SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

PCOMP(LU3IN032)
Année 2020/2021
Projet : Prolog

Le projet se trouve sur git https ://stl.algo-prog.info/, vous devez faire un fork, puis ajouter les enseignants comme maintainer : Carlos Agon, Emmanuel Chailloux, Basile Pesin, Antoine Miné, Christine Tasson.

Votre projet doit être *privé* et en binôme. Vous pouvez réaliser le projet dans le langage de votre choix.

Objectifs

1. Programmer un interprète du langage Prolog
   — unifier et substituer ;
   — résoudre le problème en parcourant l'arbre de preuve menant à une solution ou lever une exception ;
   — récupérer l'exception et revenir en arrière pour explorer une autre branche de résolution.
2. Écrire un fichier `RAPPORT.md` dans lequel vous déclarerez le langage choisi et vous présenterez votre projet pour chaque Jalon.
3. Rendre votre code lisible (choix des noms de fonction, structures de données, arbre de sources, commentaires, ...) en vue de la relecture par les enseignants et par vos camarades.
4. Relire et comparer l'implémentation du projet dans différents langages.

Organisation du code

Nous fournissons :

— Des fichiers d'exemple dans la syntaxe de Prolog `.pl`.
— Une structure d'AST (arbre syntaxique abstrait) en OCaml, Java et Python. La version OCaml utilise des variants (types sommes) ; les versions Java et Python utilisent des classes et le motif composite ; la version Java implante également le motif visiteur.
— Une version OCaml, Java et Python d'analyseur syntaxique (parseur, lexeur et fichier principal) permettant de charger un fichier `.pl` et de le convertir en AST et permettant de transformer une chaîne de caractères en AST.

Afin de bien organiser votre code,

— vous prendrez la peine d'isoler les fonctions utilitaires demandées dans des fichiers ou des classes ayant un nom explicite ;
— vous développerez des tests, que vous is isolerez dans des fichiers séparés facilement identifiables ;
— vous regrouperez les points d'entrées pour lancer les exemples dans un fichier ou une classe nommé `JalonX`.

Le rapport (rendu dans le fichier `RAPPORT.md`) sera rédigé au fur et à mesure. Vous y présenterez votre projet :

— Documentation du projet pour quelqu'un qui n'aurait pas lu le sujet.
— Structure de votre programme : arborescence, rôle de chaque fichier développé, ...

Pour chaque Jalon, vous détaillerez les points suivants :

— Comment exécuter votre Jalon.
— Structures de données choisies en justifiant leur pertinence du point de vue du style et de l'efficacité.
— Fonctions auxiliaires ajoutées : typage, utilité, efficacité, exemple d'utilisation, tests.
— Tests et exemples implémentés.
— Les choix importants et implémentations dont vous êtes fiers.
— Les difficultés rencontrées, comment elles ont été résolues ou pourquoi vous ne les avez pas (encore) résolues.

Barême - Dates des rendus

Barême indicatif

— Ponctualité : 2 pts
— Programmation : 10 pts répartis en
  — Jalon 1 (2pts),
  — Jalon 2 (2pts)
  — Jalon 3 (2pts)
  — Jalon 4 (2pts)
  — Jalon 5 (2pt)
— Rapport : 4 pts
— Relecture : 4 pts

Rendus sur git : Pour chaque Jalon, vous devez rendre l'ensemble des fichiers nécessaires à l'exécution de votre code et des tests correspondant. Vous devez également compléter le fichier `RAPPORT.md` avec la partie correspondante au jalon.

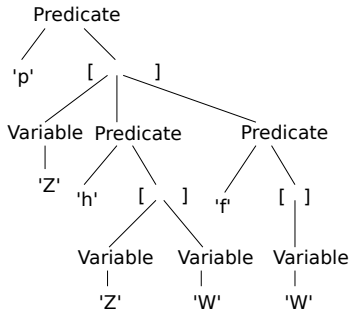
— Jalon 1 : 28 février 10h
— Jalon 2 : 7 mars 10h
— Jalon 3 : 14 mars 10h
— Jalon 4 : 28 mars 10h
— Jalon 5 : 4 avril 10h
— Accès aux projets à relire : 11 avril
— Rapport de relecture : 25 avril 10h,

Jalon 1 : Unification

Voici un exemple de terme

```
p(Z, h(Z, W), f(W))
```

et son arbre de syntaxe.



Le but de ce premier jalon est d'implémenter l'unification de termes décrit formellement par les règles suivantes :

$$\frac{\mathcal{P} \cup \{s \doteq s\}}{\mathcal{P}} \text{ (effacer)} \qquad \frac{\mathcal{P} \cup \{t \doteq x\} \;\; t \notin \mathcal{X}}{\mathcal{P} \cup \{x \doteq t\}} \text{ (orienter)}$$

$$\frac{\mathcal{P} \cup \{f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)\}}{\mathcal{P} \cup \{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}} \text{ (décomposer)} \qquad \frac{\mathcal{P} \cup \{x \doteq s\} \;\; x \in \mathrm{Var}(\mathcal{P}) \;\; x \notin \mathrm{Var}(s)}{\{x \to s\}(\mathcal{P}) \cup \{x \doteq s\}} \text{ (remplacer)}$$

La notation $\{x \to s\}(\mathcal{P})$ signifie substituer toutes les occurrences de x par s dans tous les termes de $\mathcal{P}$. La condition occur-check *$x \notin Var(s)$ signifie que l'on ne peut appliquer la règle remplacer que s'il n'y a pas d'occurrence de x dans s.*

Q.1.1 Les **équations** sont des paires de termes représentant une équation à résoudre. Les équations sont stockée dans une structure de données qui permet facilement de prendre une équation, d'ajouter des équations et de tester s'il reste des équations à traiter.

Choisir une structure adaptée pour représenter les **équations**, commenter votre choix en terme d'efficacité, tester et afficher un environnement

Q.1.2 Dans les règles de l'unification, le test d'occurrence **OccurCheck** doit être effectué avant tout remplacement et surtout avant d'ajouter une variable à l'environnement.

Écrire une fonction **OccurCheck** qui vérifie si une variable apparaît dans un terme. Commenter l'efficacité de votre fonction, et les choix que vous avez faits pour effectuer un nombre minimal de tests.

En OCaml, vous pourrez définir une fonction implantant un *pattern matching* avec `match with`. En Python, le *pattern matching* se fera plutôt avec `isinstance`. En Java, il sera plus agréable d'utiliser le motif visiteur (voir l'interface `TermVisitor` fournie).

Q.1.3 Les substitutions sont stockées sous la forme d'un **environnement** qui associe à une variable (identifiée par une chaîne de caractères) un terme. Par exemple, `env` est `{Z -> f(W); W -> Y}`.

Vous remarquerez que les variables à gauche peuvent apparaître dans les termes associés à droite. Celà ne pose pas de problème car le test d'occurence est vérifié avant l'ajout de chaque variable dans l'environnement.

Choisir une structure adaptée pour l'**environnement**, commenter votre choix en terme d'efficacité, tester et afficher un environnement.

Q.1.4 Écrire un algorithme `subst` qui, étant donnés un terme `t` et un environnement `env`, effectue les substitutions décrites par `env` de manière répétée jusqu'à ce que la substitution ne modifie plus le terme.

Par exemple, si `env` est `{Z -> f(W); W -> Y}` et `t` est `p(Z, h(Z, W), f(W))`, alors
`subst` va produire le terme `p(f(Y), h(f(Y), Y), f(Y))`.

Notez bien la répétition des substitutions : les occurrences de `Z` sont remplacées par `f(W)`, qui sont ensuite remplacées par `f(Y)`.

Q.1.5 L'algorithme d'**unification** que vous devez implanter prend en argument un **environnement** et des **équations**. Il commence par prendre les deux termes d'une équation et lui applique la substitution représentée dans l'environnement afin de tenir compte des substitutions déjà enregistrées dans l'environnement. Les deux termes obtenus sont alors décomposés selon leur structure ou, si l'un des deux est une variable et le test d'occurrence est vérifié, alors une nouvelle substitution est stockée dans l'environnement.

L'algorithme d'unification sera naturellement une fonction récursive avec *pattern-matching*. En Java, il est difficile d'étendre le motif visiteur aux fonctions à deux arguments ; nous suggérons donc d'utiliser également une fonction récursive qui implémente un *pattern-matching* ; elle utilisera `instanceof` pour tester le type des arguments.

- Écrire un algorithme `unify` qui résout un système d'équations dans un environnement donné et renvoie l'environnement permettant de résoudre le système.
- L'algorithme devra lever une exception que vous aurez définie lorsqu'une unification n'est pas possible. Vous vous en servirez pour afficher un message d'erreur explicite (pourquoi l'unification a échoué, sur quel sous-terme, dans quel environnement, ...)
- Tester l'algorithme sur les équations du TD sur l'unification.

Jalon 2 : Résolution et faits

On suppose ici que le programme ne contient que des assertions qui sont des faits dont tous les symboles de prédicats sont distincts 2 à 2. Vous allez implémentez trois interprètes qui traitent de plus en plus de programmes prolog. Pour chaque interprète, vous fournirez des programmes dans des fichiers `.pl`, que vous transformerez en AST, puis vous les interpréterez et vous afficherez sur la sortie standard l'AST et la solution. Vous ajouterez ces tests au fichier exécutable `Jalon1`.

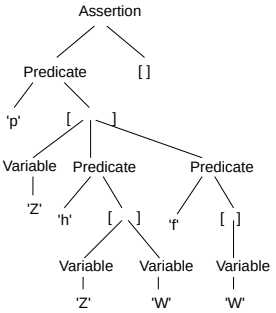
Voici un exemple de programme en `Prolog` avec un fait et un but :

```
p(Z, h(Z, W), f(W)).
?- p(f(X), h(Y, f(a)), Y).
```

Ce programme Prolog est une liste contenant :

- un *fait* (une règle `Assertion` qui ne dépend d'aucun autre terme) donné par le terme : `p(Z, h(Z, W), f(W))`
- un (seul) *but* (`Goal`) : `p(f(X), h(Y, f(a)), Y).`

Et l'AST du premier fait :



Q.2.1 Écrire un algorithme (méthode ou fonction) `interprete0` qui prend en argument l'AST d'un programme constitué d'un *fait* et d'un (seul) *but* et qui renvoie l'environnement permettant de résoudre le programme en unifiant les deux termes.

Tester votre algorithme à l'aide d'au moins deux exemples.

On considère à présent les programmes qui contiennent un seul *but* et plusieurs *faits*, mais au plus un *fait* par symbole de prédicat, comme dans l'exemple ci-contre où les symboles des prédicats des faits (`r`, `q`, `p`) sont biens distincts.

```
r(X, Y, h(Z)).
q(Z).
p(Z, h(Z, W), f(W)).
?- p(f(X), h(Y, f(a)), Y).
```

Q.2.2 Écrire un algorithme `interprete1` qui prend en argument l'AST d'un programme constitué de plusieurs *faits* (mais un seul par symbole de prédicat) et d'un *but*, qui choisit le fait correspondant au symbole de prédicat du but, et qui renvoie l'environnement permettant de résoudre le programme en unifiant les deux termes.

Tester votre algorithme à l'aide d'au moins trois exemples.

On considère à présent les programmes qui contiennent un seul *but* et plusieurs *faits*, mais au plus un *fait* par symbole de prédicat, comme dans l'exemple ci-contre par exemple : où les symboles des prédicats des faits (`r`, `q`, `p`) sont biens distincts.

```
r(X, Y, h(Z)).
q(Z).
p(Z, h(Z, W), f(W)).
?- p(f(X),h(Y,f(a)),Y), q(f(W)), r(Y,Y,h(Y))
```

Q.2.3 Écrire un algorithme `interprete2` qui prend en argument l'AST d'un programme constitué de plusieurs *faits* (un seul par symbole de prédicat) et de plusieurs *buts* et qui renvoie l'environnement permettant de résoudre tous les buts.

Tester votre algorithme à l'aide d'au moins trois exemples.

Jalon 3 : Résolutions et règles

Nous ne nous limitons plus aux seuls faits : les règles peuvent avoir un corps non-vide (c'est le cas de la règle pour `p`), ce qui va nous amener à ajouter dynamiquement des buts lors de la résolution.

Nous ne considérons toujours que les programmes qui ont, pour chaque symbole de prédicat, une seule règle le définissant, comme dans l'exemple ci-contre :

```
r(a, a).
q(X).
p(Z, h(Z, W), f(W)) :- r(X, XX), q(XX).
?- p(f(X), h(Y, a), Y), q(a).
```

Étant donné un but extrait de la liste des buts à résoudre, l'algorithme va choisir (l'unique) règle `head :- body` dont la tête `head` peut être unifiée avec le but. Puis, il va ajouter tous les termes du corps `body` à la liste des buts à résoudre et recommencer dans le nouvel environnement généré par l'unification. Ce processus s'arrête quand la liste des buts est vide.

Les règles sont vraies quelle que soit la valeur de leurs variables. Dans l'exemple ci-dessous, la variable `X` dans la règle 1 et dans le but 2 n'ont aucune raison d'être identiques et on doit renommer les variables de la règle.

```
1  p(Z, h(Z, W), f(W)) :- r(X, XX), q(XX).
2  ?- p(f(X), h(Y, a), Y), q(a).
```

```
1  p(Z1, h(Z1, W1), f(W1)) :- r(X1, XX1), q(XX1).
2  ?- p(f(X), h(Y, a), Y), q(a).
```

```
q(a,b)
r(Y,X) :- q(X,Y).
?- r(X,Y).
```

Ce renommage ne change pas la signification de la règle 1, mais il est parfois nécessaire pour éviter la capture de variable comme dans l'exemple ci-contre. Sans renommage, l'algorithme échoue, alors qu'avec renommage des règles, on trouve `X -> Y1, Y -> X1, X_1->a, Y_1->b`.

Q.3.1 Écrire un algorithme `rename` qui, étant donné un compteur `n` et une règle `Assertion (head, body)` va produire une nouvelle règle `Assertion (head', body')` dont toutes les variables ont été renommées en y ajoutant en suffixe la valeur du compteur `n`. Vous pourrez utiliser un visiteur en Java, et des fonctions récursives en OCaml et Python.

Q.3.2 Écrire un algorithme `choose` qui, étant donné un compteur `n`, un environnement `env`, un but `goal` et une suite de règles `rules` va parcourir la suite de règles jusqu'à trouver la règle `Assertion (head, body)` dont `head` peut être unifiée avec le but `goal`. La règle est ensuite renommée en `Assertion (head', body')` grâce au compteur `n`. Le but est unifié avec la tête `head'` pour produire un environnement. L'algorithme retourne ce nouvel environnement, ainsi que la suite de termes `body'` qu'il faudra maintenant prouver. Si aucune règle ne permet de faire l'unification, une exception est levée.

Pour éviter la capture de variables, il suffira d'appeler `choose` avec des valeurs croissantes du compteur `n`, ce qui générera des noms de variable uniques à chaque application de règle.

Dans l'exemple ci-dessous, `choose` produira avec le compteur `1` et l'environnement vide :

```
1  p(Z, h(Z, W), f(W)) :- r(X, XX), q(XX).
2  ?- p(f(X), h(Y, a), Y), q(a).
```

— `[r(X1, XX1); q(XX1)]` et
— `{Z1->f(X); X->a; Y->f(X); W1->a}`.

Q.3.3 Écrire un algorithme `solve` qui, étant données une suite initiale de buts `goals` et une suite de règles `rules`, part d'un environnement initialement vide et prouve successivement chaque but grâce à `choose`. À chaque étape, l'environnement et la liste de buts sont mis à jour. Cet algorithme s'arrête quand il n'y a plus aucun but à prouver et produit, en sortie, un environnement permettant de vérifier tous les buts de `goals`. Si aucun environnement existe, une exception est signalée.

Q.3.4 Écrire un algorithme `interprete3` qui prend en argument un programme Prolog qui contient un ou plusieurs buts et une seule règle par symbole de prédicat, et qui renvoie une solution ou lève une exception.

Testez votre algorithme à l'aide de trois nouveaux exemples. Vous fournirez des programmes stockés dans des fichiers `.pl`, que vous transformerez en AST, puis vous les interpréterez et enfin, vous afficherez sur la sortie standard l'AST et la solution. Fournissez dans un fichier `Jalon3` un programme exécutable qui lance ces tests.

Jalon 4 : Backtracking

```
p(X):-q(X).
p(X):-r(X).
q(a).
r(b).
?- p(b), p(a).
```

Voici un programme Prolog qui contient plusieurs règles définissant le prédicat de symbole `p`. Ici, Prolog va essayer successivement les différentes règles correspondant au but actuel. Pour chaque choix, une solution sera cherchée. En cas d'échec, Prolog essaiera la règle suivante dans la liste des règles pouvant s'unifier avec le but actuel.

Le fait de revenir au point de choix précédent s'appelle le *backtracking*. Afin de l'implanter, on a besoin de mémoriser le *contexte courant* au moment du choix. Ce contexte de choix est un triplet constitué de :

— la liste des buts à résoudre,
— la suite des règles à explorer,
— l'environnement.

Q.4.1 Créer la structure de données (type ou classe) nécessaire pour encoder un choix et des fonctions ou méthodes qui vous permettront d'afficher sur la sortie standard un choix et une suite de choix.

Vous prendrez garde au problème du partage et de la mutabilité des structures de données. Il pourra être nécessaire (en fonction du langage et des choix d'implantation) de faire des copies de certaines informations mutables pour éviter qu'un contexte soit corrompu lors de la résolution : notre but en stockant des informations dans un contexte est de pouvoir les retrouver intactes lors du backtracking.

Lors de l'exploration des solutions, un journal de tous les choix possibles qui n'ont pas été explorés est tenu à jour. Il est utilisé pour pouvoir revenir en arrière.

Q.4.2 Écrire un algorithme `solve` qui prend en argument le journal des choix qu'il faudra explorer, la suite `goals` des buts à résoudre, la suite des règles `rules` et l'environnement `env`. Cet algorithme :

— résout le premier but dans `goals` en utilisant les règles dans `rules` et l'environnement `env` ;
— s'il existe une solution, met à jour l'environnement et recommence avec le prochain but dans la suite `goals` ;
— sinon, revient en arrière pour explorer le choix suivant dans la suite des contextes `ch` ;
— lorsque tous les buts dans `goals` ont été résolus, il renvoie la solution obtenue ;
— s'il ne reste plus de choix à essayer, il renvoie une erreur signalant que le problème est non-satisfiable.

Q.4.3 Écrire un algorithme `interprete4` qui prend l'AST d'un programme Prolog et qui renvoie une solution.

Tester votre algorithme à l'aide des programmes contenus dans le dossier exemple. Vous transformerez les programmes stockés dans les fichiers `.pl` en AST, puis interpréterez et enfin, vous afficherez sur la sortie standard l'AST et la solution. Tester également votre algorithme à l'aide des programmes que vous avez écrits dans l'initiation à Prolog.

Vous fournirez un fichier exécutable `Jalon4` qui lance tous ces tests.

Jalon 5 : Toplevel Et (Toutes les solutions Ou Trace)

Q.5.1 Programmer un **toplevel** permettant de charger des règles à l'aide d'un fichier, écrire une requête, afficher une solution.

Q.5.2 Programmer un interprète capable de retourner **toutes les solutions** ; l'interprète s'interrompt après chaque solution et demande à l'utilisateur s'il souhaite continuer avec les solutions suivantes ou s'arrêter ;

Q.5.3 Afficher le parcours de l'arbre de recherche menant à une solution, en indiquant les positions des `buts` et `regles` utilisés. Cet algorithme correspond à la fonction **trace** de prolog.

Relecture

Vous devrez rendre votre rapport de relecture sur Moodle dans un fichier au format `pdf`. Nous allons choisir trois projets complets que nous rendrons accessibles. Vous pourrez les exécuter sur votre ordinateur ou sur les machines de la PPTI (des instructions d'installation seront fournies dans les fichiers `README.md` de chaque sous-répertoire « langage » : Java, OCaml et Python).

Objectifs

- Lire un programme dans des langages différents
  - lire et comprendre un code
  - critiquer de façon constructive l'implémentation d'un programme
- Comparer l'implémentation d'un même projet dans différents langages
  - comparer les structures de données
  - comparer l'implémentation des algorithmes et leur efficacité

Voici un plan pour votre rapport :

Présentation

- Nom, Prénom, Numéro d'étudiant.
- Langage utilisé pour l'implémentation de votre projet.
- Langage choisi pour la relecture (différent de celui choisi pour l'implémentation de votre projet).

Relecture (2 points) Après avoir lu le projet dans un autre langage que celui utilisé pour votre propre implémentation, vous devez :

(1 point) proposer des tests pour les quatre Jalons du projet avec des programmes Prolog que vous aurez écrits, permettant de mettre en évidence les spécificités de chaque Jalon ; présenter les résultats de ces tests et commenter vos résultats ;

(1 point) pour chacun des quatre jalons, proposer des améliorations du code pour :

- la *lisibilité* : quels éléments permettent de comprendre les algorithmes : commentaires, fonctions intermédiaires ?
- l'*élégance* : est-ce que le code contient des redondances ou au contraire, les fonctionnalités communes sont factorisées, l'arborescence du code est logique ? Est-ce que la structure du code et les structures de données choisies ont facilité développement des extensions au Jalon 5.
- l'*efficacité* : en quoi le choix des structures de données permet d'optimiser la complexité en espace et en temps ?

Comparaison (2 points) Vous devrez ensuite comparer l'implémentation du projet dans deux langages différents.

Vous décrirez quelles sont les *différences*, les *avantages* et *inconvénients* des implémentations pour les deux points suivants :

(1 point) les structures de données (une parmi l'AST, l'environnement ou les contextes de choix) ;

(1 point) les algorithmes (un parmi l'unification, la résolution avec ou sans backtracking).