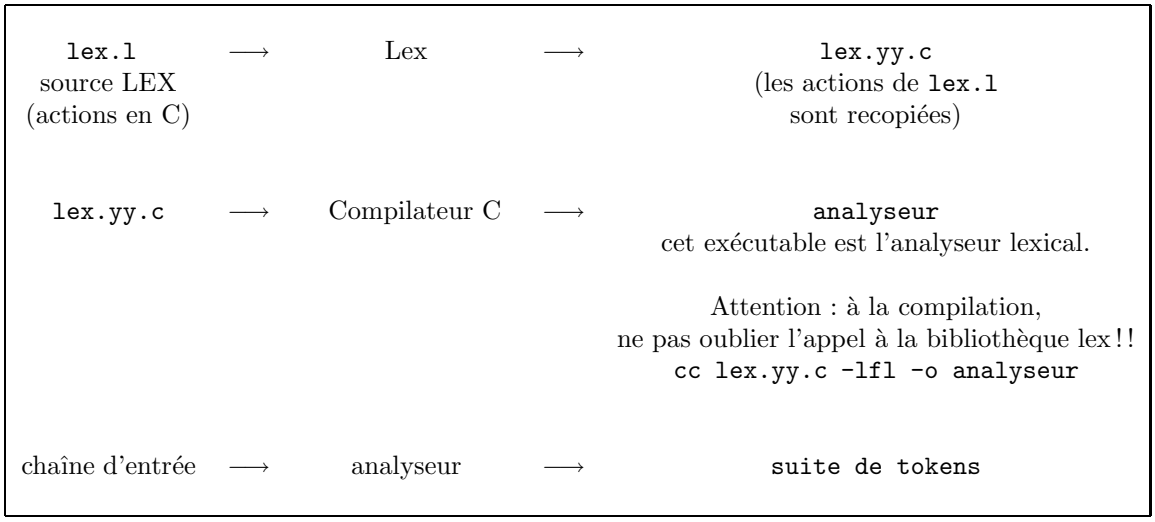
Présentation de Lex

Lex est un générateur de programme pour le traitement lexical de chaînes de caractères, en particulier pour l'analyse lexicale. Un source Lex est une spécification de haut niveau sous forme d'expressions rationnelles décrivant les classes de tokens et d'actions à exécuter à la reconnaissance d'un token. Ces actions sont écrites en `C`.

1 Schéma d'utilisation de Lex



Le fichier `lex.yy.c` est donc un programme `C` comprenant :
- une représentation de la table de transitions de l'automate construit à partir des expressions rationnelles ;
- une routine standard d'utilisation de cette table pour simuler le fonctionnement de l'automate. Cette routine s'appelle `yylex`.
- les actions de `lex.l` recopiées.

2 Description d'un source Lex

La forme générale d'un source Lex est la suivante :

```
déclarations                    <- optionnel
%%
règles de transition
%%                              <- optionnel
procédures auxilliaires         <- optionnel
```

2.1 Les déclarations (optionnelles)

On y trouve des déclarations de variables globales et des définitions rationnelles. Elles doivent commencer en première colonne. Toute ligne commençant par un blanc est intégralement recopiée dans `lex.yy.c`, ainsi que tout texte entre `%{...%}`.

Exemples de déclarations :

```
lettre    [A-Za-z]
chiffre   [0-9]
```

où `lettre` définit une variable globale qui pourra être utilisée dans les expressions rationnelles sous la forme `{lettre}` où `{` et `}` sont des métasymboles permettant de distinguer la suite de caractères `l.e.t.t.r.e` et l'identificateur `lettre`.

2.2 Les procédures auxiliaires (optionnelles)

Ce sont les routines utilisées dans les actions associées aux différentes expressions rationnelles, par exemple la procédure d'installation dans la table des symboles.

On peut y placer la fonction `main` dans le cas d'un analyseur seul. Dans ce cas la fonction `yylex` sera utilisée pour lancer la procédure d'analyse et dans la variable `yyin` sera stocké le descripteur du fichier à analyser (par défaut `stdin`). Les variables `yytext` et `yyleng` contiennent respectivement un pointeur sur le premier caractère de la chaîne reconnue et la longueur de celle-ci.

2.3 Les règles de transition

Elles sont de la forme :

```
Expression_rationnelle    Action
```

où `Action` est un fragment de code `C` à exécuter quand le token correspondant à l'expression rationnelle est reconnu.

Ainsi, la règle de transition

```
table   printf("objet'');
```

substitue `table` par `objet` mais aussi `tablette` par `objettte`....

La syntaxe des expressions rationnelles suit les règles classiques. Par exemple, les identificateurs peuvent être définis par

```
{lettre}({lettre}|{chiffre})*
```

où les accolades délimitent la variable globale `lettre`, les parenthèses délimitent une sous-expression, le métasymbole `|` désigne l'alternative et le métasymbole `*` désigne la répétition 0 ou plusieurs fois.

Les entiers sont définis par `{chiffre}+` où le métasymbole `+` désigne la répétition 1 ou plusieurs fois.

Les nombres décimaux sont définis par

```
{chiffre}+(\.{chiffre}+)?(E[-+]?{chiffre}+)?
```

où le métasymbole `?` définit une sous-expression optionnelle. Le symbole "." étant un métasymbole, le caractère "." est désigné par `"\."`.

Les **méta symboles** sont :

```
" \ [ ] ^ ? . * + | ( ) $ / { } %
```

et correspondent respectivement à :
- `" "` : tout ce qui se situe entre " " est pris comme chaîne de caractères. Ainsi "." désigne le caractère . et " " désigne un blanc significatif (par exemple `a" "b` permet de reconnaitre le token `a b`).
- \ : placé devant un méta-symbole, il le transforme en caractère (comme \.).
- `[ ]` définit une classe de caractères (i.e. représente un caractère). Par exemple, `[ab+]` reconnaît l'un des caractères `a`, `b` et `+`; `x[ab+]y` reconnait l'une des chaines `xay`, `xby` ou `x+y`.
  Les seuls opérateurs ou métasymboles actifs entre `[ ]` sont \, `-` et `^` . Le symbole `-` désigne un intervalle, le symbole `^` désigne le complémentaire et \ sert à représenter les escapes `\n`, `\t`, `\b`.
  Lorsque le caractère `-` doit être reconnu entre des crochets, il faut le placer en tête. Ainsi `[0-9]` désigne un chiffre mais `[-+0-9]` reconnait soit un chiffre, soit le symbole `-`, soit le symbole `+`.
  Le symbole `^` désignant le complémentaire doit être placé en tête. Ainsi `[^abc]` reconnait tout caractère sauf `a, b` ou `c`.
- `?` désigne une expression optionnelle. Par exemple `abc?d` reconnaît l'une des chaines `abd` et `abcd`.
- `.` remplace n'importe quel caractère sauf `\n`.

Notion de contexte sensitivité : on peut spécifier dans quelques cas précis qu'une expression ne doit être reconnue que dans un contexte donné.
- si le premier caractère est `^` , la chaîne ne sera reconnue qu'en début de ligne.
- si le dernier caractère est `$`, la chaîne ne sera reconnue qu'en fin de ligne
- le / définit un contexte sensitif à droite. Ainsi `ab/cd` reconnaît `ab` s'il est suivi de `cd`.

Pour associer la même action à plusieurs expressions rationnelles, on peut les regrouper :

```
expression1 |
expression2 |
    :
    :
expressionI      action ;
```

2.4 Les actions

L'action par défaut est la recopie de l'entrée sans modification dans le fichier de sortie. Elle est exécutée pour les chaines non reconnues. Une règle dont l'action est la recopie peut être omise. Inversement, si toute chaine d'entrée doit être reconnue et traitée (comme c'est le cas dans un compilateur), alors il faut des règles pour tout reconnaitre. L'action vide ignore l'entrée (ne pas recopie pas).

Les actions sont des fragments de programmes `C`. Ce sont les fonctions de la bibliothèque `C`, celles de la bibliothèque Lex, celles définies par le programmeur à la fin du source Lex.

3 Cas des règles ambiguës

On peut avoir plusieurs expressions rationnelles qui reconnaissent la même chaîne d'entrée. Dans ce cas Lex lève l'ambiguité en appliquant les règles suivantes :

1. le plus long préfixe est retenu ;
2. parmi plusieurs expressions rationnelles qui reconnaissent le même préfixe, la première est choisie.

Exemple : Considérons le source Lex suivant :

```
integer   action1;   {action correspondant
                      au mot réservé integer}
[a-z]+    action2;   {action correspondant
                      à un identificateur}
```

Les chaines de caractères suivantes sont reconnues ainsi :
- `"integers"` est pris pour un identificateur (8 caractères : règle 1).
- `"integer"` est pris pour le mot réservé integer (7 caractères reconnus par deux expressions : règle 2).
- `"int"` est pris pour un identificateur (3 caractères reconnus).

Ce mécanisme est utilisé pour gérer les mots-clés ou réservés. En listant en premier les expressions rationnelles décrivant les mots réservés, ils sont reconnus comme tels et non comme des identificateurs.