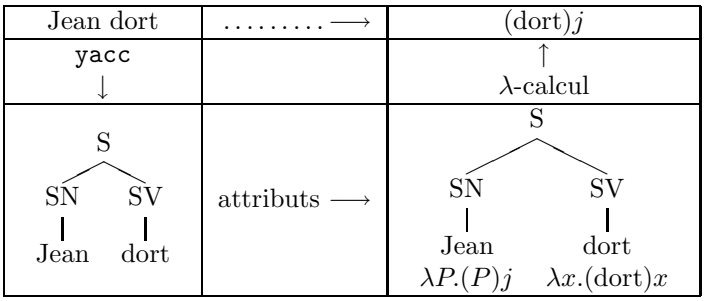
Développement d'un interpréteur de LN



0 Cadre général

L'objectif de cette série de TP est de construire un programme qui, prenant en entrée une phrase en français, est capable d'en construire une représentation sémantique (en logique du 1er ordre), et d'interpréter cette représentation dans un modèle.

Les hypothèses pertinentes pour ce TP sont les suivantes : (1) la syntaxe est non ambigüe, d'où la possibilité d'utiliser un *parser* de la famille LR(), (2) le langage est réduit au 1$^{\text{er}}$ ordre.

Programme prévu pour les 5 séances

1. Parsing + évaluation de formules de la logique propositionnelle
2. Parsing + évaluation de formules de la logique des prédicats (construction de l'arbre syntaxique des formules)
3. Parsing + β-réduction de λ-expressions
4. Parsing du langage naturel et construction de la représentation logique
5. Gestion du lexique + évaluation des phrases par rapport à un modèle.

1 Parsing + évaluation de formules propositionelles

On fait plusieurs choix syntaxiques qui changent par rapport au cours :
- Les applications d'une **fonction** M à son argument N sont systématiquement notées `(M N)`.
- Les connecteurs de la logique des propositions sont traités syntaxiquement comme des fonctions, d'où la notation `(not p)`.
- Les connecteurs binaires sont "curryfiés", d'où la notation `((and p) q)`.

Quelques exemples de formules valides :

```
p
((and p) q)
(not ((imp p) ((or p) q)))
```

1. Ecrire un parseur qui reconnaît ces formules. On suppose que le connecteurs sont reconnus par `lex`, et que toute lettre isolée compte comme une variable propositionnelle.
2. Ajouter au parser les actions sémantiques permettant de calculer la valeur de vérité de la formule. On suppose que l'on dispose d'une fonction, valuation, qui fournit pour chaque symbole de proposition, sa valeur de vérité. Dans un premier temps, on prendra comme fonction :

```
int valuation(char p)
{
  if (p % 2 == 0) return 1 ;
  else return 0 ;
}
```

3. Pour faciliter le calcul, on construit un arbre qui représente la formule parsée ; ce qui permettra aussi de proposer une fonction qui affiche sous forme infixe la formule stockée.
   Pour cela, on utilise des définitions comme les suivantes :

```
// Fichier : lparser-code.c
// Type    : C
// Contient: code additionnel pour parser de lambda-termes
// ---------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <strings.h>

// Définitions de types
#include "lparser-code.h"

// ---------------------------------------------------------------------
// Construction et manipulation de l'arbre syntaxique

noeud * cree_n(int type, int val, noeud * fg, noeud * fd)
{
  noeud * x = (noeud *) malloc (sizeof(noeud)) ;
  if (x == (noeud *)NULL) exit(-1) ;
  x->type = type ;
  x->val = val ;
  x->fg = fg ;
  x->fd = fd ;
  return x ;
}
```

   avec un fichier d'entête (`.h`) qui pourrait contenir :

```
// Fichier : lparser-code.h
// Type    : C
// Contient: types et proto-types pour lparser-code.c
// ---------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <strings.h>

// Définitions de types
#include "lparser.tab.h"

struct noeud {
  int type ;      // Même valeurs que les tokens
  int val ;
  struct noeud * fg ;
  struct noeud * fd ;
} ;
typedef struct noeud noeud ;

noeud * cree_n(int type, int val, noeud * fg, noeud * fd) ;
void affiche_formule(noeud * racine) ;
```

   ... il reste à écrire la fonction `affiche_formule()`...

Ébauche de solution pour l'évaluation directe :

```
S: E ';' { printf("Valeur : %s\n", ($1==1)?"Vrai":"Faux") ; } S
|  /* epsilon */
;

E: PROP                       { $$ = valuation($1) ; }
|  '('NON E ')'               { $$ = ($3 == 1) ? 0 : 1 ; }
|  '(' '(' ET E')' E ')'      { $$ = (($4==1)&&($6==1)) ? 1 : 0 ; }
|  '(' '(' OU E')' E ')'      { $$ = (($4==1)||($6==1)) ? 1 : 0 ; }
|  '(' '(' IMP E')' E ')'     { $$ = (($4==1)&&($6==0)) ? 0 : 1 ; }
;
```

2 Parsing + évaluation logique des prédicats

[Préliminaire : pour faciliter la manipulation des formules, on va autoriser l'écriture infixe des connecteurs binaires `(p and q)` en plus de l'écriture préfixe et curryfiée `((and p) q)`. Mais la représentation en mémoire est inchangée.]

On augmente le langage pour inclure des prédicats (unaires), et des quantificateurs. Par anticipation du traitement du λ-calcul, on ajoute un point entre le quantificateur et sa portée. Quelques exemples valent mieux qu'un long discours (constantes non logiques en anglais pour faciliter le traitement du LN par la suite) :

```
(sleep j)
((and (sleep j)) (snooze j))
all x. ((imp (dog x)) (bark x))
some x. ((and (cat x) (not (sleep x))))
(not all x. ((imp (man x)) ((love x) m)))
```

1. Augmenter la grammaire du parser pour analyser ces expressions. Le lexique est entièrement "codé en dur" dans le fichier lex. Par exemple (tokens définis dans le fichier yacc) :

   ```
   x|y|z|t          { yylval=yytext[0]-'x' ; return VAR ; }

   some             { yylval=0 ; return QUANT ; }
   all              { yylval=1 ; return QUANT ; }

   j|john           { yylval=0 ; return CTE ; }
   m|mary           { yylval=1 ; return CTE ; }

   sleep            { yylval=0 ; return PRED ; }
   man              { yylval=1 ; return PRED ; }
   snore            { yylval=2 ; return PRED ; }
   love             { yylval=3 ; return PRED ; }
   ```

2. Construire en mémoire l'arbre syntaxique correspondant (fonction `cree_n()`).
3. A partir de l'arbre syntaxique, calculer la valeur de vérité des formules sans quantificateurs, en supposant une fonction d'accès au modèle.
4. Ajouter le calcul des formules comportant variables ou quantificateurs.

3 Parsing et β-réduction de λ-expressions

On ajoute le lambda, représenté par un anti-slash (\), dans le langage précédent, ce qui permet de donner du sens à des formules comme :

```
\x. \y. ((love x) y)
(\P. some x. (P x) \y. (man x))
```

Les fonctionnalités à implémenter sont les suivantes :

1. augmentation de la grammaire `yacc` et construction des arbres correspondants
2. implémentation d'une fonction qui peut renommer toutes les occurrences libres d'une variable donnée avec un autre nom ;
3. implémentation d'une fonction capable de faire la liste des variables utilisées dans un λ-terme donné ;
4. implémentation d'une fonction capable du dupliquer un sous-arbre (*i.e.* une sous-formule) donné ;
5. implémentation de la β-réduction :
   - pour chaque terme de la forme 'redex' :
   - vérifier que les ensembles de variables en jeu sont non intersectifs ;
   - renommer les variables le cas échéant ;
   - substituer toutes les occurrences (liée par le λ) de la variable en jeu par une copie du sous-arbre correspondant à l'argument