Premier rendu :

Binary Decision Diagrams

À rendre pour le ***21 mars 2017 à 23h59*** *au plus tard. Envoyez une archive* qui compile *à* `aurore.alcolei@ens-lyon.fr, bertrand.simon@ens-lyon.fr, daniel.hirschkoff@ens-lyon.fr`

1 Un *front-end* pour les formules logiques

Votre programme devra prendre en entrée un fichier contenant une formule logique.

Syntaxe des formules logiques. Voici comment sont notés les connecteurs logiques que votre programme devra savoir reconnaître.

- parenthèses `( ··· )`
- conjonction `/\`
- disjonction `\/`
- ou exclusif (*xor*) `X`
- implication `=>`
- équivalence `<=>`
- négation `~` (tilde)
- variables propositionnelles : les variables sont représentées par un entier strictement positif (`7` désigne x_7). Un entier négatif désigne la négation d'une variable.

  variable niée : on aura le droit d'écrire `-12` pour `~12`, et même `~~-3` (une triple négation), mais pas `-~3`
- fin de formule : `0` (*zéro*) suivi d'un retour à la ligne représente la fin de la formule. N'importe quoi peut suivre dans le fichier, et ce ne sera pas pris en compte.

La conjonction (représentée par `/\`) a une priorité plus grande par rapport à la disjonction, et la négation a une priorité encore plus grande. La priorité de l'implication est plus petite que la priorité de la disjonction, la priorité de l'équivalence est plus petite que la priorité de l'implication. Le ou exclusif est entre la disjonction et l'implication.

Ainsi, par exemple, `~3 => 5 \/ -2 0 <retour chariot>` représente $\neg x_3 \Rightarrow (x_5 \vee \neg x_2)$.

Ingérer les formules logiques. Faites une copie des fichiers de l'archive qui vous est fournie pour les expressions arithmétiques (partez plutôt de la version "enrichie", où est traité le `-`). Repartez de ces fichiers pour faire un analyseur pour les formules logiques, écrites suivant la syntaxe donnée plus haut.

Il vous faudra adapter à peu près tous les fichiers :

- le fichier où est défini le type pour les expressions arithmétiques (qui deviennent des formules logiques) ;
- le fichier pour l'analyse lexicale, où sont donnés les lexèmes ;
- le fichier pour l'analyse syntaxique, où sont données les règles de grammaire.

Il vous est conseillé de programmer une fonction d'affichage simple afin de pouvoir vérifier que la formule reconnue est la bonne. Il est judicieux de traiter les opérateurs petit à petit, en vérifiant incrémentalement que ça marche, plutôt que de récupérer tous les bugs d'un coup.

Les conflits shift/reduce. Votre analyseur ne devra engendrer aucun conflit (shift/reduce, reduce/reduce) à la compilation. Clairement, l'expérience est l'ingrédient principal pour apprendre à les gérer.

NB : vous *devez* utiliser `ocamllex` et `ocamlyacc` pour l'analyse des formules en entrée, même si vous êtes capables de programmer l'analyseur directement. Le but est d'apprendre à utiliser cette technologie.

2 Calculer un BDD

Votre programme devra construire, à partir de la formule saisie en entrée, un BDD la représentant.
Voici comment se déclinera cet exercice en fonction des binômes :

`D` Les **débutants** feront semblant d'avoir un BDD, mais pourront se contenter de construire l'arbre de décision de la formule, sans faire de partage (bien entendu, si en plus vous faites du partage, c'est d'autant mieux !).

`I` Les **intermédiaires** pourront commencer par un programme qui fait tout sauf le partage (menu des débutants), et, dans un second temps, modifieront le code pour faire du partage.

Il faudra au passage que la structuration du code se prête de manière raisonnablement naturelle à ce développement en deux étapes : l'idée est qu'il faut éviter que l'ajout du partage signifie modifier plusieurs fichiers à beaucoup d'endroits différents.

Vous expliquerez dans le fichier README comment vous avez pris en compte cet aspect des choses dans la structuration du code, et comment se traduit le passage de l'absence de partage au partage du point de vue du code.

`A` Les **avancés** implémenteront directement la traduction avec partage (sans passer par une représentation coûteuse, que l'on réduit ensuite).

Dans la mesure du possible, il sera bon de faire une librairie *générique* pour le partage maximal, qui permet de calculer les BDDs, et pourrait être appliquée à d'autres structures de données (cf. plus bas, transformation de Tseitin).

Réordonnancement dynamique. La littérature sur les BDDs indique qu'une technique permettant d'améliorer l'*expressivité* des programmes (au sens où elle agrandit l'espace des formules pouvant être traitées) consiste à faire du réordonnancement dynamique des formules, suivant une approche appelée en anglais `sifting`. Voir à ce propos les références disponibles depuis la page www du cours.

Implémentez cette technique (en fixant un critère qui déclenche une phase de réordonnancement dynamique), et tâchez d'évaluer dans quelle mesure elle permet d'améliorer les choses. Rendez compte de votre travail d'implémentation et de vos observations dans le fichier README.

Comme toujours, il vaut mieux rendre un programme où tout marche bien et a été testé mais où le réordonnancement dynamique n'est pas traité plutôt qu'un programme qui tente de faire le réordonnancement dynamique, mais qui n'est pas terminé proprement.

3 Satisfaction de formules

3.1 Transformation de Tseitin

Programmer la transformation de Tseitin, pour obtenir une formule au format SAT (forme normale conjonctive, *CNF* en anglais) à partir de la formule saisie en entrée.

`A I D` Tous les binômes devront traiter cette partie.

`A` Les binômes avancés devront en outre s'assurer qu'ils construisent une formule en CNF avec le partage maximum.

3.2 BDD contre DPLL : s'appuyer sur `minisat` pour debugger

Une fois construite une formule en CNF, il vous faudra l'écrire dans un fichier, au format DIMACS. L'exemple ci-dessous devrait suffire pour comprendre l'essentiel de ce format.

```
p cnf nvar nclau
-4 2 5 0
2 -1 0
```

`nvar` : nombre de variables (ici 5)
`nclau` : nombre de clauses (ici 2)
$(\overline{x_4} \vee x_2 \vee x_5) \wedge (x_2 \vee \overline{x_1})$

Le but est ensuite d'appeler le programme `minisat` (disponible sur les machines des salles libre-service) pour tester la satisfiabilité de la formule engendrée (on rappelle que la formule calculée via la transformation de Tseitin et la formule de départ sont équisatisfiables).

Si on lance la commande `minisat form1.cnf sortie.txt`, on récupère dans le fichier `sortie.txt` la sortie de l'appel à `minisat` sur le fichier `form1.cnf`.

1. Lorsque la réponse est `UNSAT`, à quoi doit ressembler le BDD engendré ?
2. Lorsque la réponse est `SAT`, la seconde ligne du fichier `sortie.txt` contient une affectation des variables qui satisfait la formule.

Utilisez les deux observations ci-dessus pour programmer une fonction qui vous permettra de tester si tout marche bien dans votre programme. En particulier, si la formule est satisfiable, il vous faudra vérifier que l'affectation fournie par `minisat` est acceptée par votre BDD.

Le fichier `runminisat.ml` vous propose un exemple pour l'appel à `minisat` depuis Caml. Vous pouvez vous en inspirer pour votre rendu.

`A I D` Tous les binômes sont sensés programmer cette "boucle via `minisat`", même si pour les débutants l'utilisation du "faux BDD" limitera sans doute considérablement la complexité des exemples pouvant être traités.

4 Structuration du code, tests, format de l'exécutable

Architecture. Il vous est fortement recommandé de commencer par concevoir le squelette de votre programme, en réfléchissant à la répartition des composantes entre fichiers. Cela devra vous permettre de répartir le travail au sein du binôme, et de fixer des échéances.

Il sera particulièrement utile de commencer par écrire les types pour les données qui jouent un rôle central dans le projet. Ainsi, le travail sur les formules devra pouvoir commencer avant que la partie "lex-yacc" ne soit terminée, en testant à partir de petits exemples écrits à la main.

Vous pouvez fournir un schéma (par exemple, en ASCII moche) expliquant la structuration de votre code.

Tests. Comme toujours, prévoyez des fichiers de test, et incluez-les dans votre rendu. Commencez par tester avec de petits exemples, qui vous aideront pour debugger. Lorsque vous serez raisonnablement confiants quant à la correction de votre code, vous pourrez passer à une phase où vous le poussez dans ses retranchements afin de comprendre quel type de formule le fait "ramer".

Votre ensemble de tests devra en particulier comporter les exemples suivants :

- la fonction "parité", qui renvoie 0 si elle a un nombre pair d'arguments égaux à un, et 1 sinon ;
- des fonctions booléennes correspondant à des opérateurs simples (additionneurs, multiplieurs), de tailles variables ;
- des rotations (on envoie x_1 sur x_2, x_2 sur x_3, x_n sur x_1) ;

- la formule du principe des tiroirs : si on range $n+1$ pommes dans n tiroirs, il existe au moins un tiroir qui contient deux pommes :

$$\Big(\bigwedge_{p=1}^{n+1}\bigvee_{t=1}^{n} x_{p,t}\Big) \quad \Rightarrow \quad \bigvee_{t=1}^{n}\bigvee_{p=1}^{n+1}\bigvee_{q=1}^{n+1}(x_{p,t}\wedge x_{q,t}) \ .$$

Format. Le programme exécutable qui est engendré à la compilation doit *obligatoirement* s'appeler `f2bdd`. Il doit être appelé de la manière suivante : `./f2bdd toto.form`
où `toto.form` est un fichier contenant une formule en entrée.

Par ailleurs, il faudra que `f2bdd` accepte les options suivantes :

- `-tseitin blabla.cnf` pour traduire la formule en entrée vers le format SAT, et stocker le résultat dans le fichier `blabla.cnf` (au format DIMACS)
- `-minisat` pour traduire la formule en entrée vers le format SAT, récupérer la sortie de `minisat` sur cette formule, et afficher un message de satisfaction ou de désespoir (pour le cas où `minisat` signale une erreur dans votre code).
  - Le message de satisfaction sera `OK`, le message de désespoir sera `ARGH`, suivi éventuellement de quelques informations décrivant un diagnostic sur ce qui a marché ou n'a pas marché.
  - On peut appeler l'option `-minisat` toute seule. Si on appelle les deux options `-minisat` et `-tseitin` (dans cet ordre — vous pouvez aussi autoriser l'autre ordre si vous voulez), on pourra de plus récupérer dans le fichier passé en argument à `-tseitin` le résultat de la traduction de la formule donnée en entrée.
- Vous pouvez également programmer une option `-debug`, qui affichera des informations sur l'exécution du programme. Vous pouvez même faire un `-debug k`, où `k` est un entier, pour fournir une information plus ou moins détaillée.
- Si l'on ne passe aucune option, le programme affiche simplement un entier, le nombre de nœuds contenus dans le BDD (nœuds internes et feuilles).

Exemples : `./f2bdd -minisat titi.form` `./f2bdd -minisat -tseitin tutu.form`

Un exemple simple de programme gérant la ligne de commande et les options est disponible depuis la page www du cours.

5 Tout le monde : en quoi consiste le rendu

Vous devez envoyer avant la date limite une archive qui compile, à
`aurore.alcolei@ens-lyon.fr, daniel.hirschkoff@ens-lyon.fr, bertrand.simon@ens-lyon.fr`.

Incluez dans votre archive un fichier README, contenant des commentaires sur votre code : indiquez notamment s'il y a des parties que vous n'avez pas traitées, s'il y a des bugs/imperfections dont vous êtes conscient(e).

Décrivez comment le travail s'est réparti au sein du binôme.

Parlez également des tests que vous avez faits : comment on peut les rejouer, et quelles conclusions on peut en tirer. En particulier, a-t-on une idée de la taille maximale des formules que peut traiter votre programme en entrée ?

Comme au DM, rendez quelque chose de propre. "Propre" signifie en particulier :

- du code bien structuré et bien commenté ;
- des programmes testés, avec les fichiers de test que vous avez utilisés ;
- un fichier README.