Projet d'Algorithmes pour la compilation

Devan SOHIER

2021-2022

Le but de ce projet est d'écrire en C un analyseur syntaxique, prenant en entrée un fichier décrivant un analyseur LR et une chaîne de caractère, et exécutant le premier sur la deuxième.

Ce projet manipule des textes au format ASCII, et son comportement face à des encodages différents est non-défini (une erreurs système n'est bien évidemment pas souhaitable, mais le résultat de l'analyse, qu'elle réussisse ou échoue, n'est pas interprétable). La grammaire, la table et le texte analysés doivent être à ce format (donc pas de caractères accentués, pas d'encodages Unicode, etc.)

1 Grammaires et tables

Une règle de grammaire est représentée par une structure contenant deux champs :
- un caractère, symbole représentant le non-terminal membre gauche de cette règle ;
- une chaîne de caractères C représentant la production de cette règle, avec pour convention que les non-terminaux sont représentés par le symbole correspondant affecté d'un bit de poids forts à 1 (les bits de poids fort des codes ASCII sont tous à 0, laissant octets avec un bit de poids fort à 1 représenter les diverses extensions du code, comme les ISO latin par exemple ; il n'y a donc pas de risque de confusion) ; dans une interprétation numérique du type de données `signed char`, cela revient à dire que les non-terminaux sont représentés par des codes négatifs.

Une grammaire est représentée par une structure de données comprenant trois champs :
- un caractère : le symbole de l'axiome de la grammaire ;
- un entier représentant le nombre de règles de la grammaire ;
- un tableau de règles de la grammaire.

L'utilisation de ces différents champs est explicité par la fonction `print_grammar` fournie. Exécutez la, et lisez la attentivement.

Une table d'analyse est représentée par une structure contenant deux champs :
- un entier donnant le nombre de lignes de la table ;
- un tableau d'entiers (des `short`) à deux dimensions, linéarisé, contenant 256 colonnes (une pour chaque caractère ASCII avec un bit de poids fort à 0, terminal de la grammaire pouvant être présent dans le texte analysé, et une pour chaque caractère non-terminal potentiel, avec son bit de poids fort à 1) ; chaque case peut contenir soit 0 (représentant l'échec), soit -127 (représentant le succès), soit une valeur positive (représentant un décalage vers l'état dont le numéro est cette valeur ; aucun décalage vers 0 n'est donc possible, ce qui n'est pas un problème pour les tables construites avec l'algorithme SLR), soit une valeur négative (représentant une réduction par la règle dont le numéro est l'opposé de cette valeur ; les numéros de règles commencent donc à 1, ne peuvent dépasser 126, et sont décalés par rapport à l'indexation de la structure de données `grammar`).

L'utilisation de ces différents champs, ainsi que la manière dont l'indice d'une règle permet de récupérer cette règle, est explicité par la fonction `print_table` fournie. Exécutez la, et lisez la attentivement.

2 L'analyseur que vous devez écrire

Vous devrez écrire un analyseur, prenant en entrée une grammaire, une table d'analyse LR de cette grammaire (si la table ne correspond pas à la grammaire, le comportement de l'analyseur n'est pas défini), et une chaîne de caractère à analyser, et exécutant l'automate à pile sur cette chaîne de caractères.

Il devra de plus construire, au cours de l'analyse, un arbre d'analyse de la chaîne pour la grammaire donnée. A la fin de l'exécution, il dira si le texte est accepté ou pas, et affichera l'arbre d'analyse, sous un format que vous déciderez (cela peut être celui vu en TD avec en préfixe de tout parenthésage l'étiquette de la racine du sous-arbre correspondant, comme dans les exemples donnés dans cet énoncé), comme illustré par un exemple à la figure 1.

```
           Flot  |  Pile
    ---------------------------------------
        aaaabbbb | 0
d2       aaabbbb | 0a2
d2        aabbbb | 0a2a2
d2         abbbb | 0a2a2a2
d2          bbbb | 0a2a2a2a2
r1          bbbb | 0a2a2a2a2S3
d4           bbb | 0a2a2a2a2S3b4
r0           bbb | 0a2a2a2S3
d4            bb | 0a2a2a2S3b4
r0            bb | 0a2a2S3
d4             b | 0a2a2S3b4
r0             b | 0a2S3
d4               | 0a2S3b4
r0               | 0S1
                accept
S(a()S(a()S(a()S(a()S()b())b())b())b())
```

FIGURE 1 – Affichage de l'analyseur que vous devrez écrire quand il est exécuté sur la grammaire $S \to aSb|\varepsilon$, sa table SLR, et le mot d'entrée *aaaabbbb*

3 Pour lire grammaires et tables dans un fichier

Pour vous aider dans ce travail, un code vous permettant de lire une grammaire et une table dans un fichier texte au format décrit ci-dessous vous est fourni. La fonction `read_file` lit un tel fichier et renvoie une grammaire et une table aux formats décrits ci-dessus.

L'analyseur est fourni dans un format de fichier texte ASCII. Le fichier commence par une description de la grammaire (hors contexte), dont le format est le suivant :

- l'alphabet de la grammaire est l'alphabet ASCII (à l'exception du retour à la ligne et du « $ », réservé à représenter les non-terminaux dans les productions des règles, et le caractère fin de chaîne dans la table) ;
- chaque règle est représentée par une ligne (le retour à la ligne ne peut donc pas faire partie de l'alphabet de la grammaire, et l'analyseur ne pourra donc pas travailler sur des textes multi-lignes) ;
- la ligne commence par un caractère représentant le caractère non-terminal membre droit de la règle (un non-terminal a donc un nom constitué nécessairement d'un seul caractère : c'est une limitation de ce format empêchant de manipuler des grammaires ayant plus de 94 non-terminaux) ; suit la production de cette règle précédée de « : » : tous les caractères jusqu'à la fin de la ligne font partie de la production, y compris les espaces (attention donc à ne pas « laisser traîner » une espace en fin de ligne ; dans la production, un non-terminal est représenté par son nom précédé du symbole « $ », évitant toute confusion avec le non-terminal représenté par le même caractère.

La table d'analyse est construite de la façon suivante :

- les lignes sont séparées par des retours à la ligne ;
- à l'intérieur d'une même ligne, les éléments de colonnes consécutives sont séparés par une tabulation ;
- la première ligne contient les symboles manipulés par la grammaire, avec d'abord les terminaux, puis la fin de chaîne (représentée par $ dans cette ligne), et enfin les caractères non-terminaux ; sa première case est toujours vide ;
- une ligne commence par un entier qui ne sert concrètement à rien mais permet de rendre la table plus lisible : la première ligne, même si elle est numérotée 15, correspondra toujours à l'état 0 ;
- les cases suivantes peuvent être vide (représentant l'échec de l'analyse), contenir a (représentant le succès de l'analyse), d suivi d'un nombre (représentant le décalage vers l'état de numéro donné) ou r suivi d'un nombre (représentant la réduction par la règle de numéro donné ; attention, la numérotation des règles commence à 1, alors que leur indexation commence à 0 dans les structures de données détaillées dans la suite) ;
- une ligne peut être interrompue avant que toutes ses colonnes soient données, les cases restantes étant alors considérées (logiquement) comme vides ;
- aucun autre caractère (espaces, notamment) n'est supporté.

Il ne vous est pas demandé pas de construire ces fichiers à partir de ces grammaires, mais juste d'exécuter

l'automate à pile LR ainsi décrit sur une chaîne de caractère fourni. Quatre fichiers de test vous sont fournis. Vous êtes encouragés à en écrire quelques autres à partir de grammaires, vues en cours ou pas, qui vous permettront de tester votre programme.

Un code vous est fourni pour lire un tel fichier. La fonction `read_file` renvoie une grammaire et une table que votre analyseur pourra directement utiliser. Par convention, on choisit de représenter les non-terminaux par leur code ASCII avec le bit de poids fort à 1 (le code ASCII comprend 128 caractère et est codé sur 7 bits ; le premier bit est toujours à 0, afin de permettre des extensions de ce code, telles que les « ISO latin »). Dans la table, les décalages sont indiqués par des valeurs positives, les réductions par des valeurs négatives. Les fonctions `print_grammar` et `print_table` peuvent vous permettre de mieux comprendre l'encodage.

4 Les conditions du rendu de ce projet

Vous me remettrez un projet par groupe de un ou deux étudiants, par mail (à l'adresse `devan.sohier@uvsq.fr`) le 6 janvier au plus tard. Ce mail contiendra deux pièces jointes portant les noms des étudiants du groupe :

— l'une, au format tgz, contenant un répertoire portant le même nom, le code source et un Makefile ;

— l'autre, au format pdf, lisible par Adobe Reader, contiendra votre rapport.

L'invocation de `make` doit produire un exécutable appelé `LRanalyzer`. Chaque fichier source doit également contenir en commentaire le nom de ses auteurs. Votre code ne doit pas faire appel à des bibliothèques externes non-standard et doit compiler sous Gcc ou clang. Un exemple d'utilisation de votre projet est montré à la figure 2.

Un rendu ne satisfaisant pas ces conditions encourt le risque de ne pas être corrigé.

5 La notation de ce projet

L'élément essentiel de notation est le rapport. Il devra donc être soigné. Son introduction et sa conclusion devront contextualiser le projet dans le cadre de la chaîne de compilation et plus particulièrement de l'analyse LR et non de votre situation personnelle [1].

Votre rapport ne doit pas être un manuel d'utilisation, et devra au contraire expliquer les choix que vous aurez opérés. Il existe notamment plusieurs manières d'implémenter piles et arbres : vous devrez expliquer et argumenter les choix que vous avez faits (en l'occurrence, il n'y a pas dans l'absolu de bons choix — même s'il peut en exister de mauvais), ainsi que tout autre parti pris d'implémentation. Votre rapport ne peut comporter d'éléments de code que de façon très limitée, comme support à vos explications.

L'utilisation de sources bien référencées est encouragée. Le plagiat (c'est-à-dire, vous attribuer le travail d'un autre, en l'utilisant sans le mentionner et sans le modifier substantiellement) est par contre proscrit, tant en matière de rapport que de code : il vous vaudra un 0, sans préjuger de procédures disciplinaires éventuelles qui s'ensuivraient et pourraient vous coûter jusqu'à une exclusion pendant 5 ans de l'enseignement supérieur.

La fonctionnalité de votre code sera bien sûr prise en compte (voir la section 4), ainsi que votre code, qui devra donc être lisible et structuré.

Sur la base d'un projet fonctionnel, il existe de nombreuses pistes d'amélioration (la liste ci-dessous n'est pas limitativet) :

— la possibilité d'avoir des nombres quelconques de non-terminaux (et donc des non-terminaux portant des noms de plusieurs caractères) ;

— des affichages sous un format plus réutilisable (typiquement, en LaTeX) ;

— une fonction de lecture de fichiers plus propre incluant tous les contrôles mémoire et fichiers opportuns, et sans limitations sur les tailles des grammaires qu'elle peut traiter.

Sans que cela soit impératif, il sera apprécié que vous en exploriez certaines (mais, j'insiste, sur la base d'un projet déjà fonctionnel).

1. « *Dans le cadre du cours formidable de M. Sohier que je suis parce que je veux devenir ingénieur…* » : toute flatteuse qu'elle soit, et que je croie ou non à sa sincérité, ce n'est pas une bonne introduction. Je sais déjà que c'est dans le cadre d'un cours d'ingénierie, je peux m'intéresser à vos ambitions, mais pas dans le cadre d'un rapport. Ce qui m'intéresse c'est la manière dont vous replacez votre travail dans un contexte scientifique et technique. De même pour la conclusion en « *J'ai beaucoup appris de ce projet* » : outre que c'est mon rôle d'en juger, elle vous éloigne du problème, qui est de savoir comment on peut aller plus loin dans la thématique de ce projet.

```
$> tar xvfz nom1_nom2.tgz
$> cd nom1_nom2
$> make
$> cat ../test4
A:$Aa$B
A:$B
B:$Bbc
B:c
        a       b       c       $       A       B
0                       d3              1       2
1       d4                      a
2       r2      d5              r2
3       r4      r4              r4
4                       d3                      6
5                       d7
6       r1      d5              r1
7       r3      r3              r3

$> ./LRanalyzer ../test4 "cacbcbcac"
            Flot  |  Pile
    ------------------------------------
        cacbcbcac | 0
d3       acbcbcac | 0c3
r3       acbcbcac | 0B2
r1       acbcbcac | 0A1
d4        cbcbcac | 0A1a4
d3         bcbcac | 0A1a4c3
r3         bcbcac | 0A1a4B6
d5          cbcac | 0A1a4B6b5
d7           bcac | 0A1a4B6b5c7
r2           bcac | 0A1a4B6
d5            cac | 0A1a4B6b5
d7             ac | 0A1a4B6b5c7
r2             ac | 0A1a4B6
r0             ac | 0A1
d4              c | 0A1a4
d3                | 0A1a4c3
r3                | 0A1a4B6
r0                | 0A1
                  accept
A(A(A(B(c()))a()B(B(B(c())b()c())b()c()))a()B(c()))
$>
```

FIGURE 2 – Tout rendu doit pouvoir être manipulé ainsi sous peine de ne pas être corrigé (`nom1` et `nom2` doivent bien sûr être remplacés par les noms des étudiants du groupe)