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

Aujourd'hui: Introduction

Première partie I

Lignes directrices

Introduction

Le Traitement Automatique des Langues (TAL) présent dès les origines de la Programmation en logique

- Recherche d'un formalisme syntaxique puissant, fondé sur une base logique solide
- Q-systèmes de Colmeraurer – 1970
- Grammaires de Clauses Définies (DCG - Definite Clause Grammars) Pereira et Warren – 1980
- Parsing as Deduction Pereira, Warren, Shabes, Shieber
- Prolog et contraintes, Dahl

Paradigme déclaratif

La force de la programmation en logique est de pouvoir

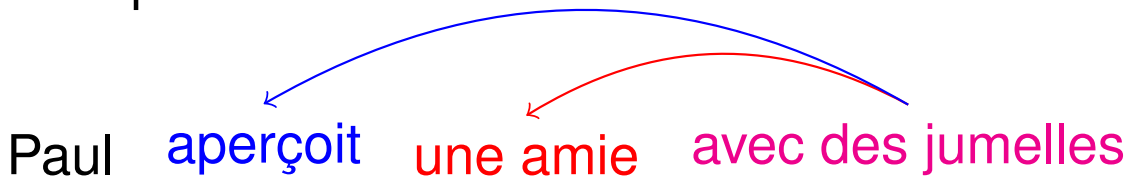
- exprimer de manière déclarative des informations complexes
- sans se préoccuper de la manière dans elles seront utilisées opérationnellement
  confiance dans le principe de déduction logique sous jacent

  algorithme = logique + contrôle [Kowalski]
- évolution nette par rapport à des approches antérieures mêlant données et opérations (RTN – *Recursive Transition Network*)

Non-déterminisme et ambiguité

Prolog conçu pour la gestion du non-déterminisme

- exploration de l'espace de recherche par retour arrière (backtrack)
- important pour gérer les ambiguïtés importantes du langage, en particulier en analyse syntaxique

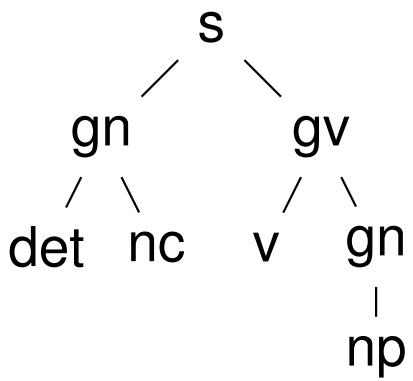


- mais gestion du non-déterminisme pas encore assez efficace
  $\Rightarrow$ extensions tabulaires (partage de calculs)

Notation par constructeur

Flexibilité de construction (récursif) de termes avec les constructeurs, constantes et variables

- représentation d'arbres s(gn(det,nc),gv(v,gn(np)))



- représentation d'expressions : $a \wedge (b \vee c) \rightsquigarrow$ (a,(b;c))
  mais aussi

```
:-op(yfx,[or],500).  %% associatif droit
:-op(yfx,[and],400). %% associatif droit
?- F= a and (b or c).
```

- représentation de λ-expressions : $\lambda x \lambda y, \mathrm{aime}(x, y) \rightsquigarrow$ X^Y^aime(X,Y)

Information partielle et sous-spécification

Prolog s'appuie sur l'unification :

- permet des notations compactes en ne précisant que la partie de l'information nécessaire pour appliquer une clause
  $\Rightarrow$ sous-spécification

- permet de propager l'information

```
?-det(NumDet,GenDet),
  NumDet=plur,                          % les
  nom(NumNom,GenNom),
  NumDet=NumNom,GenDet=GenNom,          % enfants
  v(NumV,GenV),
  NumV=NumNom,GenV=GenNom,GenV=fem       % sont venues
```

- utile pour gérer des objets linguistiques complexes, comme les mots (beaucoup de propriétés), partageant des info (accord)

- Extensions vers Structures de traits typées [TFS - *Typed Feature Structure*] pour encore plus de compacité.

Méta-interprétation et compilation

Facile en Prolog d'écrire un méta-interprète pour Prolog
autrement dit : écrire Prolog en Prolog (similaire en Lisp, Scheme)

```
solve(Goal) :- clause((Goal :- Body)), solve(Body).
solve((Formula1,Formula2)) :- solve(Formula1), solve(Formula2).
solve((Formula1; Formula2)) :- solve(Formula1).
solve((Formula1; Formula2)) :- solve(Formula2).
```

Permet d'écrire en Prolog des extensions de Prolog :

- pour des formalismes syntaxiques
- pour de nouvelles stratégies d'analyse
- très pratique pour du prototypage rapide

Cependant, la méta-interprétation a un coût en temps
$\Rightarrow$ à terme, préférable d'évoluer vers une compilation

$$\text{compilation} = \text{méta-interprétation} + \text{évaluation partielle}$$

En s'inspirant d'un méta-interprète, obtention d'un compilateur en Prolog.

DyALog

Un Prolog maison, adapté au TAL, permettant :

- la gestion de divers formalismes syntaxiques (DCG, BMG=DCG++, TAG, RCG)
- une meilleure gestion du non-déterminisme (approche tabulaire)
- plus de choix sur les stratégies d'analyses
- l'utilisation de structures de traits (typées ou non)
- ...

`http://alpage.inria.fr/dyalog.fr.html`

DyALog : mode toplevel

DyALog peut être utilisé en mode interactif (−toplevel),
mais moins de fonctionnalités que d'autres Prolog

```
%> dyalog -toplevel
Entering DyALog toplevel
DyALog> ancetre(X,Y) :- parent(X,Y).
DyALog> ancetre(X,Z) :- parent(X,Y),ancetre(Y,Z).
DyALog> parent(paul,marie).
DyALog> parent(marie,jean).
DyALog> ?-ancetre(paul,Y).
        Y = jean
        Y = marie
DyALog> quit.
Leaving toplevel. Good bye!
```

DyALog : mode compilateur

DyALog surtout conçu pour **compiler** des programmes et des analyseurs syntaxiques.

- le programme : `ancetre.pl`

```
%% Directive: fait Prolog de type base de données
:-extensional parent/2.

%% Clauses (recursive)
ancetre(X,Y) :- parent(X,Y).
ancetre(X,Z) :- parent(X,Y),ancetre(Y,Z).

%% Requete - argv = accès aux args après '--'
?- argv([X]), ancetre(X,Y).
```

- la base de données généalogique : `famille.db`

```
parent(paul,marie).
parent(marie,jean).
```

Note : Séparation claire entre programme et données

DyALog : mode compilateur (suite)

- compilation : dyacc ancetre.pl −o ancetre
- la requête (avec argv([paul]))

```
%> ./ancetre famille.db -- paul
Answer:
    X = paul
    Y = marie
Answer:
    X = paul
    Y = jean
```

- variantes pour la requête :
  - ./ancetre −db famille.db −− paul
  - cat famille.db | ./ancetre − −− paul

DyALog : aide

dyalog –help

```
Usage for command dyalog
   –h –help          — this help
   –v <level>        — trace for <level> in dyam, index, share,
    or all
   –verbose <level>  — same as –v
   –db <filename>    — load database filename
   –I <path>         — add <path> to DyALog search path
   –server           — enter server mode
   –loop             — enter loop mode
   –forest           — display the shared forest
   –fcount           — count number of derivations per answer
   –slex <string>    — parse from <string>
   –flex <filename>  — parse from <filename>
   –a <args>         — all remaining arguments given to DyALog
   — <args>          — all remaining arguments given to DyALog
```

DyALog : aide (suite)

- dyacc –help
- Documentation incomplète à
  `http://alpage.inria.fr/docs/dyalog.pdf`
  et dans la distribution de DyALog

Deuxième partie II

Quelques gammes : Expressions régulières, Automates à états finis et Transducteurs

Outline

Expressions régulières

Une valeur sûre de l'informatique et du TAL

- analyse lexicale (lexer)
- entités nommées
- morphologie

Constructeurs de base :

- constantes : alphabet fini (caractères, ou ce qu'on veut)
- Concaténation
- Alternation
- Étoiles de Kleene (répétitions)

Constructeurs additionnels (sucre syntaxique)

- intervalles de constantes `[...]`,
- optionel `E?` ,
- au moins une fois `E+`,
- entre *n* et *m* fois `E{n,m}`,
- différence `E1-E2`,
- intersection `E1 & E2`,
- ...

Représentation Prolog

Nombre : (\+|−)?[0−9]∗(\.[0−9]+)? ⇝

```
:−op(yf,['@?'],500).
:−op(yf,['@+'],500).
:−op(yf,['@∗'],500).

rx(numbers,
    ( (c(+) ; c(−)) @?,
      range("0123456789") @∗,
         ( c(0'.), range("0123456789") @+ ) @?
)).
```

Notes :

- "0123456789" équivalent à la liste [0'0,0' 1,..,0 '9]
- D'autres représentations sont possibles !

Un interpréteur en Prolog

Utilisation d'un prédicat regexp(RegExp,Left,Right)

- pour reconnaître l'expression régulière RegExp
- entre la "*position*" Left
- et la "*position*" Right

Note : Notion de position définie plus loin mais

- utilisation récurrente en Prolog/TAL de paires de positions
- lien avec notion de différence de liste Left−Right ≡ Left=[c1 ,.., cN|Right]
- et instance de la notion d'accumulateur

```
reverse(X, Rev) :- reverse(X, [], Rev).
reverse([], Rev, Rev). % Rien de plus à renverser
reverse([H|T], Prev, Rev) :- reverse(T, [H|Prev], Rev).
```

Déconstruire

Déconstruire les divers constructeurs d'expressions régulières :

```
regexp((E1,E2),L,R) :-
        regexp(E1,L,M),
        regexp(E2,M,R).

regexp((E1;E2),L,R) :-
        ( regexp(E1,L,R)
        ; regexp(E2,L,R)
        ).

regexp(E @*,L,R) :-
        ( L=R ;
          regexp(E,L,M),
          regexp(E @*,M,R)
        ).
```

Déconstruire (suite)

Quelques constructeurs supplémentaires

```
regexp(true,L,L).

regexp(range(Range),[C|R],R) :-
  domain(C,Range).  %% ou member(C,Range) en Prolog

regexp(E @?,L,R) :-
  regexp((true;E),L,R).

regexp(E @+,L,R) :-
regexp(E,L,M),
regexp(E @*,M,R).
```

Lire les symboles

```
regexp(c(C),L,R) :- 'C'(L,C,R).
regexp(range(Range),L,R) :- 'C'(L,C,R), domain(C,Range).
```

Abstraction du lecteur de symbole avec le prédicat 'C'/3 :

```
:-extensional 'C'/3. %% directive DyALog
'C'([C|R],C,R).
```

L'abstraction permet de changer facilement de source de lecture :

- source = liste [il ,mange,une,pomme] (formulation ci-dessus)
  immédiat et réversible : lecture liste / génération liste
- source = treillis de mots

  ```
  'C'(0,il,1).
  'C'(1,mange,2).
  'C'(2,une,3).
  'C'(3,pomme,4).
  ```

  avantage : plus efficace et extensible

La requête

```
:-extensional rx/2.
?-argv([Name,Symb]),
  rx(Name,RX),
  atom_chars(Symb,CharString),
  regexp(RX,CharString,[])
  .
```

Compilation : dyacc regexp.pl -o regexp

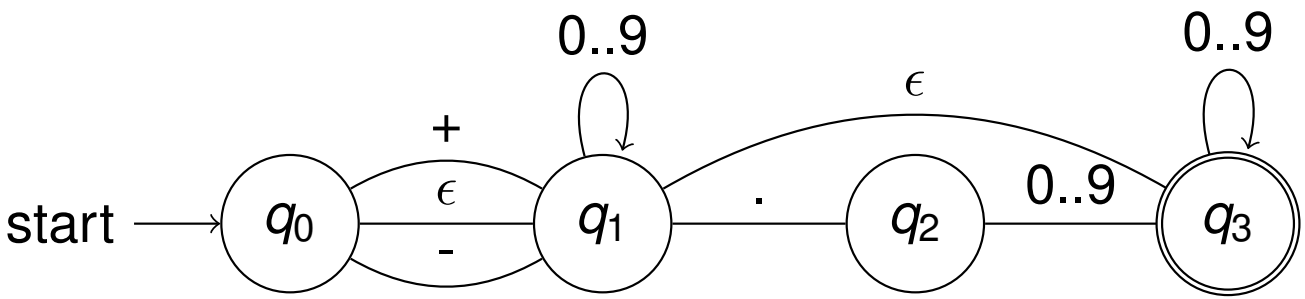
Exécution : ./regexp numbers.rx -- numbers +123.98

Outline

Automates à états finis

- Équivalence entre expressions régulières et Automates à États Finis [FSA – *Finite State Automata*]
- FSA $\equiv$ langage opérationnel bas niveau
  (vs RegExp $\equiv$ langage déclaratif haut niveau)
- simplicité formelle $\Rightarrow$ moteur Prolog simple

'numbers' en FSA




```
init_state(numbers,0).
final_state(numbers,3).
trans(numbers,0,1,c(0'+)).
trans(numbers,0,1,c(0'-)).
trans(numbers,0,1,true).
trans(numbers,1,1,range("0123456789")).
trans(numbers,1,2,c(0'.)).
trans(numbers,2,3,range("0123456789")).
trans(numbers,3,3,range("0123456789")).
trans(numbers,1,3,true).
```

Moteur FSA en DyALog

```
:-extensional trans/4.
:-extensional init_state/2.
:-extensional final_state/2.

fsa(Name,L,R) :-
        init_state(Name,Init_State),
        fsa(Name,Init_State,L,R).

fsa(Name,S,L,L) :- final_state(Name,S).
fsa(Name,S,L,R) :-
        trans(Name,S,T,A),
        fsa_action(A,L,M),
        fsa(Name,T,M,R)
        .
```

Moteur FSA : actions

```
fsa_action(true,L,L).
fsa_action(c(X),L,R) :- 'C'(L,X,R).
fsa_action(range(D),L,R) :- 'C'(L,X,R),domain(X,D).
```

Note : facile d'ajouter de nouvelles actions

Moteur FSA : requête

```
?-argv([Name,Symb]),
  rx(Name,RX),
  atom_chars(Symb,CharString), % prédicat builtin
  fsa(RX,CharString,[])
  .
```

Compilation : dyacc fsa.pl -o fsa

Exécution : ./fsa numbers.fsa -- numbers +123.98

Outline

Lire et émettre

Facile d'étendre en un moteur pour des transducteurs à états finis

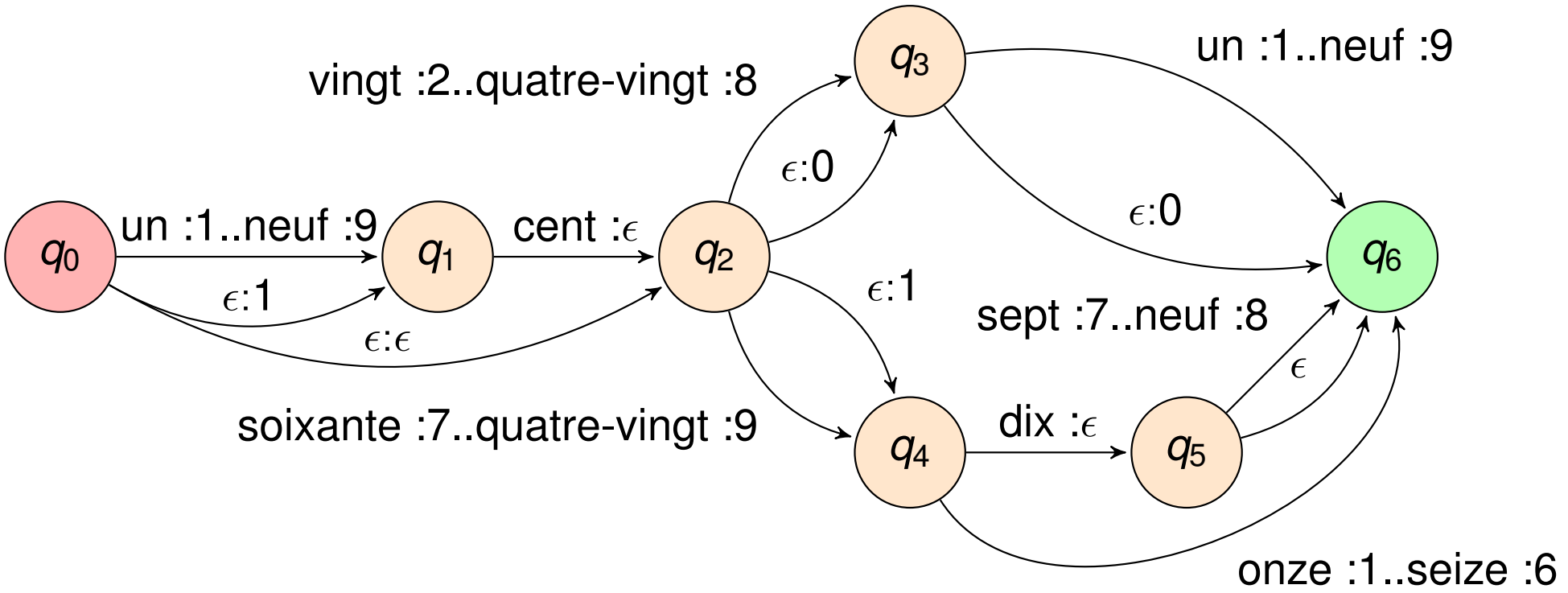
- permet de lire sur une bande et écrire de l'autre
- les rôles peuvent être théoriquement être inversés

Exemple : normaliser des nombres entre 0 et 999 écrits en toute lettre
ex : lire deux cent seize et retourner [2,1,6]
ex2 : réciproquement, lire [2,1,6] et retourner deux cent seize.

Les transitions du transducteur sont maintenant décorées d'une paire d'action :

```
fst_trans(numbers2,0,1,true : c(1)).
fst_trans(numbers2,0,1,c(un) : c(1)).
fst_trans(numbers2,0,1,c(deux) : c(2)).
..
```

Forme du transducteur (simplifié)



Au moins deux faiblesses dans ce transducteur !

Moteur FST

Similaire au moteur FSA, mais :

- utilise 2 paires de positions L1,R1 et L2,R2
- utilise 1 paire d'actions A1 et A2

```
fst(Name,S,L1,R1,L2,R2) :-
    fst_trans(Name,S,T,(A1:A2)),
    fst_action1(A1,L1,M1),
    fst_action2(A2,L2,M2),
    fst(Name,T,M1,R1,M2,R2).
```

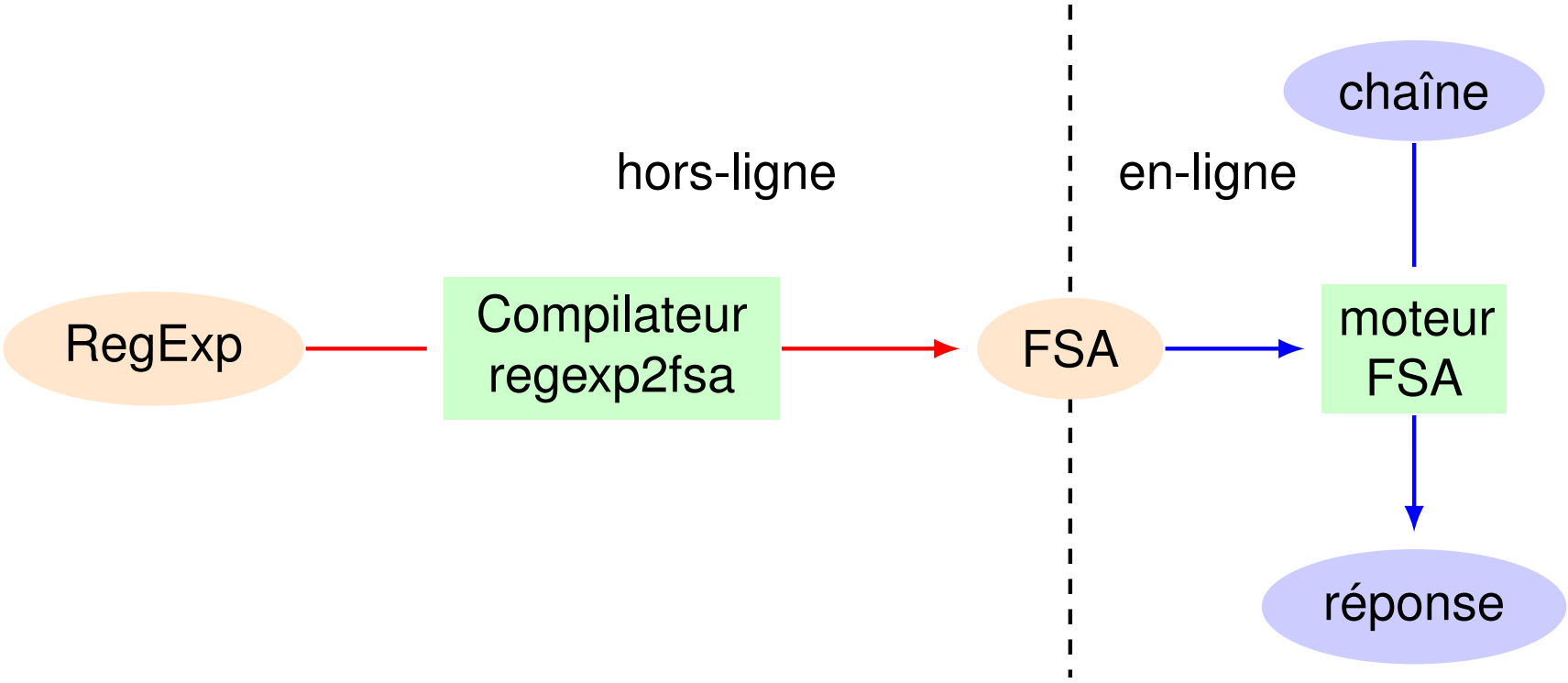
Notes :

- Jeux d'actions identiques ou différents pour chaque niveau (1 ou 2).
- Immédiat d'étendre à 3 (ou plus) bandes
  par exemple pour lire les nombres en français, les normaliser et les générer en anglais.

Outline

Principe

- Utiliser la notation haut-niveau des expressions régulières
- Obtenir l'efficacité des moteurs FSA ou FST
  (ou directement le moteur Prolog)



Regexp2fsa

Modèle générique de compilateur

1. lire le source (une regexp)
2. transformer (similaire au moteur regexp)
   déconstruire les expressions en introduisant de nouveaux états
   regexp2fsa(Name,RegExp,State1,State2)
3. émettre la cible (FSA)
   émettre les transitions avec emit_trans(Name,State1,State2,Action)

Déconstruire les expressions

```
regexp2fsa(Name,(E1,E2),S1,S2) :-
        regexp2fsa(Name,E1,S1,S3),
        regexp2fsa(Name,E2,S3,S2)
        .
regexp2fsa(Name,(E1;E2),S1,S2) :-
        regexp2fsa(Name,E1,S1,S2),
        regexp2fsa(Name,E2,S1,S2)
        .
regexp2fsa(Name,E @*,S1,S2) :-
        regexp2fsa(Name,E,S1,S1),
        regexp2fsa(Name,true,S1,S2)
        .
```

Note : similaire pour les autres constructions

Construire les transitions

```
regexp2fsa(Name,A,S1,S2) :-
    domain(A,[c(X),range(D),true]),
    (var(S2) -> mutable_new_state(Name,S2) ; true),
    emit_trans(Name,S1,S2,A).
```

Emettre les transitions

```
%% bibliotheque analogue a C printf
:-require 'format.pl'.

%% directive DyALog: prédicat à la Prolog
:-rec_prolog emit_trans/4.

emit_trans(Name,S1,S2,A) :-
    format('trans(~w,~w,~w,~w).\n',[Name,S1,S2,A]).
```

Chapeau de compilation

```
:-extensional rx/2.
:-rec_prolog regexp2fsa/4.

?-rx(Name,RX),
  mutable_init_state(Name, Init),
  regexp2fsa(Name,RX, Init ,Final),
  format('init_state(~w,~w).\n',[Name, Init]),
  format('final_state(~w,~w).\n',[Name, Final]),
  %% la compilation se contente d'émettre
  %% => pas de réponse avec fail
  fail.
```

Engendrer des états

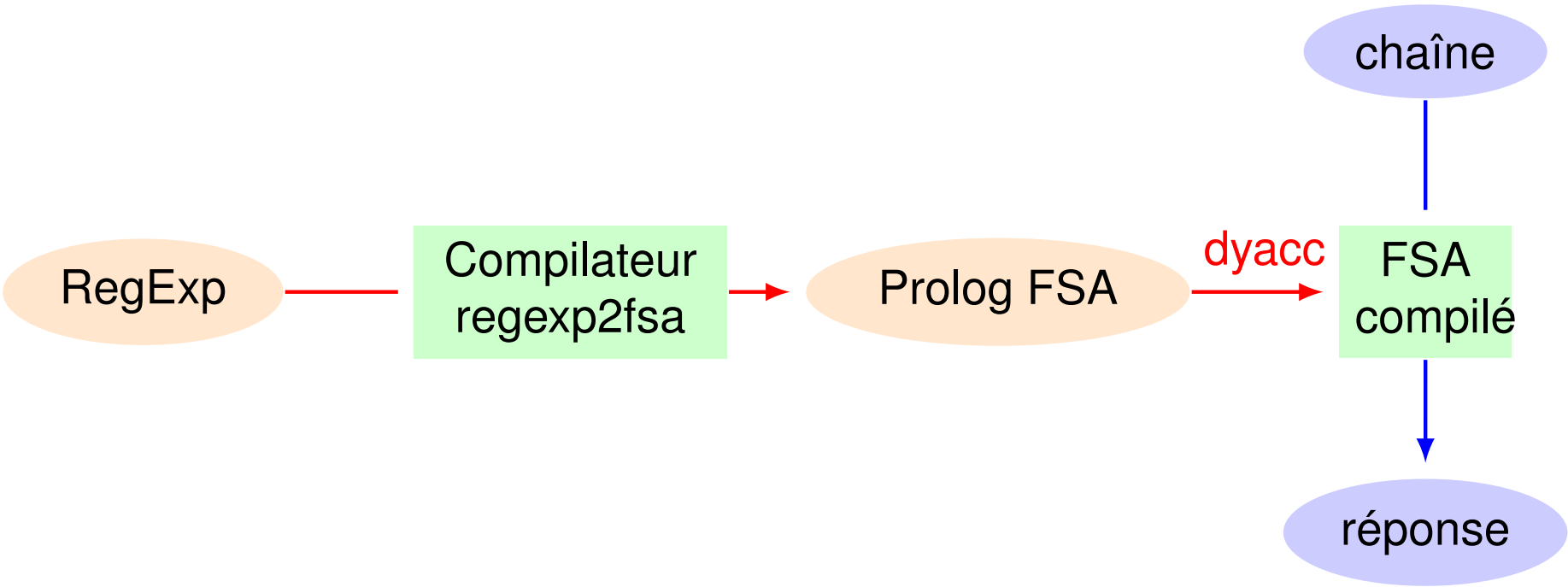
- Utilisation de prédicats *builtin* non logiques de DyALog : mutable/1 et mutable_inc/1
- Remplacement possible par des assert/retract
- Gestion possible sans prédicats non logiques (en propageant le max des états déjà générés)

```
:-std_prolog mutable_init_state/2, mutable_new_state/2.

mutable_init_state(Name,0) :-
    mutable(MX,1),
    record(state(Name,MX)).

mutable_new_state(Name,S) :-
    recorded(state(Name,MX)),
    mutable_inc(MX,S).
```

Compiler vers Prolog/DyALog

Une compilation plus poussée des expressions régulières peut prendre Prolog comme cible (plutôt que des transitions FSA),

- en émettant des clauses Prolog encapsulant le comportement du moteur FSA pour chaque transition de l'automate.
- (alternative) composer deux compilateurs :
  regexp $\rightsquigarrow$ fsa, puis fsa $\rightsquigarrow$ Prolog
  approche fréquente de compilation multi-passe avec des représentations intermédiaires.



Pour aller plus loin

Voir le package FSA de Gert van Noord

- disponible sur `http://www.let.rug.nl/~vannoord/Fsa/`
- fonctionne avec YAP, SICSTUS et SWI.
- Beaucoup plus d'opérateurs d'expression régulières
- Optimisations des FSA/FST (minimisation et déterminisation)
- FSA et FST pondérés