Université de Strasbourg

Master Informatique

Compilation — Travaux Pratiques n° 1
Analyse lexicale

I. Javanais ou Pig Latin

Le Javanais (en anglais Pig Latin) est un jeu d'enfants où l'on effectue des transformations simples sur les voyelles de mots. On vous demande d'écrire un traducteur Français - Javanais qui fait précéder chaque groupe de voyelles de av. Ainsi, Prise en main serait traduit par Pravisave aven mavain.

II. Comptons les mots !

— Écrire un programme Lex qui copie un fichier en remplaçant chaque suite non vide de blancs par un seul blanc.
— Écrire un programme Lex qui compte le nombre de mots d'un texte puis d'un fichier dont le nom sera passé en paramètre. Votre programme doit afficher la même valeur que la commande Unix wc -w sur le même fichier (/usr/include/stdio.h par exemple).

III. Mon code sur Internet

On veut présenter proprement, sur le Web, des extraits de code C ou C++. Pour cela, il faut produire à partir d'un source C/C++ un fichier HTML respectant le formatage initial (retours ligne, indentation) et mettant en relief mots-clés, instructions du pré-processeur, chaînes de caractères et commentaires. Voici un exemple de programme source :

```
#include <fifo.h>
#define NVOISINS 8
for (x = 0; x < N; x++) {
  if (M[x] != 0)                    /* le pixel appartient a un maximum */
    {
    for (k = 0; k < NVOISINS; k += 1) /* parcourt les voisins */
      {                               /* si un voisin n'est pas dans la FIFO */
      y = voisin(x, k, rs, N);      /* et pas maximum, on le met en FIFO */
      if ((y != -1) && (! IsSet(y, EN_FIFO)) && (M[y] == 0)) {
        FifoPush(FIFO, y);
        Set(y, EN_FIFO);
        if (trace) printf("empile point %d (%d,%d)\n", y, y%rs, y/rs);
        } /* if y ... */
      } /* for k */
    } /* if M */
  } /* for x */
```

Et voici le résultat désiré (probablement tronqué) :

```
<html><body><pre>
<span class="prepro">#include &lt;fifo.h&gt;</span>
<span class="prepro">#define NVOISINS 8</span><span class="kw">for</span> (x = 0; x &lt; N; x++)
  {  <span class="kw">if</span> (M[x] != 0)                    <span class="comment">/* le pixel appartient a un maximum */</
    <span class="kw">for</span> (k = 0; k &lt; NVOISINS; k += 1) <span class="comment">/* parcourt les voisins */</span>
      {                              <span class="comment">/* si un voisin n'est pas dans la FIFO */</span>
      y = voisin(x, k, rs, N);     <span class="comment">/* et pas maximum, on le met en FIFO */</span>
      <span class="kw">if</span> ((y != -1) &amp;&amp; (! IsSet(y, EN_FIFO)) &amp;&amp; (M[y] == 0))
        {
        FifoPush(FIFO, y);
        Set(y, EN_FIFO);
        <span class="kw">if</span> (trace) printf(<span class="str">"empile point %d (%d,%d)\n"</span>, y, y%rs, y/rs);
        } <span class="comment">/* if y ... */</span>
      } <span class="comment">/* for k */</span>
    } <span class="comment">/* if M */</span>
  } <span class="comment">/* for x */</span>
</pre></body></html>
```

où les unités lexicales importantes sont balisées. On remarquera que les caractères "if" dans "FifoPush" ne doivent pas constituer un mot-clé.

Pour obtenir, par exemple, les mots-clès sont en gras, les chaînes de caractères en vert et les commentaires en rouge on utilisera le fichier CSS suivant :

```
.kw { font-weight: bold; }
.str { color: green; }
.comment { color: red; }
```

Voici la liste des mots-clés de C++ :

asm	auto	break	catch	case	char
class	const	continue	default	delete	do
double	else	enum	extern	float	for
friend	goto	if	inline	int	long
new	operator	overload	private	protected	public
register	return	short	signed	sizeof	static
struct	switch	this	template	typedef	union
unsigned	virtual	void	volatile	while	

IV. Table des symboles

Rôle de l'analyseur lexical dans un compilateur

Dans un compilateur, l'analyseur lexical transforme le flux d'entrée de caractères (provenant du fichier qui contient le programme source) en un flux de codes numériques qui représentent les unités lexicales (mots-clés, identificateurs, opérateurs, parenthèses, etc). Il produit également une table des symboles, qui est utile pour retrouver la forme originale des codes représentant les identificateurs (noms de variables, de fonctions...)

Le but de l'exercice est de réaliser, avec l'aide de Lex, un analyseur lexical pour un sous-ensemble du C++, le C-. Voici la description des unités lexicales de C- :

- mots-clés : `cin const cout else if typedef while`
- symboles non-alphanumériques : `+ - * / % ! || && < <= > >= == != « » & | = ( ) [ ] { } , ' ;`
- identificateurs : lettre (lettre | chiffre)*
- identificateurs prédéfinis : `char int float void main`
- constantes entiers : chiffre (chiffre)*

— constantes flottants : comme en C++.
— constantes caractère : 'x' ou '\n'

Ne sont pas des unités lexicales et doivent être ignorés : espace, retour à la ligne, commentaire (entre /* et */).

L'analyseur produit par Lex (la fonction yylex()) devra retourner le code du prochain token lu dans le flux d'entrée. Puisque le reste du compilateur n'est pas développé, on se contentera d'un programme principal de test appelant yylex() en boucle et affichant les résultats produits.

Codage des unités lexicales

Pour réaliser un analyseur lexical, il faut tout d'abord convenir d'un codage des unités lexicales. Celles-ci sont de plusieurs types :

— Les symboles non alphanumériques (par exemple en C++ : + - * / { } [ ] = « » etc).
— Les mots-clés (par exemple en C++ : if, while, for, etc).
— Les identificateurs : ce sont les noms de variables, types, fonctions... utilisés par le programmeur.
— Les valeurs constantes : nous nous limiterons ici aux constantes de type entier (123, 10000, etc), flottant (3.14159, 0.211E-10, etc) et de type caractère ('a', '\n', etc).

Les unités lexicales simples

Les symboles non alphanumériques et les mots-clés peuvent être codés chacun par un entier différent. Par exemple :

symbole	code	symbole	code	symbole	code
+	1	{	23	«	45
-	2	}	24	»	46
*	3	[	25	||	47
/	4	]	26	&&	48
if	100	for	112	while	124

Lorsque l'analyseur lexical reconnaît dans le flux d'entrée un de ces symboles, il émet dans le flux de sortie (ici : l'écran) le code correspondant.

Les valeurs constantes

Dans le flux de sortie de l'analyseur lexical, il faudra émettre dans ces cas :

— un code numérique signifiant : soit constante entiere, soit constante flottante, soit constante caractère
— la valeur lexicale de cette constante.

Les identificateurs

On ne les connaît pas à l'avance. Il faut donc les enregistrer au fur et à mesure de leur apparition dans le programme source, et les stocker dans une table des symboles. La table

des symboles établit la correspondance entre un symbole identificateur (nom de variable ou de fonction, par exemple) et un entier (l'index du symbole dans la table). Certains identificateurs (comme main, char, int, float) sont prédéfinis : ils font partie du langage et sont chargés dans la table des symboles lors de l'initialisation de l'analyseur lexical.

Dans le flux de sortie de l'analyseur lexical, il faudra émettre dans ce cas :

— un code numérique signifiant : identificateur

— un entier correspondant à l'index de cet identificateur dans la table des symboles.

Important : L'apparition du même symbole plusieurs fois dans le programme source devra toujours donner lieu à l'émission dans le flux de sortie du même index.

La table des symboles

La table des symboles établit la correspondance entre un symbole identificateur (nom de variable ou de fonction, par exemple) et un entier (index du symbole dans la table). La structure de données la plus simple pour gérer cette table est un tableau de chaînes de caractères. Pour l'extrait de programme :

```
void segmente()
{
  int mesure;
  int increment = 1;
  int maximise = 0;

  mesure = 0;
...
```

on obtient la table des symboles suivante :

index	chaine
0	segmente
1	mesure
2	increment
3	maximise

Il sera bien entendu nécessaire de disposer d'une fonction de recherche d'une chaîne de caractères dans cette table, ainsi que d'une procédure d'ajout d'une chaîne de caractères dans la table. Vous pourrez utiliser une représentation de type tableau dans votre TP. Cependant, il existe des structures mieux adaptées à ce problème, notamment en ce qui concerne la complexité de l'opération de recherche d'une chaîne de caractères. C'est le cas notamment des tables de hachage, parfois appelées aussi tables à adressage dispersé, utilisées dans la plupart des compilateurs.