L'analyse lexical

Rappel

Dans cette phase, les données sont lues caractère par caractère, pour former les lexèmes, qui sont ensuite fournis à l'analyseur syntaxique.

Les lexèmes sont les non-terminaux de l'analyse lexicale. Les terminaux se composent des mots du langage de programmation (par exemple "x11" ou 'f" ou "23" ou "{").

Tous ces langages sont réguliers.

Les commentaires sont supprimés dans cette phase (nous n'allons pas nous en occuper dans ce cours). Normalement, on fournit de l'information sur la position des mots (dans le projet: la ligne).

Survol de cette période

- Langages réguliers
- Réalisation des langages réguliers en Java
- Fonctionnement d'un analyseur lexical

Langages réguliers

Rappel: Un langage est dit régulier s'il peut être décrit par un grammaire dont les règles ont la forme $A \to abc$ ou $A \to Babc$ ou $S \to \epsilon$ (symbole de départ S).

Mais aussi: Considérons un ordre $A_1 < \ldots < A_n = S$ sur les non-terminaux. Un langage est dit régulier s'il peut être décrit par un grammaire dont les règles ont la forme $A_i \to ab|c$ ou $A_i \to \mathcal{R}(\{\, A_1, \ldots, A_{i-1} \,\})$. (sans preuve)

$\mathcal{R}(A)$ est une **expression régulière** sur l'alphabet A (nous utilisons la syntaxe EBNF):

- ϵ est une expression régulière;
- a est une expression régulière pour chaque $a \in A$;
- si R et S sont des expressions régulières, alors RS (concaténation) et $R|S$ (choix) et $\{\, R \,\}$ (répétition, souvent écrit R^*) sont aussi des expressions régulières.

Exemple: Les identificateurs

$$\begin{array}{rcl} I & \to & L \,\{\, L \mid N \,\} \\ L & \to & 'a' \mid \ldots \mid 'z' \mid 'A' \mid \ldots \mid 'Z' \\ N & \to & '0' \mid \ldots \mid '9' \end{array}$$

Les **automates finis (non-)déterministes (DFA, NFA)** sont encore un autre moyen de décrire des langages réguliers. On s'en sert dans les générateurs d'analyseurs lexicaux.

Expressions régulières

Nous allons regarder la correspondance entre les expressions régulières et les grammaires régulières sur un exemple.

Prenons la grammaire (symbole de départ S):

$$\begin{array}{lcl} S & \rightarrow & BA \,|\, C \\ A & \rightarrow & a \,|\, c \\ B & \rightarrow & b \\ C & \rightarrow & \{\, B \,\} \end{array}$$

Il y a par exemple une hiérarchie $A < B < C < S$, mais aussi $B < A < C < S$ ou $B < C < A < S$ (puisque A et B et C sont utilisés par S, et B est utilisé par C).

$S \rightarrow BA \,\|\, C$	se transforme en	$S \rightarrow BA$ $S \rightarrow C$	partager les branches d'une choix
$S \rightarrow BA$	se transforme en	$S \rightarrow Ba$ $S \rightarrow Bc$	ajouter les côtés droits au non-terminal le plus à gauche
$C \rightarrow \{\, B \,\}$	se transforme en	$C \rightarrow C'$ $C' \rightarrow \epsilon$ $C' \rightarrow C'b$	réaliser la boucle

Dans ce cas particulier, il n'aurait pas été nécessaire d'introduire un non-terminal auxiliaire. Pourtant, des fois il peut être utile de faire ainsi.

En remontant la production d'ϵ vers $S \rightarrow \epsilon$ et en éliminant le C', on obtient:

$$\begin{array}{lcl} S & \rightarrow & Ba \\ S & \rightarrow & Bc \\ S & \rightarrow & \epsilon \end{array} \qquad \begin{array}{lcl} B & \rightarrow & b \\ C & \rightarrow & b \\ C & \rightarrow & Cb \end{array}$$

Le non-terminal A a complètement disparu.

Réalisation des langages réguliers

Supposons que nous avons une fonction `void nextChar();` qui consomme et renvoie le caractère suivant lu en entrée, ainsi qu'une fonction `void error();` pour signaler des erreurs. Le caractère lu en dernier (et qui est également le prochain à traiter) est stocké dans la variable `char ch;`

E	Pr(E) (programme de reconnaissance)
"x"	`if` (`ch` == "x") `nextChar()`; `else error()`;
EFG	Pr(E); Pr(F); Pr(G)
{ E }	`while` (`ch` commence E) { Pr(E) }

"a"E| "b"F | "c"G	`switch` (ch) {
	`case` "a": Pr("a"E); `break`;
	`case` "b": Pr("b"F); `break`;
	`case` "c": Pr("c"G); `break`;
	`default`: `error()`;
	}

Hypothèse importante: "E" et "b" et "c" sont distincts. Pourquoi?

On suppose que la grammaire permet d'analyser les données par la gauche. Ceci n'est pas garanti par les grammaires régulières, mais par les grammaires utilisant des expressions régulières.

La grammaire $\{\, S \to Sa, S \to a \,\}$ est régulière. Mais que se passe-t-il quand on écrit un programme qui la reconnaît?

Exemple: Les identificateurs

```
void ident() {
   if (isLetter(sym)) nextChar(); else error();
   while ((isLetter(sym)) || (isDigit(sym))) {
      switch (sym) {
      case ’a’: case ’b’: ... case ’z’:
      case ’A’: case ’B’: ... case ’Z’: letter(); break;
                                        // traitement des caracteres
      case ’0’: case ’1’: ... case ’9’: digit(); break;
                                        // traitement des chiffres
      default: error();
      }
   }
}


boolean isLetter(char ch) {
  return ((’a’ <= ch && ch <= ’z’) || (’A’ <= ch && ch <= ’Z’));
}


boolean isDigit(char ch) {
  return (’0’ <= ch && ch <= ’9’);
}
```

Fonctionnement

L'analyseur lexical fournit des lexèmes à l'analyseur syntaxique.

L'action de base est la fonction `void next();` qui est appelée par ce dernier.

En retournant d'un appel, le lexème est stocké dans la variable `int symbol;` Les lexèmes sont typiquement décrits dans un interface `Token`.

Une variable `int pos;` peut contenir la position (dans notre projet cela sera la ligne) du lexème.

```
class Scanner implements Token {
   /** Constructor **/
   Scanner(InputStream in);

   /** lexeme et position **/
   int symbol;
   int pos;

   /** representation du symbole **/
   String chars;

   /** lis lexeme suivant **/
   void next();

   /** fermer le flux d'entree **/
   void close();
}
```

Pour plus de détails, bien regarder le projet.