X,Y,Z).
A("a" @ X, "b" @ Y, "c" @ Z) → a(X,Y,Z).
A("","","") → .
aabbcc S([0,6]) →
aabbcc A([0,2],[2,4],[4,6]) →
aabbcc A([1,2],[3,4],[5,6]) →
aabbcc A([2,2],[4,4],[6,6]) →
```

RCG is an operational formalism used to encode linguistic formalisms such as TAG where discontinuous constituents are used.

RCG allow modular grammar writing

concatenation $G(X @ Y) \rightarrow G1(X), G2(Y)$.

union $G(X) \rightarrow G1(X) | G2(X)$.

intersection $G(X) \rightarrow G1(X), G2(X)$.

RCG avec arguments

```
s(N) (X@Y@Z) → a(N) (X,Y,Z).
a(M) ("a" @ X, "b" @ Y, "c" @ Z) → a(N) (X,Y,Z), {M is N+1}.
a(0) ("","","") → true.
axiom(s(N)).
```

Prédicats RCG builtins

```
%% Bibliothèque de prédicats spéciaux
:-require 'rcg.pl'.
```

```
s (X@Y) → s (X), { rcg_eqstrlen(X,Y) }.
s ("a") → true.
```

```
?-recorded('N'(N)),rcg_phrase(s (0:N)).
```

- rcg_eqstr(R1,R2)
- rcg_length(R1,Length)
- rcg_eqstrlen(R1,R2)

RCG et intersection

Opération d'intersection immédiate : il suffit d'avoir plusieurs prédicats portant sur un même intervalle

```
s(X @ Y @ Z) → anb(X @ Y), bnc(Y @ Z).
anbc(X @ Y) → anb2(X,Y).
anb2("","") → true.
anb2("a" @ X, "b" @ Y) → anb2(X,Y).
```

Exemple d'utilisation pour des verbes à contrôle

Paul veut manger une pomme

```
control_verb(Subj @ V @ Sent) →
  np(Subj),
  v(V),
  ctr_sentence(Subj, Sent).
```

Dix-septième partie XVII

Compléments sur DyALog

```
s -> np, vp.
head(s, vp). % ==> s -> np <+ vp
vp -> v(Type), v_args(Type).
head(vp, v). % ==> vp -> v(Type) +> v_args(Type)
```

Lexicalisation

Lexicalization et DyALog

Non obligatoire dans les grammaires pour **DyALog**, mais permet une meilleure efficacité :

Principe : chargement conditionnel des clauses (ou arbres pour les TAGs) :

```
'$loader' ( phrase ([ qui ], _, _ ),
             ( np -> np, [ qui ], s_rel )
           ).
```

- Automatiquement induite pendant la compilation (option `-autoload`) pour TAG et TIG (noeuds lexicaux + ancres), DCG & BMG (lexicaux)
- Utiles pour tester des conditions plus spécifiques (pas nécessairement liées à la lexicalisation)

- Facile de rendre des analyses partielles

```
% full parsing  
?-recorded( 'N' (R) ), L=0, phrase(S::s{ }, L,R) .
```

```
% partial parsing  
?-tag_phrase(s,L,R); phrase(np,L,R) .
```

- possibilité d'utiliser des stratégies bidirectionnelles dirigées par les têtes(DCGs)
- utilisation des FSA en entrée : gestion des amabiguïtés, mots incnnus, ...
- prédicats d'introspection pour explorer le contenu de la table
⇒ ouvre la voie vers des algorithmes de corrections

Modulation

```
dcg_mode(np/2,+(-,-),+,-)  
Stratégie descendante : :-dcg_mode(.,+,+,-)  
Stratégie ascendante : :-dcg_mode(.,-,-,-)
```

Définir son lecteur

```
:-skipper(skip_lexical).  
:-std_prolog skip_lexical/3.  
skip_lexical(Left,Token,Right) :-  
    'C'(Left,  
        lemma{ lex => Token, cat => epsilon },  
        Right) .
```

Namespaces et modules

DyALog fournit

- la notion de **namespaces** pour éviter les conflits de noms : `private lfoo`
- la notion de **modules**, fondée sur les namespaces et sur un mécanisme d'import

```
:-module(forest).           %% Forest module with namespace forest
:-import{ file=>'format.pl', preds => [format/2] }.
:-std_prolog display_forest/1.
...
:-end_require.             %% public interface
```

```
:-import{ module=>forest, preds => display_forest/1 }. %% import
forest module
```

Mais pas de vraie notion de prédicats privés (similaire à la notion de modules en Perl)

Modules

Directives `:-include`, `:-require`, `:-import` et `:-module/1` import et export.

```
:-import{ module=>sqlite,
         file => 'libdyalogsqlite.pl',
         preds => [
                 open/2,
                 close/1,
                 exec/4
                 ]
        }.

test :-
    argv([File]),
    open(File,DB), %% équivalent à sqlite !open
    exec(DB,'create_table_toto_(k_TEXT,v_INTEGER)',[],_R),
    exec(DB,'insert_into_toto(k,v)_values_(?,?)',[a,2],done),
    close(DB)
.
```

Interface vers C

```
socket_bind(Socket,Address) :-
( Address = 'UNIX'(Name) ->
  '$interface'('Socket_Bind_Unix'(Socket:int,Name:string)
, [] )
; Address = 'INET'(Host,Port),
( var(Port) -> In_Port = 0 ; In_Port = Port ),
  '$interface'('Socket_Bind_Inet'(Socket:int,
                                 Host:string,
                                 In_Port:int,
                                 Out_port:-int),
               [] )
).
```