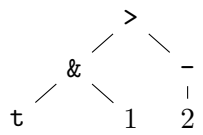
TP de traitement des formules propositionnelles

Ce TP introduit les concepts du cours de traitements des langages informatiques dans le cas particulier du langage des formules propositionnelles. Ce TP sert donc aussi d'illustration au cours de logique. [1]

Les formules propositionnelles sont des *expressions* qui ne contiennent que des opérations et des variables propositionnelles. Concrètement, une telle formule est ici constituée à partir
– des constantes `'t'` (vrai) et `'f'` (faux) ;
– des variables booléennes, nommées en utilisant des entiers strictement positifs ;
– de la négation logique `'-'` (ainsi "`-t`" vaut "`f`") ;
– et des opérateurs binaires suivants : `'&'` (et logique), `'|'` (ou logique), `'>'` (implication logique).
Par exemple, "`(t & 1) > -2`" se lit *"vrai et 1 impliquent non 2"*. Ce langage est donc un des plus simples qu'on puisse imaginer. Ceci dit, il est suffisamment expressif pour exprimer des problèmes NP-complets, donc difficiles à résoudre.

On appelle *analyseur syntaxique* (ou *parser* en anglais) un programme informatique capable de lire une suite de caractères comme "`(t & 1) > -2`" et de l'interpréter comme une formule logique. Écrire un tel programme est un problème *à priori* non trivial, mais qui a des solutions bien comprises depuis la fin des années 1960. L'objectif du cours de période 2 est de présenter une de ces solutions (parmi les plus simples). Mais, avant d'en arriver là, on va commencer par se placer *en aval* de l'analyse syntaxique, c'est-à-dire en supposant que celle-ci a déjà été réalisée !

On commence donc par étudier le résultat de l'analyseur syntaxique lorsque la suite de caractères en entrée correspond bien à une formule syntaxiquement correcte. Ce résultat est un arbre appelé *arbre de syntaxe abstraite* (dessiné ci-contre) qui représente la structure du *parenthèsage* de la formule.



Ainsi, les différents concepts de TL qu'on va illustrer au travers de ce TP programmé en C sont, par ordre d'apparition :
– les arbres de syntaxe abstraite ou *AST* (pour *Abstract Syntax Trees*) en sections 0 et 3 ;
– l'évaluation d'expression (ou interprétation d'expression) en section 1 ;
– la compilation, au sens d'une traduction d'une structure d'AST en une autre structure d'AST en sections 4 ;
– l'analyse syntaxique en notation préfixe en section 5 ;
– et l'analyse syntaxique en notation infixe via une grammaire hors-contexte LL(1) en section 6.
La description des fichiers fournis est donnée dans l'annexe section 7.

0 La notion d'AST illustrée sur les formules propositionnelles

On introduit une notation mathématique pour définir les AST indépendamment de la façon de les représenter dans tel ou tel langage de programmation. [2] Cette notation sera formellement définie en

1. Historiquement, beaucoup des concepts de TL sont d'ailleurs issus de la formalisation de la logique.
2. Ce premier TP est en C, alors que le projet GL en fin de période 3 sera en Java. Les manipulations d'arbres sont assez différentes dans ces 2 languages.

cours. On se contente ici de la présenter de manière intuitive.

0.1 Une notation pour les structures d'AST

Dans le cours, on formalise une structure d'AST comme une *signature multisortée* dont le rôle est de définir *la forme possible des arbres*. Une telle signature est donnée par un ensemble fini de *sortes* (qui nomment des types d'arbre), un ensemble fini de *constructeurs* (qui correspondent à des noms écrits sur les nœuds) et un type (ou profil) pour chaque constructeur. Ce type permet de considérer le constructeur comme une fonction qui, étant donné un n-uplet d'arbres (avec éventuellement $n = 0$), retourne un nouvel arbre dont ceux-là sont les fils. Concrètement, ce type donne la sorte attendue de chaque fils du constructeur, ainsi que la sorte des arbres dont le constructeur est racine.

Dans le cas des formules propositionnelles, on utilise deux sortes : **Prop** pour le type des formules et **Pos** pour le type des entiers strictement positifs (qui représentent des noms de variables). On introduit ensuite un constructeur pour chaque opération ou constante du langage aisément identifiable à partir du nom du constructeur (e.g. *true* pour `'t'` et *and* pour `'&'`).

$$
\begin{aligned}
\mathit{true} &: \mathbf{Prop} \\
\mathit{false} &: \mathbf{Prop} \\
\mathit{neg} &: \mathbf{Prop} \to \mathbf{Prop} \\
\mathit{and} &: \mathbf{Prop} \times \mathbf{Prop} \to \mathbf{Prop} \\
\mathit{or} &: \mathbf{Prop} \times \mathbf{Prop} \to \mathbf{Prop} \\
\mathit{implies} &: \mathbf{Prop} \times \mathbf{Prop} \to \mathbf{Prop} \\
\mathit{var} &: \mathbf{Pos} \to \mathbf{Prop}
\end{aligned}
$$

Le cas le moins évident est celui de *var* utilisé pour représenter les variables. Ici, **Pos** est qualifiée de *sorte externe* : la signature ne dit pas comment fabriquer des éléments de ce type (on suppose qu'on sait le faire par ailleurs). Tandis que **Prop** est une *sorte interne*. On ne peut fabriquer des arbres de type **Prop** qu'en utilisant les constructeurs de la signature.

Avec cette notation, l'arbre associé à l'exemple introductif "`(t & 1) > -2`" s'écrit [3]

$$\mathit{implies}(\mathit{and}(\mathit{true}, \mathit{var}(1)), \mathit{neg}(\mathit{var}(2)))$$

Une façon de coder cette signature en C est donnée par l'inferface suivante. La sorte **Prop** est un type abstrait de pointeur, tandis que **Pos** est représentée abusivement par **int**. Les constructeurs sans arguments sont codés comme des constantes. Les autres constructeurs sont représentés comme des fonctions.

```
typedef
  const struct Prop_Node *Prop;
Prop const true;
Prop const false;
Prop neg(Prop);
Prop and(Prop, Prop);
Prop or(Prop, Prop);
Prop implies(Prop, Prop);
Prop var(int);
```

Le principal intérêt d'une telle structure d'AST est de permettre la programmation des traitements sur le langage indépendamment de l'analyse syntaxique : on détaille une façon de programmer ces traitements en section 0.2.

Le deuxième intérêt de cette structure est qu'elle facilite l'écriture d'un programme qui génère une formule de type `Prop`. En effet, le *compilateur* vérifie *statiquement* que les formules générées sont bien formées pour toute exécution du programme. Par exemple, on ne peut pas écrire de programme compilable qui génèrerait une formule mal formée correspondant à "`- & 1`". L'analyseur syntaxique va être lui-même un exemple d'un tel programme.

3. En cours, on l'écrira plutôt sans parenthèse "*implies and true var* 1 *neg var* 2". En effet, avec les constructeurs écrits *avant* leurs arguments, il n'y a qu'une façon de parenthéser qui respecte le typage des constructeurs.

0.2 Implémentation des AST en C

On présente ici un patron de programmation très classique en C pour coder les AST. Les noms des constructeurs apparaissent dans un type énuméré appelé ici `Prop_Root`.

```
typedef enum { N_true, N_false,
               N_var, N_neg, N_and, N_or, N_implies } Prop_Root;
```

On a défini précédemment que le type `Prop` correspond au type **struct** `Prop_Node *` où **struct** `Prop_Node` est le type des noeuds d'arbres défini par :

```
struct Prop_Node {
        Prop_Root root;
        union {
                int id;   // case var
                Prop son; // case neg
                struct { Prop sonL, sonR; } bin; // case and, or, implies
        } u;
};
```

Cette structure a donc deux champs : le champ `root` qui contient le nom du constructeur à la racine de l'arbre, et le champ `u` qui est une union de tous les types d'arguments possibles. Chaque champ de cette union correspond donc à un profil de constructeur non constant : on a trois cas, celui de `var` (champ `id`), celui de `neg` (champ `son`), et enfin celui des constructeurs binaires (champ `bin`). Ainsi les champs `son`, `sonL` et `sonR` représentent des sous-arbres.

Les fonctions constructeurs de l'interface se contentent donc d'allouer un nouveau noeud (via un `malloc`) en l'initialisant correctement (voir le code fourni dans le fichier `prop.c`).

Un calcul sur un AST correspond typiquement à appliquer un traitement approprié en fonction du constructeur à la racine de l'arbre et des traitements récursifs effectués sur les sous-arbres. À titre d'exemple, voilà comment on programme la fonction `max_var(p)` qui retourne 0 si `p` est constante ou la variable maximale de `p` sinon. Sur l'exemple introductif, la variable maximale est 2.

```
int max_var(Prop p){
        switch (p->root) {
        case N_true: case N_false:
                return 0;
        case N_var:
                return p->u.id;
        case N_neg:
                return max_var(p->u.son);
        case N_and: case N_or: case N_implies:
                return max(max_var(p->u.bin.sonL), max_var(p->u.bin.sonR));
        default:
                // This case should be impossible
                assert (1!=1);
                return 0;
        }
}
```

1 Tâche 1 : évaluation des formules propositionnelles

Cette tâche consiste à programmer un *évaluateur* (ou *interpréteur*) des formules propositionnelles. Concrètement, il s'agit d'implémenter la fonction `eval` du fichier `prop.c`.

```
int eval(Prop p, set env);
```

Étant donné un environnement, c'est-à-dire une fonction associant une valeur booléenne à chaque nom de variables, *l'évaluation* d'une formule X est le booléen obtenu lorsqu'on remplace dans X chaque nom de variable par sa valeur dans l'environnement (et qu'on calcule l'expression booléenne sans variable ainsi obtenue). [4]

Sur l'exemple introductif "`(t & 1) > -2`", si l'environnement associe `t` aux variables 1 et 2, alors la formule s'évalue sur le booléen "`(t & t) > -t`" qui vaut "`t > f`" c'est-à-dire "`f`". Si l'environnement associe `t` à 1 et `f` à 2, alors la formule s'évalue sur "`(t & t) > -f`" qui vaut "`t > t`" c'est-à-dire "`t`".

Concrètement, l'environnement `env` est une valeur de type `set` défini dans le fichier `set.h` : la valeur associée à une variable `v` dans cet environnement correspond au résultat de `in(v,env)` qui teste si `v` appartient à l'ensemble `env`.

```
int in(int v, set env);
```

Il faut ensuite tester votre fonction `eval` à partir du programme fourni `tests.c`. Après compilation avec `make`, le lancement de la commande `./tests` doit afficher quelque chose comme :

```
### 10 BASIC PROP TESTS PASSED
```

Si le programme affiche une erreur *après* ce message, c'est normal. Par contre si ce message ne s'affiche pas, il y a une erreur dans votre fonction `eval` que vous pouvez sans doute comprendre en regardant le message d'erreur et éventuellement le code source de `tests.c`. Si besoin, vous pouvez insérer du code de débogage dans `tests.c`, notamment dans la partie de la fonction `main` prévue à cet effet (en modifiant la constante `DEBUG_LEVEL` en tête de fichier).

2 Tâche 2 : syntaxe des formules en forme normale négative

Dans cette section, on s'intéresse à une forme de formules simplifiées appelée *forme normale négative* (ou NNF acronyme de *Negative Normal Form*). Une formule **Prop** est en NNF si et seulement si
- elle n'a pas de sous-formule stricte syntaxiquement constante (mais la formule elle-même peut être réduite à une constante),
- et, elle ne contient pas l'opérateur binaire `'>'`,
- et, chaque opérateur `'-'` s'applique directement à un nom de variable.

Ainsi, l'exemple introductif n'est pas une NNF. Mais il a une NNF logiquement équivalente : "`-1 | -2`".

Cette tâche consiste à vérifier qu'une formule **Prop** est une NNF en complétant le code fourni dans le fichier `prop.c` pour les fonctions suivantes :

```
/* decide if p is a non-constant Nnf */
static int is_Ncst(Prop p);

/* decide if p is a Nnf */
int is_Nnf(Prop p);
```

Comme à la tâche 1, testez votre implémentation à l'aide de `./tests` qui affiche en cas de succès :

```
### 10 check_isNnf TESTS PASSED
```

4. La terminologie utilisée ici correspond s'emploie fréquemment pour les langages d'expression, comme les *expressions arithmétiques*. En logique, on utilise *interprétation* ou *modèle* à la place de *environnement*. Et l'évaluation correspond à la relation de *satisfaisabilité* : "*un modèle satisfait une formule*" revient donc au fait que l'évaluation de cette formule sur ce modèle répond *vrai*.

3 Tâche 3 : construction de formules en forme normale négative

La tâche 4 consiste à transformer n'importe quelle formule **Prop** en une NNF logiquement équivalente. On formalise la syntaxe des NNF au travers une signature définissant une sorte **Nnf**. Tout d'abord, on appelle *littéral* ("*literal*" en anglais) une formule qui est soit réduite à un nom de variable, soit la négation d'un nom de variable. Ici, un tel littéral est directement codé par un entier non nul. On introduit donc la *sorte externe* **NNInt** comme ensemble des entiers non nuls.

On introduit aussi les *sortes internes* **Ncst** (pour désigner les NNF non constantes) et **Nnf** (pour désigner les NNF quelconques). On exprime ainsi directement les contraintes syntaxiques des formules NNF par des règles de typage.

$$
\begin{aligned}
\mathit{true} &: \mathbf{Nnf} \\
\mathit{false} &: \mathbf{Nnf} \\
\mathit{cast} &: \mathbf{Ncst} \to \mathbf{Nnf} \\
\mathit{and} &: \mathbf{Ncst} \times \mathbf{Ncst} \to \mathbf{Ncst} \\
\mathit{or} &: \mathbf{Ncst} \times \mathbf{Ncst} \to \mathbf{Ncst} \\
\mathit{literal} &: \mathbf{NNInt} \to \mathbf{Ncst}
\end{aligned}
$$

La signification des constructeurs se déduit directement de leur nom. En particulier, une formule $\mathit{cast}(X)$ représente exactement la même formule que X.

3.1 La signature "smart" des Nnf

Pour préparer la tâche 4, on utilise cependant dans le code C une signature moins restrictive que la précédente. Elle autorise en effet les constantes dans les sous-formules (mais interdit toujours les négations sauf sur un littéral). Ici, **NNInt** est codé par un `int` non nul.

```
typedef const struct Nnf_Node *Nnf;
Nnf nnf_true();
Nnf nnf_false();
Nnf nnf_and(Nnf, Nnf);
Nnf nnf_or(Nnf, Nnf);
Nnf nnf_literal(int);
```

En interne cependant, les fonctions `nnf_and` et `nnf_or` éliminent les constantes qui se trouveraient en argument. En anglais, on les qualifie de "*smart constructors*" parce qu'elles ne font pas que construire un AST, elles évaluent aussi partiellement leur arguments. Typiquement, `nnf_and(nnf_true(), p)` retourne `p` et `nnf_and(nnf_false(), p)` retourne `nnf_false()`.

L'unique traitement offert ici pour une `Nnf` consiste à la traduire en `Prop`

```
Prop to_Prop(Nnf);
```

3.2 À faire

Le fichier fourni `nnf.c` redéfinit le type `Nnf` comme un synonyme de `Prop`. L'intérêt de cette définition est que l'implémentation de la fonction `to_Prop` est triviale : c'est l'identité.

Mais à l'extérieur de ce fichier, ces deux types sont incompatibles : on ne peut construire des `Nnf` qu'au travers des "*smart constructors*" précédents, qui sont déclarés dans le fichier `nnf.h`.

Vous devez implémenter ces *smart constructors* dans `nnf.c` de manière à garantir que les formules construites vérifient les contraintes de la section 2, et en utilisant l'hypothèse que les *smart constructors* sont utilisés en conformité avec le typage de `nnf.h`.

Comme à la tâche 1, testez votre implémentation à l'aide de `./tests` qui affiche en cas de succès :

```
### 10 BASIC NNF TESTS PASSED
```

4 Tâche 4 : mise en forme normale négative

La tâche 4 consiste à implémenter l'algorithme qui permet de traduire n'importe quelle formule de type `Prop` en une formule de type `Nnf` qui soit logiquement équivalente dans la fonction suivante :

```
Nnf to_Nnf(Prop p);
```

Ainsi, pour tout `p` de type `Prop` et tout `env` de type `set`, on veut

`eval(p,env)==eval(to_Prop(to_Nnf(p)),env)`

On attend aussi que le coût de votre algorithme `toNnf` soit linéaire en fonction du nombre de nœuds traversés (et même, on attend un seul parcours de l'arbre). De même, la taille de l'arbre produit (en nombre de nœuds) doit aussi être linéaire.

L'interface de `Nnf` fournie par le module `nnf` et réalisée en tâche 3 s'occupe déjà de propager les constantes `true` et `false`. On n'a donc à s'occuper ici que de la propagation de la négation vers les feuilles. Concrètement, il faut implémenter la fonction suivante du fichier `prop.c` qui est paramétrée par un booléen `ng` mémorisant si on doit propager une négation vers les feuilles ou pas, sachant qu'une double négation équivaut à aucune négation :

```
static Nnf to_Nnfx(Prop p, int ng);
```

Ainsi, `to_Nnf(p)` est implémentée par `to_Nnfx(p, 0)` et `to_Nnfx(p, 1)` retourne une `Nnf` logiquement équivalente à `to_Nnf(neg(p))`.

Lorsque la propagation d'une négation traverse le nœud un *and* ou un *or*, on utilise les lois de De Morgan (voir http://en.wikipedia.org/wiki/De_Morgan's_laws) : un *and* se transforme en *or* et réciproquement. Pour le nœud *implies*, il suffit d'utiliser l'équivalence logique $implies(X, Y)$ équivaut à $or(neg(X), Y)$ pour se ramener aux cas précédents.

Comme à la tâche 1, testez votre implémentation à l'aide de `./tests` qui affiche en cas de succès :

```
### ALL PASSED
```

5 Tâche 5 : analyse syntaxique en notation préfixe

La notation préfixe consiste à noter l'AST en remplaçant le nom de constructeur par la notation qui lui correspond (e.g. "*and*" est affiché comme "`&`", "*var*" est affiché comme la chaîne vide), et sans parenthèse, en séparant les nombres consécutifs avec au moins un blanc. Ainsi, l'AST de l'exemple introductif "$implies(and(true, var(1)), neg(var(2)))$" est noté "`> & t 1 - 2`".

La tâche 5 consiste à programmer un *analyseur syntaxique* qui réalise en gros la réciproque : détecter si une suite de caractères sur l'entrée standard correspond à un AST de type `Prop` en notation préfixe, et dans ce cas retourner cet AST, sinon lever une erreur. La syntaxe reconnue par l'analyseur autorise des *commentaires* : ceux-ci commencent par le caractère `'#'` et se terminent en fin de ligne. Elle autorise un nombre arbitraires de *blancs* (caractère espace, ou tabulation, ou retour à la ligne) entre les *mots* consituants la formule. Dans la suite, on désigne ces mots sous le vocable de *lexèmes* (ou *tokens* en anglais). Un tel lexème est donc soit un nom de variable (constitué d'une suite de chiffres en base 10), soit un caractère parmi "`&|>-tf`".

Des exemples de fichiers en notation préfixe sont donnés dans le répertoire `tests_parsers/prefix/`. Les fichiers commençant par "`ok_`" doivent être acceptés par l'analyseur, alors que ceux commençant par "`ko_`" doivent être rejetés (ils contiennent des erreurs de syntaxe). Le fichier `ok_exparser.prop` contient par exemple :

```
# exemple de formule en notation préfixe
&
 421
 -
  f
```

Il est constitué de 4 lexèmes : `&`, `421`, `-` et `f`. Et l'AST correspondant est à cette formule est : $and(var(421), neg(false))$.

Pour simplifier la programmation de l'analyseur syntaxique, celui-ci est décomposé en deux parties : l'analyseur lexical (ou *lexer* en anglais) qui transforme la suite de caractères en suite de lexèmes (cf. section 5.1) ; et l'analyse syntaxique proprement dite qui construit l'AST à partir de la suite de lexèmes (cf. section 5.2)

5.1 Analyse lexicale

On fournit ici l'analyseur lexical dans un module `lexer`. En interne, pour chaque lexème *concret* (c'est-à-dire une certaine suite de caractères) reconnu, l'analyseur construit un lexème *abstrait* : c'est-à-dire, une valeur du type fourni `token`. Ainsi, l'analyseur lexical fonctionne comme une sorte d'itérateur Java : il fournit une fonction `next` qui déplace la tête de lecture jusqu'au prochain lexème et le retourne. La suite de caractères est lue dans l'entrée standard.

```
token next(int *v);
```

Ici, un lexème abstrait spécial `END` joue le rôle de sentinelle de fin de fichier : il sert à exprimer que la lecture du fichier d'entrée est finie. Lorsque le lexème lu par `next` est une variable (le token correspondant vaut `VAR`), alors le paramètre `v` est modifié avec la valeur de la variable lue.

Votre tâche consiste ici à compléter le fichier fourni `lexer.c`. Pour tester votre implémentation, utilisez le fichier de test fourni `tests_parsers/testlexer.sh`. Pour déboguer, vous pouvez utiliser le programme fourni `debug_lexer.c`. Par exemple, la commande

```
./debug_lexer < tests_parsers/prefix/ok_touslexemes.prop
```

doit afficher la sortie attendue dans `tests_parsers/lexer_test.expected` (c'est ce que vérifie `testlexer.sh`). Sinon, vous pouvez aussi faire des tests interactifs (sans créer de fichier) avec la commande

```
./debug_lexer
```

Il faut juste taper les deux touches "Ctrl-d" pour fermer l'entrée standard.

5.2 Principe de l'analyse syntaxique en notation préfixe

L'analyseur syntaxique invoque donc la fonction `next` du lexer pour construire l'AST au fur et à mesure de la lecture du fichier d'entrée : la suite des lexèmes n'est donc pas stockée en mémoire, et la lecture du fichier s'interrompt à la première erreur detectée dans le fichier.

Pour programmer l'analyseur syntaxique, on se base sur la propriété suivante de la notation préfixe :

> *Pour toute suite de lexèmes u, il existe* **au plus** *une suite de lexèmes v qui est un préfixe de u et qui correspond à l'écriture en notation préfixe d'un AST de sorte* **Prop**.

Par exemple, si u correspond à "`& t 1 - 2`", alors l'unique v possible correspond à "`& t 1`". Par contre, si u correspond à "`& t`", il n'y a aucun v possible.

Cette propriété permet en effet de programmer récursivement une fonction `parse_rec`, qui étant donné la suite u des lexèmes *restant à lire*,
- s'il existe un préfixe v de u tel que v est l'écriture en notation préfixe d'un AST de type `Prop`, alors `parse_rec` retourne cet AST et la tête de lecture du lexer après l'appel se trouve sur le premier lexème après la suite v dans u ;[5]
- si un tel v n'existe pas alors, `parse_rec` stoppe l'exécution du programme avec un message d'erreur approprié.

```
void parse_rec (Prop *p) ;
```

L'unicité du v recherché permet en effet une programmation récursive simple de cette procédure : le premier lexème de v (et donc u) n'est pas la sentinelle de fin (sinon on lève l'exception) et correspond à un nœud dont on connaît le nombre de fils : 0 pour `'t'` ou `'f'` ou un nom de variable, 1 pour `'-'` et 2 pour les lexèmes parmis "`&|>`"). Pour chaque fils attendu, on effectue en séquence un appel récursif à `parse_rec` afin de récupérer l'AST correspondant. On retourne alors l'AST obtenu par assemblage du nœud initial et de ses fils.

La fonction principale de l'analyseur syntaxique se contente d'appeler `parse_rec` et de vérifier qu'en sortie de l'appel, on a bien atteint la fin du fichier.

```
Prop parse() ;
```

5.3 À faire

Le programme `prefix_prop` propose un certain nombre de traitements des formules propositionnelles en notation préfixe. Lancer "`./prefix_prop`" pour voir la liste des traitements proposés. L'action effectuée sur l'AST (le "backend") est déjà programmée dans le module `cmdline` en utilisant les traitements programmés dans les sections précédentes.

Votre travail consiste donc à implémenter le "front-end" qui construit l'AST à partir des caractères lus dans l'entrée standard. Une fois que le fichier `lexer.c` est complété (cf. section 5.1), il faut compléter la fonction `parse_rec` (décrite ci-dessus) fournie dans le fichier `prefix_prop.c`.

Vérifiez que l'analyseur fonctionne en lançant :

```
tests_parsers/testprefix.sh tests_parsers/prefix/*.prop
```

Lancer `testprefix.sh` sans argument pour voir son mode d'emploi.

6 Tâche 6 : analyse syntaxique en notation infixe

Pour l'analyseur en notation infixe, on réutilise le lexer défini en section 5.1 en ajoutant deux lexèmes pour "`(`" et "`)`". Le fonctionnement de l'analyseur va donc fortement ressembler à celui décrit en section 5.2, mais pour une syntaxe concrète plus complexe.

Pour écrire cet analyseur, il faut commencer par effectuer une série de transformations de BNF à partir de la spécification initiale de l'analyseur donnée en 6.1. Ici, il faut s'appuyer sur les concepts vus en cours (qui ne sont pas redétaillés ici).

5. Autrement dit, la suite u' des lexèmes qui restent à lire après l'appel récursif vérifie $u = v\ u'$.

Précisons qu'il existe des outils pour faire ces transformations automatiquement, de sorte que le développeur peut directement écrire son analyseur dans un format proche de celui de la spécification donnée en 6.1. Mais le but de ce TP est justement de faire ces transformations "à la main" sur un petit exemple afin d'acquérir la compréhension du fonctionnement de ces outils qui s'avère indispensable quand on les utilise sur des exemples plus complexes.

6.1 Spécification de la syntaxe infixe

La spécification de l'analyseur infixe est donnée par la BNF attribuée ci-dessous et les règles de précédence données plus loin. Cette BNF comporte un seul non-terminal (et axiome) noté **P**. Ses terminaux sont VAR et tous les symboles entre apostrophes (ils correspondent aux tokens reconnus par l'analyseur lexical). Le système d'attribut spécifie les AST associés aux mots reconnus par l'analyseur.
Les profils des symboles de la BNF sont **P**↑**Prop** VAR↑**Pos**
La BNF attribuée est

P↑p	::=	`'t'`	$p := true$
	|	`'f'`	$p := false$
	|	VAR↑v	$p := var(v)$
	|	`'-'` **P**↑p_1	$p := neg(p_1)$
	|	`'('` **P**↑p `')'`	
	|	**P**↑p_1 `'&'` **P**↑p_2	$p := and(p_1, p_2)$
	|	**P**↑p_1 `'|'` **P**↑p_2	$p := or(p_1, p_2)$
	|	**P**↑p_1 `'>'` **P**↑p_2	$p := implies(p_1, p_2)$

Cette BNF ci-dessus est ambiguë : pour un même mot, plusieurs AST peuvent être synthétisés. Pour désambiguïser, on restreint les arbres d'analyses possibles avec les règles de précédences ci-dessous :
– les opérateurs ont les niveaux de précédence suivants

opérateur	`'>'`	`'|'`	`'&'`	`'-'`
précédence	3	2	1	0

– comme dans tout langage d'expression, les constantes, les variables et les parenthèses sont au niveau de précédence 0
– tous les niveaux de précédences sont associatifs à droite

6.2 A faire : transformer la BNF en une BNF non-ambiguë

Votre premier travail consiste à écrire une BNF attribuée non-ambiguë, qui reconnaisse le même langage que celle donnée en section 6.1, et qui, pour chaque mot de ce langage, synthétise le même AST lorsqu'on restreint la BNF 6.1 aux arbres d'analyse qui satisfont les règles de précédence.

Pour exprimer les contraintes de précédence comme des règles de BNF, il faut introduire un non-terminal $\mathbf{P}_n$ par niveau de précédence n (avec $0 \leq n \leq 3$) où chaque non-terminal a la signification suivante :

> Le langage reconnu par $\mathbf{P}_n$ est le langage reconnu par **P** (de la BNF en 6.1) tels que tout opérateur de niveau de précédence strictement supérieure à n apparaît uniquement dans un sous-mot de forme `'('` u `')'` où u est lui-même un mot de **P**.

Ainsi, $\mathbf{P}_3$ correspond à l'ensemble des mots reconnus par **P** (et donc est l'axiome de votre BNF). Et, pour $0 \leq n < 3$, on a $\mathbf{P}_n \subset \mathbf{P}_{n+1}$. De plus, si u_1 et u_2 sont dans **P**, alors un mot de la forme "u_1 `'|'` u_2" peut être reconnu par $\mathbf{P}_2$ (à condition que u_1 et u_2 soient eux-mêmes dans $\mathbf{P}_2$) mais pas par $\mathbf{P}_1$.

On ne demande pas de prouver à ce niveau que la BNF obtenue est non-ambiguë. Cela va être vérifié de manière indirecte en la transformant en BNF LL(1).

6.3 A faire : transformer la BNF non-ambiguë en une BNF LL(1)

Vous devez maintenant transformer la BNF de la section 6.2 en BNF LL(1) : vous devez obtenir une BNF attribuée LL(1) qui reconnaît le même langage et synthétise les mêmes AST que celle de la section 6.2.

Comme il n'y a que des opérateurs associatifs à droite, il suffit en principe de factoriser les règles à gauche. En cas d'opérateurs associatifs à gauche, il faudrait éliminer les récursions immédiates à gauche.

Vérifiez que la BNF est bien LL(1) en effectuant le calcul de directeurs.

6.4 A faire : implémenter l'analyseur LL(1)

Votre travail consiste maintenant à implémenter l'analyseur spécifié section 6.1 en le dérivant de la BNF LL(1) obtenue en section 6.3. Concrètement, il s'agit de compléter le fichier fourni `infix_prop.c`, en modifiant la fonction `parse` de manière à ce qu'elle accepte tout préfixe de l'entrée qui est un sous-mot de $\mathbf{P}_3$ et produise l'AST associé. Comme vu en CTD, il faut éventuellement introduire une fonction spécifique pour chaque autre non-terminal de la BNF LL(1). Vérifiez que l'analyseur fonctionne en lançant :

```
tests_parsers/testinfix.sh tests_parsers/infix/*.prop
```

7 Annexe : description des fichiers fournis

Fichiers qu'il n'est pas nécessaire de modifier (par ordre d'utilisation)

`Makefile` pour compiler ; ce fichier décrit aussi les dépendances entre fichiers (à consulter donc).

`sorts.h` déclare les types `Prop` et `Nnf`.

`prop.h` déclare les opérations sur le type `Prop`.

`set.h`, `set.c` définissent les opérations sur le type `set`.

`tests.c` teste les fonctions des modules `prop` et `nnf`.

`nnf.h` déclare les opérations sur le type `Nnf`.

`debug_lexer.c` permet de tester l'analyseur lexical.

`tests_parsers/` contient des scripts et des jeux de tests pour tester les analyseurs.

`cmdline.h`, `cmdline.c` définissent l'analyse de la ligne de commande sélectionnant le traitement à réaliser sur la formule lue par les analyseurs.

Fichiers à compléter

`prop.c` implémente les opérations sur le type `Prop`.

`nnf.c` implémente les opérations sur le type `Nnf`.

`lexer.h` déclare l'interface de l'analyseur lexical et le type `token` (à modifier à la tâche 6).

`lexer.c` implémente l'analyseur lexical.

`prefix_prop.c` implémente l'analyseur en notation préfixe.

`infix_prop.c` implémente l'analyseur en notation infixe.