Université de Lille 1 SCIENCES ET TECHNOLOGIES

Master informatique
2e semestre
Module de Compilation

l'Analyse Lexicale

Gilles GRIMAUD Samuel HYM

Premier jet, Janvier 2018

Ce sujet est disponible en ligne à `www.lifl.fr/~grimaud/UE/Compil/TP_intro/`.

1 Avant propos

Ce premier sujet de TP autour de la compilation pourra être réalisé dans le langage de votre choix. Les enseignants vous invitent cependant à privilégier l'un des langages suivants :

— C ;
— Haskell ;
— Java ;
— python (2 et 3) ;
— ruby ;
— C++ ;

Si vous souhaitez en utiliser un autre, merci d'en informer les enseignants avant de commencer. Dans un soucis de clareté, le sujet est rédigé en présentant des prototypes de fonction C.

Les sujets de compilation comportent un certain nombre d'encarts. Ces encarts contiennent des developpements autour du sujet. Prenez le temps de les lire. Assurez vous d'avoir compris les points qui y sont décrits. N'hésitez pas à discuter de ces points avec les enseignants, si des zones d'ombres subsistent.

Ce TP doit être terminé dans à la fin de la première séance. Le logiciel produit sera intensivement réutilisé par la suite. Il en sera de même pour les programmes des 4 TPs suivants. Ecrivez les avec soin. Testez les. Utilisez sans retenue toute les bonnes pratiques qui vous ont été enseignées. Un compilateur est un programme composé de nombreuses briques logicielles. Chaque brique doit être de bonne qualité pour que l'ensemble de l'édifice ne s'effondre pas.

Pour vous aider à réaliser ce TP vous êtes invité à faire un `fork` sur le *gitlab* du projet `lexer` dans le groupe `ms2_Compil` disponible à l'url : `https://gitlab-etu.fil.univ-lille1.fr/ms2_Compil/lexer.git`. Les exemples de code fournis dans ce dépot utilisent le langage C et haskell. Nous vous invitons à les adapter au langage de votre choix.

Lorsque vous traitez une question, commencez par donner la réponse dans un fichier `réponse.md` à la racine de votre projet en précisant à quelle question vous répondez.

2 Le « champ lexical »

L'analyse lexicale consiste à traiter une *source*, présenté sous la forme d'une séquence de caractères (chaine de caractère ou fichier texte par exemple) de tel sorte que chaque unité lexicale[1] c'est-à-dire chaque « mot » du langage source soit clairement identifié. Le programme qui découpe ainsi une *source* est appelé un analyseur lexical, un *lexer* ou encore un *tokenizer*.

Pour concevoir un tel programme il faut définir le « champ lexical » du langage utilisé par la *source* que l'on souhaite traiter. Dans le cadre de ce TP c'est un sous-ensemble du langage source `Javascript` qui est étudié. Voici une liste des « mots » du langage qui seront utilisés dans un premier temps :

1. l'unité lexicale est aussi appelée *token*

Générateurs d'analyseurs lexical

Le traitement d'une *source* est finalement assez simple. Il faut définir la liste des *tokens*, c'est-à-dire le « champ lexical ». Cela peut être fait de manière assez simple en utilisant par exemple des expressions régulières (si vous ne savez toujours pas ce qu'est une expression régulière, il est encore tant de vous instruire : `https://fr.wikipedia.org/wiki/Expression_r\unhbox \voidb@x \bgroup \let \unhbox \voidb@x \set`
Puis à chaque symbole du langage ainsi reconnu il faut associé une identité (un identifiant) qui pourra être exploité par la suite du processus de compilation.
Ainsi il existe un certains nombre de programmes qui sont capables de prendre un fichier source décrivant les différents mots du langage et qui exécute un code choisi par le développeur en fonction du mot identifié dans le *source*.
Parmi les logiciels les plus anciens (et sans doute les plus connus), il y a `lex` ou sa version libre : `flex`.
Parmis les plus récents et les plus pertinents il y a `ragel` ou encore le package `ply.lex` qui permet d'écrire un analyseur lexical pour `python`.

`function` déclare une fonction ;

`return` termine une fonction et retourne (éventuellement) une valeur ;

`var` déclare une (ou une liste de) variable(s) ;

`=` déclare une opération d'affectation (algorithmique) ;

{, } déclare le début et la fin d'un bloc d'opérations ;

+,−,∗,/ effectue une opération arithmétique (sur des entiers dans un premier temps) ;

==,! =,<,>, <=, >= effectue des comparaisons (entières dans un premier temps) ;

`if` déclare un « si » algorithmique ;

`else` déclare un « sinon » algorithmique ;

`while` déclare un « tantque » ;

`for` déclare un « pour », à la manière du langage C (dans un premier temps) ;

`//` les ... sont des commentaires que l'analyse lexicale supprime ;

`/*...*/` les ... sont des commentaires que l'analyse lexicale supprime ;

`"`*`string`*`"` *string* sont une chaine de caractère quelquonque ;

idenfier est un identifiant qui commence par une lettre (majuscule ou minuscule) éventuellement suivi d'un nombre arbitraire de caractères alphanumériques ;

numbers *numbers* est un entier formé par une succéssion de caractères représentants des chiffres (qui peuvent être décimaux ou hexadécimaux si le nombre commence par `0x`) ;

`'`*`char`*`'` *char* est un caractère ASCII (UTF ?) ;

3 Mon premier analyseur lexical

Le but d'un analyseur lexical est de reconnaitre des mots du langage qui sont aussi appelés token.

Un mot du langage est définit par un « identifiant » qui permet de le distinguer des autres. L'identifiant « `PLUS` » est par exemple distinct de identifiant « `STRING` »...

Certains mots sont entièrement décrit par ce seul identifiants. C'est le cas, par exemple du token `PLUS` ou même du token `WHILE`. Cependant d'autres mots sont associés à des valeurs distinctes d'une occurence du token à une autre. Le token `STRING` peut par exemple être associé, dans une occurence, à la chaine de caractère `"Hello world!"` alors qu'une autre occurence l'associe à `"Bye bye..."`. Aussi en plus de son identifiant, un mot du langage est définit par une valeur propre potentiellement différente à chaque occurence.

3.1 Les structures de données

Le champ lexical du langage que vous devrez traité a été définit précédement. Dans cette définition vous pouvez distingué 4 sortes de mots (token) :

les signes Il s'agit des opérations arithmétiques et logiques, ainsi que des caractères de début et de fin de bloc de code ;

les mots clefs ce sont les mots clefs du langage, qui ont un sens implicite et unique ;

les identifiants qui, contrairement au mots clefs, ne sont pas définit par le langage, mais qui peuvent être explicitement reconnus par le fait qu'ils commencent par un caractère de l'alphabet et ne sont suivis que de caractères alphanumériques ;

les constantes tel que les nombres, les chaines de caractères... Elles sont toutes différentes les unes des autres, comme les symboles, mais elles peuvent être distinguées d'eux par leurs structures (les nombres ne sont composés que de chiffres, les chaines de caractères commencent et finissent par un caractère spécial `"`,...).

Pour répondre aux questions suivantes, consultez dans le repertoire `src` de votre `fork` le fichier `symbols.h`. Il définit une serie de code pour tous les mots du langage qui ont été définit précédement (et pour bien d'autres mots possibles). Ce fichier définit un identifiant unique pour chaque tokens. Si vous n'utilisez pas le C adaptez ce fichier pour le langage de programmation que vous avez choisi.

Dans le répertoire `src` créez un fichier source nommé `lexer.xxx` (ou xxx représente l'extension de votre langage, par exemple `.c`, `.js` ou `.hs` par exemple).

Exercice 1 (une table des opérateurs)
Dans le fichier `lexer.xxx`, réalisez une table qui énumère l'ensemble des signes du langage. Cette table sera nommée par la suite textttsigns_table et qui liste tout les signes du langage qui ont été définis dans l'énnoncé. Pour chaque entrée token de cette table la table fait référence à un numero de token et en guise de valeur à la chaine de caractère qui lui correspond (par exemple `"+"` pour le token `PLUS`). □

Exercice 2 (une table des mots clefs)
Réalisez un table qui liste tout les mots clefs du langage qui ont été définis dans l'énnoncé et pour chacun d'eux, en guise de valeur, la chaine de caractère qui lui correspond (par exemple `"while"` pour le token `WHILE`). □

Exercice 3 (Une structure token)
Proposez une structure de donnée (ci-aprés nommée `token`) représentant un token lu par l'analyseur lexical.

Cette structure de donnée compilera les informations suivante :

`id` un numero unique de token, conformément aux déclarations trouvées dans le fichier `symbole.c` ;

`nom` le nom du token ;

`valeur` la valeur du token, qui peut être différente du nom pour les identifiants, les nombres et les chaines de caractères par exemple ;

`ligne` le numéro de la ligne dans le fichier source ou le token a été identifié ;

`col` le numéro du premier caractère du token dans la ligne.

□

Exercice 4 (la fonction `dump_token`)
Implémentez la fonction `dump_token`. Cette fonction prend un token en paramêtre, et elle produit sur la sortie standard une séquence d'octets formatée comme suit :

— 1 octet pour le numéro du token ;
— 2 octets pour le numéro de la ligne ou le token a été lu dans le source ;

— 2 octets pour le numéro de la colonne ou le token a été lu dans le source ;
— 2 octets qui forment un nombre indiquant le nombre d'octets qui suivent ;
— n octets pour indiquer la valeur du token (n est différent de 0 seulement pour les token `IDENTIFIER`, `NUMBER`, `FLOAT`, `STRING` et `CHAR`.

□

Exercice 5 (au plus simple...)
Realisez un programme principal qui charge entièrement le contenu d'un fichier source (nom de fichier en paramêtre) en mémoire, sous la forme d'une simple chaine de caractère par exemple. Votre programme principal appelle ensuite en boucle une fonction `lex` (qui sera implémentée dans la question suivante) qui retourne un token. Lorsque la fin de la chaine est atteinte, un token particulier, `feof_token`, est retourné par la fonction `lex`. La boucle principale termine alors. A chaque itération de la boucle, le token retourné par le fonction `lex` est simplement *sérialisé* sur la sortie standard à l' aide de la fonction `dump_token` (implémentée précédement). □

Exercice 6 (la fonction `lex`)
Implémentez une fonction `lex`. Cette fonction lit une série de caractères depuis le fichier source et créée une structure `token`. Plus précisément : La fonction `lex` commence pas ignorer les "blancs" via une fonction `skip_blanks`. Notez que les "blancs" a ignorer sont les espaces, mais aussi les tabulations, les retours à la ligne et les nouvelles lignes. Notez encore que la fonction `skip_blanks` ignore les commentaires (de la forme `/* */` et de la forme `//`). Ensuite la fonction `lex` cherche à reconnaitre soit :

1. un nombre (formé par une suite de chiffre) ;
2. une chaine de caractère (décrite entre 2 `('"')`) ;
3. un signe (présent dans la table des signes) ;
4. un identifiant (formé par une lettre, suivie d'une serie de chiffres et de lettre) :
   (a) qui est alors un mot clef, s'il est dans la table des mots clefs du langage,
   (b) ou un veritable identifiant si ce n'est pas un mot clef.

4 Grands fichiers et (backtracking)

L'analyseur lexical considéré dans ce TP lit des octets depuis un fichier (ou, par défaut, depuis l'entrée standard). Pour cela, le standard POSIX propose, en C, la fonction `fgetc`. Si vous utilisez un autre langage trouvez la fonction équivalente pour vous.

Exercice 7 (lire un caractère)
Implémentez une fonction `int readchar()` qui retourne un caractère lu, soit, depuis un fichier désigné comme source, soit, depuis l'entrée standard. Bien sur, à chaque appel successif, la fonction retourne le caractère suivant, et `FEOF` (ou équivalent) une fois la fin de fichier atteinte. Cette fonction veillera à tenir à jour des informations relative au numéro du caractère dans le fichier à la ligne courante et de la colonne courante dans le fichier source. □

Pour reconnaitre un mot, l'analyseur lexical lit des caractères. Le plus souvent, un mot n'est reconnu que lorsque le premier caractère suivant ce mot à été lu. Cela peut être génant. par exemple lorsque l'analyseur lexical traite la séquence `"134+17"`, il ne sait qu'il a fini de lire le token de type `NUMBER` de valeur `134` que lorsqu'il a lu le caractère `+` qui formait le token suivant dans le source.

Exercice 8 (dé-lire un caractère !)
Implémentez une fonction `int unreadchar(int c)` qui annule la lecture du caractère `c`. Cette fonction retourne `true` si, suite à cette annulation il sera encore possible d'annuler des lectures, et `false` si cette annulation est la dernière possible. □

Exercice 9 (lire et relire une caractère...)
Modifier la fonction `int readchar()` afin qu'elle retourne les caractères qui ont été « dé-lus » s'il y en a, plutot que des charactères venus du fichier. Notez encore que si on dé-lit `'='` puis `'!'` on lira par la suite `'!'` puis `'='` avec la fonction `readchar()`. □

4.1 Lecture d'un mot

Exercice 10 (Ignorer les caractères non significatifs)
Implémentez une fonction `void skip_blanks()` qui lit tout les caractères non significatifs depuis le fichier source (en utilisant la fonction `readchar`). Cette fonction termine lorsqu'elle trouve un caractère significatif ou lorsqu'elle atteint la fin du fichier. Ce premier caractère utile est « dé-lu » (via `unreadchar`) pour qu'il puisse être traité par la suite. □

Exercice 11 (Lire un mot clef)
Vous avez réalisé un table `keyword_table` qu regroupe l'ensemble des mots clefs de votre langage. Réalisez une fonction `struct token_s *read_keyword()` qui retourne l'entrée de la table correspondant à ce qui est lu avec `readchar()` ou `NULL` si ce qui a été lu ne correspond à aucune entrée. Dans ce dernier cas assurez vous, avant que la fonction ne termine d'avoir « dé-lu » (via `unreadchar`) les caractères qui ne formait pas un mot clef du langage. Notez qu'un mot clef ne peut être reconnu que s'il est complet et non imédiatement suivi d'autre chose. Par exemple `whiletrue` ne peut pas être reconnu comme une séquence des deux mots clefs `WHILE` et `TRUE`. □

Exercice 12 (Lire un signe)
De la même manière que pour les mots clefs, vous avez réalisé une table `signs_table` qui regroupe l'ensemble des signes définis dans le langage source que vous traitez. Réalisez une fonction `struct token_s *read_sign()` qui retourne l'entrée de la table correspondant à ce qui est lu avec `readchar()` ou `NULL` si aucune entrée de la table ne correspond. Si rien ne correspond assurez vous d'avoir « dé-lu » (via `unreadchar`) les caractères qui ne formaient pas un signe. Notez que contrairement aux mots clefs, les signes concaténés sont immédiatement reconnus pour ce qu'ils sont. Par exemple =− doit être reconnu comme l'enchainement de deux tokens : `ASSIGNMENT` suivi de `MINUS`. □

Exercice 13 (Lire un identifiant)
Pour qu'un mot soit considéré comme un identifiant, il doit commencer par un premier caractère alphabetique, et il peut être suivi d'un nombre arbitraire de caractère alphanumériques. Réalisez une fonction `int read_identifier(struct token_s *token)` qui, lorsque elle reconnait un identifiant, renseigne la structure token dont l'adresse est passée en paramêtre et retourne 1. Cette structure retourne 0 sinon. □

Exercice 14 (Lire un nombre)
Pour qu'un mot soit considéré comme un identifiant, il doit commencer par un premier caractère alphabetique, et il peut être suivi d'un nombre arbitraire de caractère alphanumériques. Réalisez une fonction `int read_number(struct token_s *token)` qui, lorsque elle reconnait un identifiant, renseigne la structure token dont l'adresse est passée en paramêtre et retourne 1. Cette structure retourne 0 sinon. □

Exercice 15 (La fonction principale)
la fonction `struct token_s lex(char *value,int value_ln)` retourne le prochain mot lu dans le fichier source. Si ce mot est précédé de caractères non significatifs (espace, tabulation retour à la ligne ou encore commentaires) ces derniers sont éliminés avant que la fonction lex ne cherche à décoder le prochain token du fichier.

Notez bien que cette fois ci la fonction `lex` traite le fichier chargé "en flux", sans l'installer entièrement en mémoire avant de commencer à le traiter. Au contraire, elle consomme un minimum de caractères dans le flux avant de produire le token associée sur la sortie standard. □

Exercice 16 (Le programme principal)
Réalisez un programme principal qui utilise cette nouvelle fonction `lex` pour parser les fichiers sources. Comparez votre sortie avec la sortie précédente. □

Bravo ! Vous savez creer des analyseurs lexicaux !

5 lexer générique

Question 16.1 Comment rendre votre

Fin du sujet