Projet

Un interpréteur pour les automates à pile déterministes

Version du 1 avril 2022

Les *automates à pile* sont un mécanisme reconnaisseur pour les grammaires, comme les automates finis sont un mécanisme reconnaisseur pour les expressions rationnelles. Ces automates (qui ont général sont non déterministes), et leur relation avec les grammaires, seront présentés en cours. Pour ce projet, nous nous intéressons seulement à l'exécution des automates à pile déterministes.

Un *automate à pile* est donné par

1. un ensemble fini Σ de symboles d'entrée ;
2. un ensemble fini Q de symboles d'état ;
3. un état initial $q_0 \in Q$;
4. un ensemble fini Γ de symboles de pile ;
5. un symbole initial de pile $Z_0 \in \Gamma$;
6. un ensemble de transitions, où une transition est un élément de

$$(Q \times \Sigma \cup \{\epsilon\} \times Z \times Q \times Z^*)$$

Une *configuration* est de la forme

$$(q, \gamma, w) \in Q \times \Gamma^* \times \Sigma^*$$

où q est l'état actuel, γ la pile actuelle, et w la partie de l'entrée qui reste à lire. La configuration initiale pour un mot d'entrée $w \in \Sigma^*$ est (q_0, Z_0, w), et les configurations acceptantes sont toutes les configurations de la forme (q, ϵ, ϵ). Il y a deux possibilités pour passer d'une configuration à une autre :

— De $(q_1, \gamma X, aw)$ on peut passer à $(q_2, \gamma\alpha, w)$ quand il y a une transition (q_1, a, X, q_2, α)
— De $(q_1, \gamma X, w)$ on peut passer à $(q_2, \gamma\alpha, w)$ quand il y a une transition $(q_1, \epsilon, X, q_2, \alpha)$

Des exemples seront donnés pendant le cours.

Un automate à pile est déterministe quand de toute configuration on peut passer à une seule autre configuration. Contrairement aux automates finis du L2, les automates à pile déterministes sont moins puissants que les automates à pile non déterministes.

Pour ce projet nous proposons plusieurs syntaxes concrets pour les automates à pile. Votre tâche sera de réaliser l'analyse lexicale et grammaticale et transformer des descriptions d'automates à pile en syntaxe abstraite, de vérifier que l'automate est déterministe, et puis d'exécuter l'automate sur un mot donné par l'utilisateur.

1 Première étape

La syntaxe de la première étape est une traduction directe de la définition des automates à pile. Un exemple d'un automate dans cette syntaxe est

```
input symbols: a, b, c
stack symbols: A, B, C, Z
states: 1, 2
initial state: 1
intial stack symbol: Z

transitions:

(1,a,Z,1,Z;A)
(1,b,Z,1,Z;B)
(1,c,Z,2,Z)

(1,a,A,1,A;A)
(1,b,A,1,A;B)
(1,c,A,2,A)

(1,a,B,1,B;A)
(1,b,B,1,B;B)
(1,c,B,2,B)

(2,a,A,2,)
(2,b,B,2,)

(2,,Z,2,)
```

Cet automate reconnaît l'ensemble $\{wc\overline{w} \mid w \in \{a,b\}^*\}$, c'est donc un cas particulier des palindromes.

On peut décrire la grammaire avec les productions suivantes. Pourtant nous vos conseillons d'utiliser pour la réalisation de votre projet les raccourcis de la bibliothèque standard de menhir. Les non-terminaux sont en gras et en italiques, les terminaux (jetons) en police `code` :

- ***automate*** → ***declarations transitions***
- ***declarations*** → ***inputsymbols stacksymbols states initialstate initialstack***
- ***inputsymbols*** → `input symbols:` ***suitelettres-nonvide***
- ***stacksymbols*** → `stack symbols:` ***suitelettres-nonvide***
- ***states*** → `states:` ***suitelettres-nonvide***
- ***initialstate*** → `initial state:` `lettre`
- ***initialstack*** → `initial stack symbol:` `lettre`
- ***suitelettres-nonvide*** → `lettre` | `lettre ,` ***suitelettres-nonvide***
- ***transitions*** → `transitions:` ***translist***
- ***translist*** → ϵ | ***transition translist***
- ***transition*** → `(` `lettre ,` ***lettre-ou-vide*** `, lettre , lettre ,` ***stack*** `)`
- ***lettre-ou-vide*** → ϵ | `lettre`
- ***stack*** → ϵ | ***nonemptystack***
- ***nonemptystack*** → `lettre` | `lettre ;` ***nonemptystack***

avec la définition de jetons :

- `lettre` → `[0-9a-zA-Z]`

Il est conseillé d'écrire le programme en plusieurs étapes de difficulté croissante. Dans un premier temps, écrivez un analyseur lexical et syntaxique qui vérifie uniquement la bonne syntaxe de l'entrée mais ne produit rien. Ajouter ensuite la construction de l'arbre de syntaxe abstraite (dont le type doit être défini en OCaml) d'un programme décrit par la grammaire

ci-dessus. Ensuite, écrivez un interpréteur pour l'arbre construit qui lit du terminal un mot, exécute l'automate sur le mot et affiche le détail de l'exécution.

Quand vous cherchez une transition à appliquer à une configuration vous pouvez simplement parcourir la liste des transitions du haut vers bas, et prendre la première transition qui s'applique.

Gestion des erreurs Votre programme doit traiter proprement toutes les erreurs qui peuvent se produire pendant l'interprétation, c'est-à-dire afficher un message explicite, détaillant l'erreur rencontrée, et puis terminer l'interpréteur. Les erreurs à capturer sont, en plus des erreurs de l'analyse syntaxique :

— il n'y a aucune transition qui s'applique ;
— l'entrée est épuisée sans que la pile soit vide ;
— la pile est vide sans que l'entrée soit épuisée.

2 Deuxième étape

Ajoutez des vérifications de la bonne formation de l'automate :

— l'état initial doit être un élément de l'ensemble des états ;
— le symbole de pile initial doit être dans l'ensemble des symboles de pile ;
— l'automate doit être déterministe (attention aux transitions ϵ !)

3 Troisième Étape

Cette étape consiste à implémenter une syntaxe des automates à pile qui est plus proche des constructions des langages de programmation. La partie des déclaration reste la même, mais la partie des transitions est remplacée par un programme. Un exemple est

```
input symbols: a, b, c
stack symbols: A, B, C, Z
states: 1, 2
initial state: 1
intial stack symbol: Z

program:
  case state of
    1: begin
       case next of
         a: push A
         b: push B
         c: change 2
       end
    2: begin
       case top of
         A: begin case next of a: pop end
         B: begin case next of b: pop end
         Z: pop
       end
```

Les instructions d'un tel programme sont

— `pop`, qui supprime l'élément au sommet de la pile ;
— `push` x, qui ajoute le symbole x au sommet de la pile ;
— `reject`, qui arrête l'exécution avec un message de refus ;
— `change` x, qui change l'état en x ;

— une distinction de cas soit sur l'état actuel (`state`), soit sur le symbole au sommet de la pile (`top`), soit sur le symbole suivant de l'entrée (`next`). Une utilisation de `next` implique que le symbole suivant de l'entrée est consommé.

Il est maintenant à vous d'écrire la grammaire qui correspond à la syntaxe des fichiers dans `exemples_etape3`. Faites en sorte qu'elle soit $LR(1)$, et sinon gérez les priorités comme vu en cours.

Indication : la priorité d'une règle de la grammaire est par défaut la priorité de son dernier jeton, mais on peut aussi directement donner la priorité d'une règle avec l'étiquette `%prec`. Par exemple, dans

```
liste:
| e=element l=liste { e::l }
|                   { [] } %prec UNJETON
```

la priorité de la règle liste $\rightarrow \epsilon$ est celle du jeton `UNJETON`.

4 Rendu

Le langage de programmation utilisé doit être OCaml, l'analyse lexicale doit être réalisée avec *ocamllex*, et l'analyse grammaticale avec *menhir*. Nous vous conseillons de vous servir des raccourcis de la bibliothèque standard de menhir.

Le projet est à faire en binôme, les deux membres d'un binôme peuvent être dans des groupes de TP différents. Les monômes sont autorisés mais seront notés comme les binômes.

Le travail doit être réalisé tout au long de la suite du semestre via un dépôt git pour chaque groupe (binôme ou monôme). L'usage de ce gestionnaire de version permet d'éviter toute perte de donnée, et d'accéder si besoin à vos anciennes versions.

— Votre dépôt git doit être hébergé par le serveur GitLab de l'UFR d'informatique : `https://gaufre.informatique.univ-paris-diderot.fr`

— Votre dépôt doit être rendu privé dès sa création, avec accès uniquement aux membres du groupe et aux enseignants de ce cours. Tout code laissé en accès libre sur gaufre ou ailleurs sera considéré comme une incitation à la fraude, et sanctionné.

— Il va de soi que votre travail doit être strictement personnel : aucune communication de code ou d'"idées" entre les groupes, aucune "aide" externe ou entre groupes. Nous vous rappelons que la fraude à un projet est aussi une fraude à un examen, passible de sanctions disciplinaires pouvant aller jusqu'à l'exclusion définitive de toute université.

5 Rapport et Documentation

Il est nécessaire que vous indiquiez dans le fichiez `LisezMoi.txt` les commandes qu'il faut lancer pour compiler et exécuter votre programme. Vous êtes libres de vous servir de `dune`, `ocamlbuild`, ou d'écrire des Makefile à la main. De plus, il vous est demandé de brièvement expliquer (dans le fichier `LisezMoi.txt` ou dans un fichier séparé) quelles sont exactement les parties du sujet que vous avez réalisées, et dans le cas où vous travaillez en binôme comment vous avez organisé le travail dans le binôme.

La date de rendu, ainsi que la date de la soutenance, seront annoncées plus tard (les deux seront sans doute après les vacances de printemps).

Annexe : Usage de GitLab

La création d'un depôt git sur le GitLab de l'UFR permettra à votre groupe de disposer d'un dépôt commun de fichiers sur ce serveur. Chaque membre du groupe pourra ensuite disposer d'une copie locale de ces fichiers sur sa machine, les faire évoluer, puis sauvegarder les changements jugées intéressants et les synchroniser sur le serveur. Si vous ne vous êtes pas déjà servi de GitLab dans d'autres matières, cette section décrit son usage le plus élémentaire. Pour plus d'informations sur git et GitLab, il existe de multiples tutoriels en ligne. Contactez-nous en cas de problèmes.

Accès au serveur et configuration personnelle. Connectez-vous via l'interface web : `https://gaufre.informatique.univ-paris-diderot.fr`. Utilisez pour cela les mêmes nom et mots de passe que sur les machines de l'UFR, et pas vos comptes "ENT" de paris diderot ou u-paris. Cliquez ensuite sur l'icône en haut à droite, puis sur "Settings". A droite, allez ensuite dans la section "SSH Keys", et ajoutez ici la partie public de votre clé ssh (ou de vos clés si vous en avez plusieurs). Cela permettra facilitera grandement l'accès ultérieur à votre dépôt git, et vous évitera de taper votre mot de passe à chaque action.

Si vous n'avez pas encore de clé ssh, générez en une sur votre machine. L'usage de ssh n'est pas spécifique à git et GitLab, et permet des connections "shell" à des machines distantes. Si vous n'utilisez pas encore ssh et les clés publiques/privées ssh, il est temps de s'y mettre ! Pour plus d'information sur ssh, consultez :

`http://www.informatique.univ-paris-diderot.fr/wiki/doku.php/wiki/howto_connect`

Création du dépôt. Pour ce projet, nous vous fournissons quelques fichiers de tests. Votre dépôt git sera donc un dérivé (ou "fork") du dépôt publique du cours. En pratique :

- L'un des membres de votre groupe se rend sur la page du cours à partir du X Avril : `https://gaufre.informatique.univ-paris-diderot.fr/bauer/gasp2/` s'identifie si ce n'est pas déjà fait, et appuie sur le bouton "fork" (vers le haut, entre "Star" et "Clone"). Attention, un seul "fork" par groupe suffit.
- Ensuite, allez dans la section "Settings" en bas à gauche, défilez un peu et cliquez sur "Visibility", et selectionnez "Private" comme "projet visibility", puis "Save changes" un peu plus bas. Vérifiez qu'un cadenas apparaît maintenant à côté de `gasp` quand vous cliquez sur "Projet" en haut à gauche.
- Toujours dans "Settings" en bas à gauche, mais sous-section "Members" maintenant. "Invitez" votre collègue de projet, ainsi que les identifiants `habermeh`, `bauer`, `fagnot`, `picantin` en choisissant "Maintainer" comme rôle.
- Voilà, votre dépôt sur le GitLab est prêt !

Création et synchronisation de vos copies locales de travail. Chaque membre du projet "clone" le dépôt du projet sur sa propre machine, c'est-à-dire en télécharge une copie locale : `git clone` suivi de l'addresse du projet tel qu'il apparaît dans l'onglet "Clone" sur la page du projet, champ en "SSH". Pour cela, il faut avoir installé `git` et `ssh` et configuré au moins une clé ssh dans GitLab.

Une fois le dépôt créé et cloné et en se plaçant dans le répertoire du dépôt, chaque membre pourra à tout moment :

- télécharger en local la version la plus récente du dépôt distant sur Gitlab : `git pull`.

— téléverser sa copie locale modifiée sur GitLab :
`git push`.

Avant toute synchronisation, il est demandé d'avoir une copie locale "propre" (où toutes les modifications sont enregistrées dans des "commits").

Modifications du dépôt : les commits. Un dépôt Git est un répertoire dont on peut sauvegarder l'historique des modifications. Chaque action de sauvegarde est appelée une révision ou "commit". L'index du dépôt est l'ensemble des modifications qui seront sauvegardées à la prochaine révision. La commande

`git add` suivi du nom d'un ou plusieurs fichiers

permet d'ajouter à l'index toutes les modifications faites sur ces fichiers. Si l'un d'eux vient d'être créé, on ajoute dans ce cas à l'index l'opération d'ajout de ce fichier au dépôt. La même commande suivie d'un nom de répertoire ajoute à l'index l'opération d'ajout du répertoire et de son contenu au dépôt. La révision effective du dépôt se fait par la commande

`git commit -m` suivi d'un message entre guillemets doubles.

Invocable à tout instant, la commande

`git status`

permet d'afficher l'état courant du dépôt depuis sa dernière révision : quels fichiers ont été modifiés, renommés, effacés, créés, etc., et lesquelles de ces modifications sont dans l'index. Elle indique également comment rétablir l'état d'un fichier à celui de la dernière révision, ce qui est utile en cas de fausse manœuvre.

Les commandes `git mv` et `git rm` se comportent comme `mv` et `rm`, mais ajoutent immédiatement les modifications associées du répertoire à l'index.

Il est conseillé d'installer et d'utiliser les interfaces graphiques `gitk` (visualisation de l'arbre des commits) et `git gui` (aide à la création de commits).

Une dernière chose : git est là pour vous aider à organiser et archiver vos divers fichiers sources. Par contre il vaut mieux ne *pas* y enregistrer les fichiers issues de compilations (binaires, répertoire temporaire tels que `_build` pour `dune`, fichiers objets OCaml `*.cm{o,x,a}`, etc).

Les fusions (merge) et les conflits. Si vous êtes plusieurs à modifier vos dépôts locaux chacun de votre côté, celui qui se synchronisera en second avec votre dépôt GitLab commun aura une manoeuvre nommé "merge" à effectuer. Tant que vos modifications respectives concernent des fichiers ou des zones de code différentes, ce "merge" est aisé, il suffit d'accepter ce que git propose, en personnalisant éventuellement le message de merge. Si par contre les modifications se chevauchent et sont incompatibles, il y a alors un conflit, et git vous demande d'aller décider quelle version est à garder. Divers outils peuvent aider lors de cette opération, mais au plus basique il s'agit d'aller éditer les zones entre `<<<<<` et `>>>>>` puis faire `git add` et `git commit` de nouveau.

Intégrer les modifications venant du dépôt du cours. Si le dépôt du cours reçoit ultérieurement des correctifs ou des évolution des fichiers fournis pour le projet, ces modifications peuvent être intégrés à vos dépôts.

— La première fois, allez dans votre répertoire de travail sur votre machine, et tapez :
`git remote add prof \`
`git@gaufre.informatique.univ-paris-diderot.fr:bauer/gasp.git`
— Ensuite, à chaque fois que vous souhaiterez récupérer des commits du dépôt du cours :
`git pull prof master`
— Selon les modifications récupérées et les votres entre-temps, cela peut occasionner une opération de "merge" comme décrite auparavant.
— Enfin, ces modifications sont maintenant intégrés à votre copie locale de travail, il ne reste plus qu'à les transmettre également à votre dépôt sur GitLab :
`git push`

Les branches. Il est parfois pratique de pouvoir essayer différentes choses, même incompatibles. Pour cela, Git permet de travailler sur plusieurs exemplaires d'un même dépôt, des branches. Un dépôt contient toujours une branche principale, la branche "master", dont le rôle est en principe de contenir sa dernière version stable. Les autres branches peuvent servir à développer des variantes de la branche master, par exemple pour tenter de corriger un bug sans altérer cette version de référence. La création d'une nouvelle branche, copie conforme de la branche courante – initialement, master – dans son état courant, se fait par :

`git branch` suivi du nom choisi pour la branche.

Sans argument, cette commande indique la liste des branches existantes, ainsi que celle dans laquelle se trouve l'utilisateur. Le passage à une branche se fait par

`git checkout` suivi du nom de la branche.

Pour ajouter au dépôt distant une branche qui n'est pas encore sur celui-ci, après s'être placé dans la branche :

`git push -set-upstream origin` suivi du nom de la branche

Un `push` depuis une branche déjà sur le serveur se fait de la manière habituelle. Enfin, on peut "réunifier" deux branches avec `git merge`, voir la documentation pour plus de détails.

Noter que GitLab propose également un mécanisme de "Merge Request" : il permet de proposer des modifications, soit à son propre projet, soit au projet qui a été "forké" à l'origine, les membres du projet en question pouvant alors accepter ou non ces suggestions après discussion.