Théorie des langages
TP 2 - Une introduction au parsing LL(1)

Jonathan Fabrizio et Adrien Pommellet, EPITA

9 octobre 2023

Téléchargez au préalable les fichiers `parsing.c`, `parsing.h` and `grammar0.c` sur Moodle.

1 Un parser en C

Notre objectif est de concevoir une implémentation simple en C de l'algorithme LL(1) afin de concevoir un analyseur syntaxique. Vous n'aurez besoin aujourd'hui de rien d'autre que de votre fidèle compilateur C.

1.1 Manipuler le flux d'entrée

Nous n'allons pas encore directement interfacer le parser avec un lexer : nous nous contenterons de manipuler des symboles terminaux de type `char`, le flux d'entrée étant alors un pointeur de type `char **` vers une chaîne de caractères. Il nous faut de plus inclure un fichier d'en-tête `parsing.h` afin d'importer les fonctions suivantes sur le flux d'entrée :

```
void eat(char ** stream_pointer, char token)
void wrong_look_ahead(char ** stream_pointer, char symbol)
```

La fonction `eat` vérifie si le premier caractère du flux d'entrée pointé par `stream_pointer` est bien `token` : si c'est le cas, elle incrémente le flux d'entrée de manière à ce qu'il pointe vers le caractère suivant. Sinon, elle déclenche une erreur et affiche le message suivant sur le flux d'erreur avant d'arrêter l'exécution du programme :

```
Read token '%**stream_pointer', expected token '%token' instead.
```

La fonction `wrong_look_ahead` déclenche une erreur et affiche le message suivant sur le flux d'erreur :

```
No rule matched to non-terminal '%symbol' can generate look-ahead '%**stream_pointer'.
```

1.2 Concevoir un parser LL(1)

LL(1) est somme toute un algorithme simple ; l'implémentation en C d'un parser LL(1) pour une grammaire donnée suit les étapes suivantes :

1. Calculez le tableau LL(1) et vérifiez que la grammaire est bien LL(1) en premier lieu ; cela semble évident, mais il est toujours bon de le rappeler.
2. Pour chaque non-terminal `X`, écrivez une fonction `void X(char ** stream_pointer)`.
3. Dans le corps de la fonction `X`, utilisez une instruction `switch` ou `if` pour filtrer les différentes valeurs possibles du regard `**stream_pointer` :

```
switch (**stream_pointer) {
  case 'a':
    // Apply rule (X, a) of the LL(1) parsing table, assuming it exists.
    break;
  case 'b':
    // Apply rule (X, b) of the LL(1) parsing table, assuming it exists.
    break;
  default:
    // Catch the terminal symbols that are not matched to any rule.
    wrong_look_ahead(stream_pointer, 'X');
}
```

Notez que le symbole non-terminal `$` correspond dans notre cas au symbole de fin de chaîne `'\0'`.

4. Pour appliquer une règle de production de la grammaire, on lit sa partie droite de gauche à droite en mangeant (`eat`) les terminaux et appelant les fonctions associées aux non-terminaux au cours de cette lecture. Par exemple, pour appliquer la règle $X \to aYb$, on écrit les instructions suivantes dans le corps de `X` :

```
eat(stream_pointer, 'a');
Y(stream_pointer);
eat(stream_pointer, 'b');
```

5. Une fois l'unique règle axiomatique $Z \to S\$$ appliquée, le parsing du flux d'entrée est terminé.
6. Pour démarrer le parsing, il faut appeler la fonction axiome `Z`.

Le fichier `grammar0.c` décrit un parser LL(1) qui reconnaît la grammaire $\mathcal{G}_0$ suivante :

$$Z \to S\$ \tag{1}$$
$$S \to (S) \tag{2}$$
$$\mid n \tag{3}$$

Question 1. Compilez le fichier `grammar0.c` et testez sur quelques exemples le parser ainsi produit en passant des chaînes de caractère en argument au programme. Rappelez-vous que la chaîne passée en entrée doit être entre guillemets si elle contient des parenthèses.

Question 2. On crée une nouvelle grammaire $\mathcal{G}_1$ en ajoutant une règle $S \to \{S\}$ à $\mathcal{G}_0$. Créez un nouveau parser LL(1) `grammar1.c` basé sur `grammar0.c` qui reconnaît $\mathcal{G}_1$. Puis testez-le sur quelques exemples.

2 Analyse d'expressions arithmétiques

Nous souhaitons écrire un parser LL(1) qui puisse reconnaître et calculer des expressions arithmétiques simples.

2.1 Parsing de sommes d'entiers

Soit la grammaire LL(1) $\mathcal{G}_2$ suivante :

$$
\begin{aligned}
Z &\to S\$ && (1)\\
S &\to IU && (2)\\
U &\to +IU && (3)\\
&\mid \varepsilon && (4)\\
I &\to n && (5)
\end{aligned}
$$

Elle engendre le langage `n`, `n+n`, `n+n+n`, etc. La factorisation des préfixes grâce au non-terminal U permet de garantir que cette grammaire est bien LL(1).

Question 3. Calculez rigoureusement le tableau LL(1) de $\mathcal{G}_2$ en utilisant Null, First, et Follow.

Question 4. Écrivez un nouveau parser LL(1) `grammar2.c` reconnaissant $\mathcal{G}_2$, puis testez-le sur quelques exemples.

2.2 Calcul de sommes d'entiers

On introduit une nouvelle grammaire $\mathcal{G}_3$ en remplaçant la cinquième règle de $\mathcal{G}_2$ par une nouvelle règle $I \to \textit{integer}$, où *integer* représente n'importe quelle suite non-vide de chiffres. Notre prochain objectif est de non seulement parser, mais aussi d'évaluer les expressions arithmétiques engendrées par $\mathcal{G}_3$. Pour ce faire, nous devons nous assurer que les fonctions associées aux non-terminaux ont pour type de retour `int` et non simplement `void` et qu'elles effectuent des additions lors de l'application de règles de production ; de plus, il faut entièrement réécrire la fonction `I`.

Question 5. Réécrivez la fonction `I` de `grammar2.c` de manière à manger le premier entier lu sur le flux d'entrée et à renvoyer sa valeur. Par exemple, étant donné un flux d'entrée `0123+45`, `I` doit manger `0123` et renvoyer la valeur `123` de type `int`. Utilisez la fonction `isdigit` de la bibliothèque `ctype` pour vérifier si le regard est bien un chiffre ; si ce n'est pas le cas, le parsing doit s'arrêter.

Question 6. Écrivez un nouveau parser LL(1) `grammar3.c` reconnaissant $\mathcal{G}_3$ et évaluant puis affichant la valeur de l'expression arithmétique passée en entrée. Puis testez-le sur quelques exemples. Notez que la fonction `isdigit` peut être utilisée pour détecter les regards entiers.

2.3 Produits et soustractions

On considère désormais la grammaire LL(1) $\mathcal{G}_4$ suivante :

$$
\begin{aligned}
Z &\to S\$ && (1)\\
S &\to PU && (2)\\
U &\to +PU && (3)\\
&\mid \varepsilon && (4)\\
P &\to IV && (5)\\
V &\to *IV && (6)\\
&\mid \varepsilon && (7)\\
I &\to \textit{integer} && (8)
\end{aligned}
$$

Elle engendre le langage des expressions arithmétiques utilisant l'addition et le produit sur les entiers. Intuitivement, une telle expression est représentée sous la forme d'une somme S de produits P d'entiers. Grâce aux non-terminaux U et V, la factorisation des préfixes permet de s'assurer que cette grammaire est LL(1).

Question 7. Écrivez un nouveau parser LL(1) `grammar4.c` basé sur `grammar3.c` qui reconnaît la grammaire $\mathcal{G}_4$ et évalue puis affiche la valeur de l'expression arithmétique passée en entrée. Puis testez-le sur quelques exemples.

Question 8. On crée une nouvelle grammaire $\mathcal{G}_5$ en ajoutant une règle $U \to -PU$ à $\mathcal{G}_4$. Créez un nouveau parser LL(1) `grammar5.c` basé sur `grammar4.c` qui reconnaît $\mathcal{G}_5$ et évalue puis affiche la valeur de l'expression arithmétique passée en entrée. Puis testez-le sur quelques exemples.

2.4 Parenthèses et divisions

On étend enfin notre grammaire de manière à gérer les parenthèses et les divisions. La première extension est aisée ; la seconde, en revanche, va sans doute causer des problèmes inattendus d'associativité provoqués par l'arrondi de quotients sur les entiers à l'inférieur.

Question 9. On crée une nouvelle grammaire $\mathcal{G}_6$ en ajoutant une règle $I \to (S)$ à $\mathcal{G}_5$. Créez un nouveau parser LL(1) `grammar6.c` basé sur `grammar5.c` qui reconnaît $\mathcal{G}_6$ et évalue puis affiche la valeur de l'expression arithmétique passée en entrée. Puis testez-le sur quelques exemples.

Question 10. On crée une nouvelle grammaire $\mathcal{G}_7$ en ajoutant une règle $V \to /IV$ à $\mathcal{G}_6$. Créez un nouveau parser LL(1) `grammar7.c` basé sur `grammar6.c` qui reconnaît $\mathcal{G}_6$ et tente d'évaluer puis d'afficher la valeur de l'expression arithmétique passée en entrée. Puis testez-le sur les exemples `2/4*8`, `2*8/4` et `8/4*2`. Est-ce que le programme fonctionne toujours correctement ? Cette implémentation de la division est-elle associative à gauche ou à droite ? Notez qu'il y a diverses manières d'implémenter la division : il est par exemple possible de passer à la fonction `V` un ou plusieurs arguments supplémentaires qui permettent de transférer le numérateur et le dénominateur calculés jusqu'à présent.

3 Grammaires étoilées

Nous allons introduire une nouvelle classe de grammaires connue sous le nom de *grammaires étoilées* qui autorise l'usage de l'étoile de Kleene dans le corps des règles de production. Par exemple, la règle $X \to A(B)^*$ représente l'ensemble dénombrable de règles $X \to A$, $X \to AB$, $X \to ABB$, etc. On considère la grammaire étoilée $\mathcal{G}_8$ suivante :

$$Z \to S\$ \tag{1}$$
$$S \to I(/I)^* \tag{2}$$
$$I \to \mathit{integer} \tag{3}$$

$\mathcal{G}_8$ engendre les expressions arithmétiques avec division sur les entiers.

Question 11. Créez un nouveau parser LL(1) `grammar8.c` qui reconnaît $\mathcal{G}_8$ et évalue puis affiche la valeur de l'expression arithmétique passée en entrée. Pour ce faire, vous pouvez utiliser une instruction `while` dans le corps de la fonction `S`. Puis testez-le sur quelques exemples. Cette grammaire est-elle associative à droite ou à gauche ?