Projet : Réécriture

1 Contexte

Le but du projet est d'écrire un interpréteur pour des systèmes de réécriture. A cause de la similarité des systèmes de réécriture et des langages fonctionnels, vous écrivez donc en même temps un interpréteur d'un petit langage fonctionnel, similaire à Caml, mais avec des différences notables.

1.1 Règles de réécriture

Un petit exemple d'un système de réécriture se trouve en fig. 1.

```
rules
  r1: f X X -> X;
  r2: a -> b;

trace r1, r2;

simp [left_outer]   f (f a a) a
```

FIGURE 1 – Système de réécriture

Cette description est composée :
— `rules` : d'une liste de règles,
— `trace` : des noms des règles à tracer,
— `simp` : du terme à simplifier (ici : `f (f a a) a`), et de la stratégie de simplification à utiliser (entre `[ ... ]`).

1.2 Simplification

Vous pouvez sauvegarder la description de la fig. 1, par exemple dans le fichier `Examples/ex1.rew`. Le but est de lancer votre prouveur sur la ligne de commande pour simplifier le terme :

```
> ./eqprover Examples/ex1.rew
Simplifying: ((f ((f a) a)) a)
--r1-->
((f a) a)
--r1-->
a
--r2-->
b
```

En choisissant une autre stratégie de réduction :

```
simp [right_inner]   f (f a a) a
```

on obtient une autre dérivation :

```
Simplifying: ((f ((f a) a)) a)
--r2-->
((f ((f a) a)) b)
--r2-->
((f ((f a) b)) b)
--r2-->
((f ((f b) b)) b)
--r1-->
((f b) b)
--r1-->
b
Result: b
```

Votre travail consistera à écrire les fonctions essentielles qui effectuent cette simplification des termes (voir sect. 4). Vous pouvez utiliser des types de données prédéfinis (sect. 2) et un parser (sect. 3) pour analyser la syntaxe du fichier d'entrée.

2 Structures de données

Pour la *syntaxe abstraite* (les termes effectivement manipulés par Caml), nous utilisons le type de termes déjà vu à plusieurs reprises (voir fichier `lang.ml`) :

```
type term =
    Const of string
  | Var of string
  | Appl of term * term
```

Pour écrire des règles, vous pouvez utiliser une *syntaxe concrète* plus concise et agréable. Il pourrait donc sembler que les termes ont une structure plus complexe. Pourtant, il s'avère que la structure `term` est entièrement suffisante ; le parser traduit la syntaxe concrète vers la syntaxe abstraite, voir sect. 3.

Reprenons maintenant les autres éléments d'une description d'un système de réécriture comme par exemple dans la fig. 1.

Une règle `n: l -> r` a un nom `n` (une chaîne de caractères) et deux termes `l` et `r` :

```
type rule = Rl of string * term * term
```

Nous verrons (sect. 3) que le nom est optionnel ; pour une règle sans nom, `n` est la chaîne vide `""`.

Pour spécifier les règles à tracer, on peut

— dire "toutes les règles" (y compris sans nom), ce qui s'écrit **`trace *;`**

— les énumérer explicitement (comme en fig. 1), par exemple **`trace r1, r2;`**

— dire "aucune règle", et c'est un cas particulier du précédent : **`trace ;`**

Le type pour spécifier la trace est :

```
type tracespec =
    TraceAll
  | TraceSome of string list
```

où `TraceAll` correspond à "toutes les règles" et `TraceSome` aux deux autres cas, avec une liste éventuellement vide.

Avec ceci, nous avons tous les ingrédients pour décrire la structure d'un système de réécriture, qui est composé (voir fig. 1) :

— d'une liste de règles,

— d'une spécification de traces

— du nom d'une stratégie de réécriture (voir sect. 4.3.3 à ce sujet)

— d'un terme à réécrire

```
type rewr = Rewr of rule list * tracespec * string * term
```

3 Particularités de la syntaxe

Pour écrire des termes, nous utilisons une syntaxe s'inspirant des langages fonctionnels, mais il y a quelques différences.

Conventions lexicales Elles sont spécifiées dans le fichier `lexer.mll`.

Les identifiants commencent avec une lettre, suivi de lettres, chiffres ou souligné _.

Tous les identifiants commençant avec une minuscule sont interprétés comme des constantes (même s'il s'agit de lettres de la fin d'alphabet comme `x`). Les identifiants commençant avec une majuscule sont interprétés comme des variables. Ceci est une différence notable avec Caml, où les majuscules indiquent un constructeur. De notre part, nous ne faisons pas de différence entre constructeurs et autres noms de fonction.

Vous pouvez utiliser deux genres de commentaires : Le commentaire `//` supprime tout le reste de la ligne ; et le commentaire multi-lignes `/* ... */` supprime tout ce qui se trouve entre les balises `/*` et `*/`.

Conventions syntaxiques Elles sont spécifiées dans le fichier `parser.mly`.

L'application d'une fonction `f` à un argument `a` s'écrit `(f a)`, ce qui correspond au terme `Appl(f, a)`. En cas d'applications itérées, on peut omettre les parenthèses, comme dans `(f a b)`, ce qui équivaut à `((f a) b)`.

Les couples s'écrivent de la forme `(a, b)`. Pour les représenter, nous utilisons une fonction prédéfinie `pair`, et le terme `(a, b)` est représenté par `Appl (Appl (Const "pair", a), b)`.

Vous pouvez aussi écrire des nombres naturels (nombres décimaux). Ils sont représentés en format unaire, avec successeurs de zéro. Par exemple, `2` est `succ (succ 0)`. Pour la représentation interne en Caml, nous utilisons deux constantes prédéfinies `succ` et `zero`. Le nombre `2` est donc représenté en interne par `Appl (Const "succ", Appl (Const "succ", Const "zero"))`.

Lors de l'affichage de termes, on effectue la transformation (presque) inverse, c'est-à-dire, des applications successives de `succ` sont converties en chiffres, et pareil pour les applications de `pair`. Pourtant, les termes sont affichés avec un parenthésage complet, donc par exemple `((f a) b)` au lieu de `f a b`.

L'analyse syntaxique est effectuée par la fonction `parse` qui prend en argument le nom d'un fichier. Vous pouvez la tester comme suit, dans l'interpréteur de Caml (voir aussi annexe A.2) :

```
# parse "Examples/ex1.rew" ;;
- : Lang.rewr =
Rewr
 ([Rl ("r1", Appl (Appl (Const "f", Var "X"), Var "X"), Var "X");
   Rl ("r2", Const "a", Const "b") ..])
```

Astuce : Peut-être, les constructeurs sont préfixés du nom du module, donc `Lang.Rewr` etc., ce qui rend les termes illisibles. Pour supprimer ce préfixe, ouvrez le module avec `open Lang;;`

4 Travail à réaliser

4.1 Aspects administratifs du projet

Le projet doit être réalisé et rendu individuellement.

Lors des deux échéances (voir plus bas), vous devez déposer une archive qui contient un seul répertoire `Projet_nom`, où `nom` est votre nom de famille (sans accent ou caractères spéciaux comme des apostrophes,

sans espace, mais éventuellement avec des soulignés (_) si votre nom est composé. Le nom de l'archive (format `*tar` ou `*zip`) doit être de la forme `projet_nom.tar` ou `projet_nom.zip`.

Le répertoire contiendra les informations suivantes :

— un fichier `README` décrivant la structure du dépôt (répertoires et fichiers),
— le code source Caml,
— des exemples de systèmes de réécriture, de préférence dans un sous-répertoire `Examples`.

Le plus facile est de renommer le répertoire qui vous est fourni, et de créer, avant le dépôt, l'archive avec

```
tar cf projet_nom.tar Projet_nom
```

Le projet doit être soumis sur Moodle (voir les liens dans la section "Projet") chaque fois avant la date limite. Après cette date, Moodle coupe le site de téléchargement, une soumission ne sera plus possible. Un envoi par mail ne sera pas accepté. Par contre, vous pouvez déposer votre projet autant de fois que vous voulez avant la date limite ; uniquement la dernière soumission sera visible et sera prise en compte. En cas de problème avec Moodle, envoyez-moi un mail au moins un jour avant la date limite.

Le projet doit être programmé en Caml (voir aussi un complément d'information en annexe B) et satisfaire les exigences minimales suivantes :

— Le code Caml doit être exécutable (surtout : être compilable correctement).
— Il ne doit pas y avoir des fichiers dont les noms contiennent des caractères spéciaux, des caractères accentués ou des espaces blancs.
— Quelques cas de tests doivent être fournis avec le code, et le code doit s'exécuter correctement sur ces exemples ; voir aussi sect. 4.3.2.

Le fichier `README` doit contenir une description du travail effectué et surtout préciser quelles tâches ont été / n'ont pas été réalisées.

Le travail à effectuer est composé de deux parties, qui correspondent aux deux fournitures à rendre sur Moodle :

1. des fonctions de base (sect. 4.2), qui sont en grande partie les fonctions que vous programmez en séance de TP. Il faut les mettre au point (si vous n'arrivez pas de les terminer en séance de TP) et les tester.

   Cette partie est à déposer sur Moodle au plus tard au **8 novembre 2015 à 23h**.

2. des fonctions de réécriture et leur utilisation (sect. 4.3). Pour cette partie, on vous demande un travail plus autonome.

   Cette partie est à déposer sur Moodle au plus tard au **4 décembre 2015 à 23h** (et personne ne vous empêche de travailler sur la deuxième partie avant la date limite pour la première partie).

   Une présentation orale de vos projets aura lieu l'après-midi du 7 décembre.

4.2 Fonctions de base

4.2.1 Substitution

Les fonctions de variables libres et de substitutions ont été définies en cours, et vous avez déjà programmé une première version de ces fonctions. Il s'agit ici de mettre au point ces fonctions et de les tester exhaustivement.

Exercice 1 Écrivez la fonction `fv` qui prend un terme (élément du type `term`, voir sect. 2) et calcule l'ensemble (représenté comme liste sans doublons) des variables libres du terme.

Exercice 2 Écrivez la fonction `const_term` qui prend un terme et vérifie que le terme est constant, c.à.d., ne contient pas de variable libre. Vous utiliserez cette fonction pour vérifier que les termes à réécrire par le système de réécriture sont en fait constantes.

Exercice 3 Écrivez la fonction `apply_subst` qui prend un terme et une substitution, et applique la substitution au terme.

Il est à vous de choisir une structure de données appropriée pour représenter des substitutions, et de définir des fonctions auxiliaires (comme par exemple le traitement de listes d'association).

4.2.2 Unification

Exercice 4 Implantez une fonction d'unification. Vous pouvez utiliser la fonction d'unification vue en cours.

Variantes et particularités :

1. Vous pouvez aussi écrire une version adaptée aux besoins de ce projet : Il s'avère que nous effectuons uniquement des unifications $t_1 \stackrel{?}{=} t_2$ où le terme t_2 ne contient pas de variables. Quelles règles de l'algorithme d'unification peuvent être simplifiées ? Pourquoi est-ce que l'algorithme modifié est plus efficace ? (Commentez votre code pour répondre à la question).
2. L'algorithme d'unification discuté en TP lève une exception en cas d'échec. Or, un échec d'unification est une situation tout à fait normale lors de la réécriture et non une raison pour un arrêt du programme. Voir sect. B.1 pour des solutions possibles.

4.2.3 Infrastructure pour la réécriture

Dans cette première partie du projet, on s'intéresse à deux fonctions qui effectuent une réécriture à la racine d'un terme. La réécriture à l'intérieur d'un terme sera traitée dans la deuxième partie.

Exercice 5 Écrivez la fonction `try_rewrite` qui prend trois termes `l`, `r` et `t` (respectivement la partie gauche et droite d'une règle de réécriture `l -> r` et le terme où on veut appliquer la réécriture. La fonction renvoie :

1. `(t, false)` si une réécriture n'est pas possible, ou si une réécriture a été effectué mais le terme n'a pas été modifié (trouvez un exemple !).
2. `(t', true)` sinon, où `t'` est le terme transformé par la règle `l -> r`.

Exemple : il n'est pas possible de réécrire `f a b` avec la règle `f X X -> X`, mais la réécriture de `f (g a) (g a)` donne `g a`.

Exercice 6 Écrivez la fonction `try_rewrite_rule_list` qui prend une liste de règles (type `rule`) et un terme et le réécrit avec la première règle de la liste qui est applicable. Le résultat est un triplet :

1. terme (transformé, ou inchangé si aucune règle n'est applicable)
2. Booléen indiquant si le terme a été changé
3. nom de la règle (de type `string`) qui a été appliquée (c'est la chaîne de caractères vide si la règle n'a pas de nom).

Pour ce triplet, nous introduisons le type suivant :

```
type rewr_result = term * bool * string
```

4.3 Fonctions de réécriture

4.3.1 Réécriture dans des termes

Exercice 7 Écrivez la fonction `rewrite_in_term` qui prend comme arguments une liste de règles et un terme et effectue une réécriture dans le terme et renvoie un triplet du type `rewr_result`. Contrairement à la fonction `try_rewrite_rule_list` de l'exercice 6, la réécriture ne s'effectue pas forcément à la racine du terme à réécrire, mais peut-être à l'intérieur, en fonction de la stratégie choisie (sect. 4.3.3). Pour

la première version de la fonction, codez en dur la stragégie *leftmost-outermost* (voir les explications en début de sect. 4.3.3).

Exercice 8 Écrivez la fonction `simp` qui prend une liste de règles, une spécification de trace (de type `tracespec`, non utilisé pour l'instant) et un terme, qui applique les règles tant de fois que possible et qui renvoie la forme normale du terme dès que la réécriture ne produit plus de changements. Bien sûr, pour certaines règles et certains termes, la réécriture ne termine pas ...

Exercice 9 Écrivez la fonction `print_trace_step` qui prend comme argument une spécification de trace (de type `tracespec`), le nom d'une règle (éventuellement la chaîne de caractères vide si la règle n'a pas de nom) et un terme et affiche un bout de trace, comme indiqué en sect. 1.2.

Exercice 10 Modifiez la fonction `simp` de l'exercice 8 pour afficher des traces lors de l'exécution.

4.3.2 Tests

Écrivez des tests, c.-à-d., des systèmes de réécriture qui illustrent le fonctionnement de votre implantation.

Les exercices suivants correspondent à des programmes fonctionnels : vous interprétez donc un langage (système de réécriture) dans un autre langage (Caml). Mais il s'agit uniquement de suggestions, vous êtes libres à proposer d'autres tests de taille conséquente.

Exercice 11 Définissez, dans un système de réécriture, quelques fonctions arithmétiques élémentaires pour des nombres naturels. Utilisez pour ceci les systèmes de réécriture que vous avez vus en cours. Nous rappelons que les nombres naturels sont représentés en base "unaire", avec des constructeurs `zero` et `succ`, voir sect. 3. Implantez surtout

- l'addition
- la soustraction ; notez que sur les nombres naturels, $n - m = 0$ pour $m \geq n$
- la multiplication comme itération de l'addition

Exercice 12 Définissez des opérations de comparaison sur les nombres naturels, par exemple les fonctions "strictement inférieur" et "égal". Les règles de réécriture ne sont pas typées, il n'existe donc pas de type Booléen. Pour l'imiter, vous pouvez utiliser des nombres (par exemple 0 pour faux, 1 pour vrai) ou encore mieux, des"constructeurs" (constantes) `true` et `false`.

Exercice 13 Définissez des fonctions Booléennes, par exemple la négation (`not`), le "et" et le "ou" et une fonction `if_then_else` ternaire (condition, branche *then* et branche *else*).

Si vous avez implanté différentes stratégies de réduction (sect. 4.3.3) : est-ce que `if_then_else` fonctionne également bien pour toutes les stratégies, ou est-ce qu'il y a des stratégies qui posent problème ? Pensez aux fonctions récursives comme par exemple la division euclidienne (exercice 14). Documentez vos observations dans le fichier `README`.

Exercice 14 Définissez la division euclidienne. Pour deux nombres naturels a et b, elle calcule un couple : le quotient q de la division de a par b ; et le reste r, tels que $a = qb + r$ et $r < b$. L'algorithme soustrait successivement b de a (en incrémentant q), jusqu'à ce que a soit inférieur à b. Bien entendu, la définition a besoin des fonctions définies précédemment.

Exercice 15 Définissez des fonctions sur des listes, par exemple les fonctions `map` et `filter`. En analogie avec des nombres naturels, vous pouvez utiliser des constructeurs `nil` (liste vide, `[]` en Caml) et `cs` (cons, `::` en Caml). Avec ceci, la liste qui s'écrit `[1; 2; 3]` en Caml est représentée comme `cs 1 (cs 2 (cs 3 nil))` dans votre système de réécriture.

Exercice 16 Vous avez maintenant (presque) toutes les fonctions pour définir le crible d'Erastothène, qui a été discuté en cours.

4.3.3 Stratégies (optionnel)

Les exercices de cette section sont optionnels. Lisez pourtant les explications du paragraphe suivant.

Explications Il peut y avoir plusieurs manières de réécrire un terme avec une règle, parce que la règle peut avoir plusieurs *redexes* (*reducible expressions*) dans le terme, c.-à-d., des positions où la règle est applicable. Nous illustrons la situation avec le système de règles de la fig. 1 et le terme `f (f a a) (f a a)`.

Voici quelques stratégies courantes, qui se réfèrent à des positions dans l'arbre syntaxique du terme à réécrire.

- *Rightmost-innermost* : la réécriture s'effectue à la position qui est le plus à l'intérieur et le plus à droite. Ceci revient à évaluer d'abord les arguments des fonctions avant de les appliquer ("call by value"). C'est la stratégie choisie par Caml.
  Dans notre exemple, le redex choisi est le `a` en gras : `f (f a a) (f a a)` qui est réécrit avec la règle `r2` : `f (f a a) (f a b)`. Deux autres pas de réécriture produisent `f (f a a) (f b b)` et `f (f a a) b`.
- *Leftmost-outermost* : la réécriture s'effectue à la position qui est le plus à l'extérieur et le plus à gauche. Ceci revient à réduire d'abord la fonction avant de calculer les arguments ("call by name"). C'est la stratégie choisie par des langages paresseuses comme Haskell (qui effectuent en plus des optimisations considérables).
  Dans notre exemple, le redex choisi est la racine du terme, et la réduction avec la règle `r1` donne `f a a`. Encore une réduction avec `r1` donne `a` et finalement `b` (avec `r2`).

Ces stratégies sont fixes dans le sens qu'elles parcourent le terme toujours de la même manière, sans essayer de repérer de manière dynamique le "meilleur" redex. Et pour cause – pour toute notion utile de "meilleur" redex, la question est indécidable.

Évidemment, il y a aussi d'autres combinaisons possibles (leftmost-innermost, ... ; parallèles qui réécrivent le sous-terme gauche et droit en même temps ...). La stratégie *leftmost-outermost* a l'avantage suivant : si un terme a une forme normale (pour un système de réécriture), alors cette stratégie la trouve, tandis que d'autres stratégies risquent de ne pas terminer.

A faire De manière plus abstraite, on peut dire qu'une stratégie est une fonction de type

```
term -> term -> term ->
rewr_result -> rewr_result -> rewr_result ->
rewr_result
```

qui prend les sous-termes gauche `l` et droit `r` et le terme à réécrire `t`, ainsi que les résultats de la réécriture de ces trois termes (de type `rewr_result`), et qui sélectionne ou construit le "bon" terme à renvoyer. Ainsi, la stratégie *leftmost-outermost* va privilégier le résultat de la réécriture du terme `t` (si ce terme a été modifié par le réécriture), sinon du sous-terme `l`, sinon de `r`.

Exercice 17 Le fichier `rewriting.ml` contient le code pour la stratégie *leftmost-outermost*. Dans ce style, implantez d'autres stratégies.

Exercice 18 Modifiez le code des fonctions `simp` et `rewrite_in_term` en rajoutant comme paramètre la stratégie de réécriture. Vous pouvez maintenant changer facilement de stratégie, en appelant la fonction `simp` avec différentes fonctions de stratégie.

Exercice 19 La modification qui vient d'être faite permet de changer de stratégie plus facilement. Le désavantage est que la réécriture devient plus lente : Pour chaque terme, on effectue d'abord la réécriture sur tous les sous-termes pour sélectionner ensuite le terme à retenir. Il est mieux d'effectuer cette réécriture de manière paresseuse (*lazy*). Par exemple, pour la stratégie *leftmost-outermost*, on appelle la réécriture sur le sous-terme gauche `l` uniquement si elle a échoué pour le terme complet `t`. Voyez-vous comment il

faut adapter la définition de stratégie ? Le type `strategy` (fichier `lang.ml`) donne une indication, et si vous avez des questions, n'hésitez pas à nous en parler.

A Fichiers du projet

A.1 Liste des fichiers fournis

On vous fournira un ensemble de fichiers dont il vous faudra compléter le code (fonctions marquées `TODO`). Ne rajoutez pas d'autres fichiers et ne modifiez ni le `Makefile` ni lexer et parser, sauf si vous savez exactement ce que vous faites. Vous pouvez mettre des exemples (systèmes de réécriture) dans le même répertoire que le code Caml, ou de préférence dans un sous-répertoire.

Liste des fichiers :

- `Makefile` : pour coordonner la compilation de tous les fichiers du projet. Les fichiers sont compilés sélectivement, selon leurs dépendances. Utilisation : dans une console, faire :
  - `make` (sans arguments) pour lancer la compilation de tous les fichiers et générer l'exécutable `eqprover`, qui peut ensuite être exécuté avec `./eqprover`
  - `make subst.cmo` (pareil pour les autres fichiers `.ml`) pour compiler le fichier `subst.ml` et toutes ses dépendances. Génère uniquement le fichier objet `subst.cmo`, qui peut ensuite être chargé (avec `#use`) dans une session interactive.
- `lexer.mll` et `parser.mly` : définissent les unités lexicales (constantes, variables, nombres etc.) respectivement la syntaxe de notre système de réécriture, voir sect. 3. *Ne pas modifier.*
- `lang.ml` : Les types de données utilisés dans le projet, voir sect. 2. *Ne pas modifier.*
- `interf.ml` : Interface avec le parser. *Ne pas modifier.*
- `subst.ml` : Les fonctions de substitution, voir sect. 4.2.1.
- `unif.ml` : Unification, voir sect. 4.2.2.
- `rewriting.ml` : Les fonctions de réécriture (sect. 4.2.3 et 4.3.1) et les stratégies (sect. 4.3.3)
- `use.ml` : La liste de tous les fichiers compilés, voir annexe A.2

A.2 Utilisation de l'ensemble des fichiers

Compilez d'abord l'ensemble des fichiers, en lançant `make` dans la console, dans le répertoire où se trouvent les fichiers `*.ml`.

- *Mode interactif :* Démarrez la session Caml et chargez le fichier `use.ml`, avec `#use "use.ml";;` (le `#` fait partie de la commande et n'est pas l'invite de Caml). C'est le mode préféré lors du développement, pour tester vos fonctions.
  En général, les fonctions sont préfixées avec le nom de leur module (le nom du fichier avec la première lettre en majuscule). Pour utiliser la fonction sans ce préfixe, il faut "ouvrir" le module. Par exemple, le fichier `interf.ml` contient une fonction `parse` que vous pouvez appeler avec un nom de ficher pour obtenir l'arbre syntaxique. Après avoir chargé le fichier compilé avec `#load "interf.cmo"` [1], vous pouvez utiliser le parser avec `Interf.parse "Examples/ex1.rew";;`. Ensuite,
  ```
  # open Interf ;;
  # open Lang ;;
  # parse "Examples/ex1.rew" ;;
  ```
  vous permet d'utiliser la fonction sans préfixe, et d'obtenir un arbre syntaxique plus lisible.
- *Mode compilé :* Après compilation du fichier `eqprover.ml` (avec `make`), l'analyse syntaxique est appelée avec le nom du fichier passé comme argument de l'exécutable. Vous pouvez donc lancer le prouveur de la console, comme indiqué en sect. 1.2. C'est le mode préféré pour travailler avec des systèmes de réécriture une fois que votre code est stable.

1. La commande `#use` est utilisée pour des fichiers non compilés (`*.ml`), la commande `#load` pour des fichiers compilés (`*.cmo`).

B Caml : quelques conseils

Voici quelques indications sur des éléments les plus utilisés. Pour des questions plus pointues, référez vous au manuel de Caml disponible sur Moodle.

B.1 Exceptions

Une exception arrête immédiatement l'évaluation d'une expression de Caml. Si une exception est levée par une fonction, aussi toutes les fonctions appelantes sont arrêtées, et le programme se termine avec un message d'erreur. Par contre, on peut aussi capter une expression et ainsi éviter sa propagation.

Dans sa forme la plus simple, on lève une exception avec `failwith`, suivi d'une chaîne de caractères qui permet d'expliquer la cause. Par exemple,

```
failwith ("clash")
```

peut être utilisé lors de l'unification pour signaler un clash.

Pour distinguer plusieurs sortes d'exceptions, on peut déclarer des exceptions comme des types, par exemple

```
exception UnifError of string
```

pour une exception levée lors de l'unification, qui a un argument (une chaîne de caractères). Une exception est levée avec `raise`, par exemple

```
raise (UnifError("clash"))
```

Pour capter une exception, on utilise `try ... with`. Pour illustrer le fonctionnement, considérons la fonction :

```
let unif_option t1 t2 =
  try Some (unif ([(t1, t2)], []))
  with UnifError(ue) -> None
```

Elle exécute le code suivant `try`. Si ce code ne lève pas d'exception, la fonction renvoie le résultat de l'évaluation ; dans ce cas, `try ... with` n'a aucun effet, et on renvoie `Some` résultat de l'unification. Si le code lève une exception, on cherche un *handler* approprié pour le type d'exception. Le comportement est analogue au filtrage avec `match ... with`. Dans notre exemple, si l'unification lève une `UnifError`, on renvoie `None`. Dans le cas d'une autre erreur (par exemple division par zéro), l'exception est propagée.

B.2 Mesurer le temps d'exécution

Pour savoir quelle stratégie est plus efficace, il peut être utile de mesurer le temps d'exécution d'une commande. Voici comment faire (sur un système Linux) :

```
let _ = Sys.command "date" in () ;
print_string (parse_and_simp "Examples/arith.rew") ;
let _ = Sys.command "date" in () ;;
```

B.3 Utilisation d'`ocamlyacc`

Pour ce projet, vous n'avez pas besoin de modifier le parser. Si la technologie vous tente, voici quelques explications.

But d'un parser Nous utilisons l'outil `ocamlyacc` en combinaison avec `ocamllex` pour générer un parser à partir d'une grammaire. Un parser est un programme qui prend en entrée un fichier texte et construit un arbre syntaxique, pourvu que le texte est conforme à la grammaire.

L'essentiel concernant la description de la grammaire dans le fichier `parser.mly` : Après quelques définitions préliminaires, la grammaire consiste en une liste de *productions* (règle de grammaire). Les productions peuvent être mutuellement dépendantes. Chaque production est composée d'une ou plusieurs clauses qui décrivent une manière de construire la catégorie syntaxique correspondante. Normalement, une clause est accompagnée d'une *action sémantique*, dans notre cas une expression de constructeur qui construit une partie de l'arbre syntaxique.

Très souvent dans notre fichier `parser.mly`, une production porte le même nom que le type qu'elle permet de construire. Ceci augmente la lisibilité, mais n'est nullement une nécessité. D'autres productions servent à analyser des listes d'éléments séparés de signes de ponctuation, tels que des virgules ou points-virgules. Ces signes disparaissent dans l'arbre syntaxique.

Par exemple, la production

```
rule:
  CONST COLON termListAsTerm ARROW termListAsTerm SEMICOLON
  {Rl($1, $3, $5)}
| termListAsTerm ARROW termListAsTerm SEMICOLON
  {Rl("", $1, $3)}
;
```

permet d'analyser des règles avec nom (première clause) ou sans nom (deuxième clause). Les variables `$1, $2` etc. se réfèrent au premier, deuxième etc. symbole de la clause respective. Ainsi, dans la première clause, `$1` se réfère au symbole attaché à `CONST`, et `$3` à la liste de termes `termListAsTerm`.

Génération d'un parser Après modification de `parser.mly`, on peut (re-)générer le parser associé avec `make`. En cas de problèmes, le fichier `parser.output` permet d'analyser la cause du problème. Les outils `ocamllex` et `ocamlyacc` sont documentés en détail dans le manuel de référence de Caml (voir lien sur Moodle).