

Compilation

II. Analyse lexicale

Jacques Farré

Jacques.Farre@unice.fr

<http://deptinfo.unice.fr/~jf/Compil-L3/>

Introduction

- cette partie du cours a pour pré-requis le cours *Automates et langages* du premier semestre
 - on suppose connues les notions de langage régulier, d'expression régulière, d'automate fini, et l'utilisation de *flex* (pour *flex*, voir <http://flex.sourceforge.net/manual/>)
- on se concentrera sur les problèmes pratiques de l'analyse lexicale
 - extensions nécessaires aux expressions régulières pour traiter des cas réels
 - reconnaissance avec lecture en avant, reconnaissance contextuelle
 - réduction de l'automate produit par les expressions régulières
 - écriture « à la main » de l'analyseur lexical
 - relation avec l'analyseur syntaxique
- vocabulaire : par la suite, on emploiera indifféremment les termes *lexème*, *terminal* ou *token*

Utilité d'avoir un analyseur lexical

- le texte source comprend du « bruit » comme les espaces, les fins de ligne, les commentaires . . . qui souvent ne sont pas grammaticalement pertinents (et qu'il est même quasiment impossible d'inclure dans la grammaire)
- certaines formes sources sont grammaticalement équivalentes (les différents identificateurs, les nombres littéraux . . .), ou correspondent à un même symbole pour la grammaire (\neq , $!=$)
- l'analyse lexicale a pour rôle de traiter ces problèmes et donc de simplifier l'écriture de l'analyseur syntaxique

Analyseur lexical et analyseur syntaxique

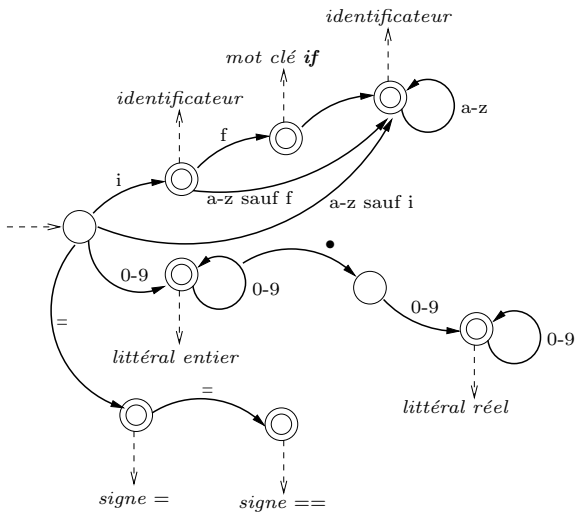
- l'analyse lexicale *produit* des lexèmes, en général au vol à la demande de l'analyseur syntaxique, qui les *consomme*
 - donc l'analyseur demande le prochain lexème à l'analyseur lexical, qui le reconnaît et retourne un *code* identifiant ce lexème
- on pourrait imaginer que l'analyse lexicale travaille indépendamment et fournisse une suite de lexèmes
 - mais on n'a pas besoin, en général, d'avoir reconnu tous les lexèmes pour faire l'analyse syntaxique : la connaissance du lexème courant suffit pour la plupart des méthodes utilisées
 - et malheureusement, pour certains langages (C notamment), la reconnaissance d'un lexème dépend du contexte grammatical dans lequel il se trouve

Langage lexical - I

- comme vu précédemment, l'analyseur lexical est chargé de reconnaître les lexèmes du texte source
- chacun de ces lexèmes est un langage
 - dans la plupart des cas, c'est un langage régulier, par exemple
 $[A-Za-z][A-Za-z0-9]^*$
pour reconnaître des identificateurs du genre X, leCompteur, i3, R2D2 ...
 - c'est un langage fini pour de nombreux lexèmes : +, ==, **for**
- l'analyseur lexical est donc un (souvent gros) automate fini déterministe qui reconnaît l'union des langages des lexèmes
 - un état terminal correspond à la reconnaissance d'un mot d'un des langages des lexèmes

Langage lexical - II

Fragment d'automate lexical



Langage lexical - III

- l'analyseur lexical n'est en général pas tout à fait un automate déterministe
- certains des langages des lexèmes sont préfixes d'un autre : par exemple +, +=, ++, ou encore mots-clés et identificateurs (for et forme), entiers et réels (3 et 3.14)
 - le principe est de reconnaître le mot le plus long
 - un état terminal n'implique donc pas toujours l'arrêt de l'automate : il peut continuer, et éventuellement revenir en arrière au dernier état final rencontré
 - et si un mot appartient à 2 langages ?
 - puisqu'un identificateur est en général une suite de lettres, for est aussi un identificateur
 - il faut utiliser un règle pour choisir : par exemple en flex, c'est la première définition qui compte

Fragment de description d'un analyseur lexical

- fragment d'analyseur lexical en *flex* :
on suppose que l'analyseur syntaxique définit ses lexèmes (des entiers!) par des constantes comme #define IDENT 257

```
[ \t\n]           { /* sauter les espaces */ }
"for"            { /* le mot-cle for */
                  return KFOR; }
[A-Za-z][A-Za-z0-9]* { /* c'est un ident */
                  return IDENT; }
[0-9]+          { /* litteral entier */
                  return ENTIER; }
"+="            { /* symbole compose */
                  return PLUS_EGAL; }
.                { /* symbole simple */
                  return ytext[0]; }
```

- si la reconnaissance de "for" avait été écrite après la reconnaissance des identificateurs, il aurait été reconnu comme identificateur et pas comme mot-clé

Langage lexical non régulier

- il peut exister des (rares) constructions lexicales qui sont des langages algébriques (c'est-à-dire non contextuels)
 - par exemple un langage qui accepterait des commentaires emboîtés : `/* aaa /* bbb */ ccc */`
 - il est inutile de mettre en œuvre un automate à pile pour reconnaître ce genre de construction
 - une fonction qui lit le texte jusqu'à la fin du commentaire fera l'affaire ; par exemple en *flex*

```
%{  
int niveau = 0; /* niveau d'emboitement du commentaire */  
  
void skip_comm();  
%}  
  
%%  
"/*"      { niveau++; skip_comm(); }  
.         { ECHO; } /* on affiche ce qui n'est */  
          /* pas dans un commentaire */
```

Langage lexical non régulier

fonction de reconnaissance des commentaires emboîtés

- *input* est une macro de *flex* qui lit le prochain caractère

```
void skip_comm() { /* version simpliste d'une fonction */
    for (;;) {      /* pour sauter les commentaires */
        int c = input();
        if (c == '*') {
            if (input() == '/') {
                niveau--;
                if (niveau == 0) return;
            }
        }
        else if (c == '/') {
            if (input() == '*') {
                niveau++;
            }
        }
    }
}
```

Langage lexical non régulier

reconnaissance conditionnelle et commentaires emboîtés

- on peut aussi, en *flex*, utiliser des *start conditions*

```
%{  
int niveau = 0;  
%}  
%s COMM      /* declaration d'une condition      */  
              /* (celle par default est INITIAL) */  
  
%%  
"/*"        { if (niveau == 0) BEGIN(COMM); niveau++; }  
              /* reconnu dans toutes les conditions */  
"*/"        { niveau--; if (niveau == 0) BEGIN(INITIAL); }  
              /* reconnu dans toutes les conditions */  
<COMM>.     { } /* on n'affiche rien si dans un commentaire */  
.           { ECHO; } /* on affiche ce qui est lu sinon */ }
```

- on peut déclarer des conditions exclusives (%x)

Reconnaissance contextuelle

- il existe des langages où une forme lexicale peut correspondre à différents lexèmes selon le contexte

- selon le contexte gauche : par exemple, **if** reconnu comme mot-clé uniquement s'il est au début d'une instruction; une *start condition* pourra être utilisée :

```
<DEBINST>if { /* traitement comme mot-cle */ }  
[a-z]+      { /* traitement avec les identificateurs */ }
```

c'est probablement l'analyseur syntaxique qui demandera à l'analyseur lexical de changer de condition

- selon le contexte droit : par exemple **if** reconnu comme mot-clé uniquement s'il est suivi d'une parenthèse; un *trailing context* pourra être utilisé (e_1/e_2 ne reconnaît e_1 que si elle suivie de e_2) :

```
if/[ \t\n]*"(" { /* traitement comme mot-cle si suivi de ( */  
[a-z]+      { /* traitement avec les identificateurs */ }
```

- les 2 peuvent être combinés
- voir le manuel de *flex* pour la reconnaissance selon la position (début de ligne, fin de ligne ...)

Problèmes de la reconnaissance contextuelle

- on les rencontre dans des langages mal définis, et pour les langages de programmation, dans des langages anciens (Fortran, PL/1, partiellement C ...)
- ils ne peuvent en général pas être résolus par le seul analyseur lexical : les contextes sont le plus souvent des langages non-contextuels (!), reconnus par l'analyseur syntaxique
- donc forte interaction entre les 2 analyseurs, au mépris de la lisibilité, de la fiabilité et des facilités de maintenance et d'extension
- c'est pourquoi les langages modernes de programmation évitent ces questions de reconnaissance contextuelle, mais malheureusement pas les langages de script, de description ... conçus la plupart du temps par des non spécialistes des langages

Exemple de problème de la reconnaissance contextuelle

Le fameux bogue d'une mission Mercury!

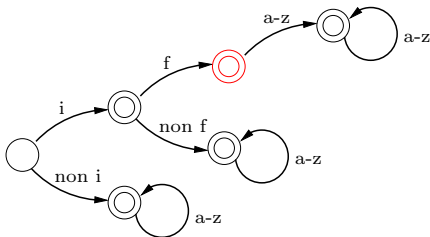
- en *Fortran*, espaces non significatifs, et déclaration implicite des identificateurs débutant par *i*, *j* ... *n* comme variables entières, les autres comme variables réelles
 - par conséquent `D012I=1.25` est interprété comme *identificateur D012I, opérateur d'affectation =, réel 1.25*
 - mais `D012I=1,25` est interprété comme *mot-clé DO, étiquette 12, variable I*: boucle qui se termine à l'énoncé étiqueté 12 pour *i* allant de 1 à 25
 - d'après la légende, on aurait perdu une fusée du projet Mercury à cause d'un '.' à la place de la ',' (à moins que ce ne soit l'inverse)...
- dans la plupart des langages de programmation actuels, les espaces sont significatifs, et les mots-clés sont réservés
 - ayant la syntaxe lexicale des identificateurs, l'inconvénient est que plus rien ne les distingue extérieurement, ce qui n'aide pas à la réparation d'erreurs

Performances de l'analyseur lexical

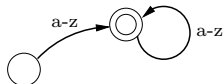
- en règle générale, l'analyseur lexical est linéaire en temps sur le nombre de caractères du texte d'entrée
 - même s'il doit parfois reculer, c'est d'un nombre borné de fois (voir l'exemple avec +, +=, ++)
- mais on pourrait imaginer un langage très mal conçu où le temps serait quadratique
 - supposons qu'il n'y ait que 2 lexèmes : a et a . . . a b ; sur l'entrée a . . . a, il doit aller, pour reconnaître **chaque** a jusqu'à la fin du texte, soit $n - 1 + n - 2 + \dots + 1 = \mathcal{O}(n^2)$ lectures
- autre facteur de performance, la taille de l'automate : elle peut poser des problèmes de cache mémoire qui ralentissent fortement l'exécution
 - si l'automate est implémenté sous forme de table, il existe des techniques de compression, mais au prix d'un accès plus coûteux aux éléments de la table

Réduction de la taille de l'automate

- la discrimination par l'automate entre les identificateurs et les mots-clés multiplie le nombre d'états



automate de discrimination entre if et les identificateurs



automate de reconnaissance des identificateurs

- on a donc intérêt à ne faire reconnaître par l'automate que les identificateurs, puis regarder dans une table si l'identificateur reconnu correspond à un mot-clé
 - la recherche dans cette table peut être rapide (par dichotomie, par adressage dispersé ...)

Faciliter le travail du reste du compilateur

- si l'analyse syntaxique peut traiter les identificateurs sans les distinguer, l'analyse sémantique ou le traitement des erreurs ont besoin de savoir de quel identificateur il s'agit (idem pour les littéraux)
- l'analyseur lexical devra donc fournir, en plus de la classe syntaxique du lexème, sa forme source (ou sa valeur pour un littéral)
- pour éviter de multiplier le nombre de chaînes de caractères identiques transmises aux autres analyseurs, l'analyse lexicale peut garder les identificateurs dans une table, et retourner l'entrée de cette table correspondant à l'identificateur reconnu
 - du point de vue performances, c'est la différence entre == et equal sur des chaînes (Java) ou strcmp (C)
- idem pour les chaînes de caractères qu'il peut être intéressant de ranger dans des tables

Traitement des identificateurs et des mots-clés

- spécification d'un module de traitement des identificateurs

```
typedef struct InfoIdent {  
    const char* texte;  
    int code_mot_cle;  
} InfoIdent
```

```
InfoIdent table[MAX];
```

```
/* une table de taille dynamique serait mieux */
```

```
InfoIdent* chercher (const char* s);
```

```
/* Retourne l'entree de la table correspondant a l'identificateur  
   (code_mot_cle = 0) ou au mot-cle ecrit comme s.
```

```
   Ajoute l'identificateur s a la table s'il n'y est pas deja.
```

```
   Initialement, la table ne contient que les mots-cles du  
   langage et eventuellement des identificateurs predefinis
```

```
*/
```

ce qui suppose que le vocabulaire de l'analyseur syntaxique est un sous-ensemble des entiers, et pas des caractères !

Relations analyseur lexical - analyseur syntaxique

- comme vu plus haut, en général, l'analyseur syntaxique demande les lexèmes au fur et à mesure de ses besoins
- l'analyseur lexical renvoie la classe de l'identificateur, et d'autres informations utiles
 - numéro de ligne, et éventuellement position dans la ligne et nom du fichier
 - le lexème lui-même, ou son entrée dans une table
 - des informations sémantiques, comme la valeur d'un nombre, d'une chaîne
- par exemple en C

```
struct LexInfo {  
    int ligne;  
    union {  
        int ival;  
        double rval;  
        const char* sval;  
    } val;  
} lexInfo;
```

Fragment de description d'un analyseur lexical

- on suppose que l'analyseur syntaxique définit ses lexèmes (des entiers!) par des constantes comme #define IDENT 257
- fragment d'analyseur lexical en *flex* :

```
[A-Za-z][A-Za-z0-9]* { /* ident ou mot-cle */  
                        InfoIdent* i = chercher(yytext);  
                        lexInfo.ligne = yylineno;  
                        if (i->code > 0) return i->code;  
                        lexInfo.sval = i->texte;  
                        return IDENT; }
```

```
[0-9]+ { /* litteral entier */  
        lexInfo.ligne = yylineno;  
        lexInfo.ival = atoi(yytext);  
        return ENTIER; }
```

Analyse lexicale avec *JavaCC*

<https://javacc.dev.java.net/>

- *JavaCC* (*Java* Compiler Compiler) est (notamment) un générateur d'analyseur lexical et syntaxique produisant du *Java*
 - notation (presque) uniforme pour les aspects lexicaux et les aspects syntaxiques : les expressions régulières
 - *JavaCC* (initialement *Jack*) est inspiré de *PCCTS*, qui a lui-même donné *ANTLR* (<http://www.antlr.org/>)
- un fichier de description pour *JavaCC* est, par convention, suffixé par `.jj`
- *JavaCC* est appelé par la commande :
`javacc options fichier.jj`
- le résultat est un ensemble de classes *Java*

Structure d'un fichier *JavaCC*

- le fichier doit avoir la structure suivante :

```
options /* optionnelle */
PARSER_BEGIN ( nom de l'analyseur )
classe java /* avec une methode main */
PARSER_END ( nom de l'analyseur )
regles lexico-syntaxiques
```

- par exemple

```
options {
IGNORE_CASE=true;
}
```

```
PARSER_BEGIN(exemple)
public class exemple { ... }
PARSER_END(exemple)
... // regles
```

- le fichier à faire compiler par *Java* sera *exemple.java*

L'analyseur *JavaCC*

- il est lancé par la méthode *main* de la classe *Java*
- par exemple, si on veut juste regarder les lexèmes reconnus

```
public class exemple {  
    public static void main(String args[]) {  
        exemple parser = new exemple(System.in);  
        Token tok = parser.getNextToken();  
        while (tok.kind != EOF) {  
            System.out.println (token.image);  
            tok = parser.getNextToken();  
        }  
    }  
}
```

- la classe est une classe *Java* à part entière, qui peut avoir des attributs, des méthodes ...
 - *JavaCC* s'empresse de lui en ajouter, regardez le code *Java* de la classe produite

Règles lexicales en *JavaCC*

Définition des lexèmes

- les lexèmes sont décrits par des expressions régulières
 - ces expressions régulières peuvent être nommées
 - un nom précédé de # est une ER auxillaire qui participe à la définition d'un lexème, mais n'en est pas un
- toujours pour l'exemple :

```
TOKEN : /* ce qu'on reconnaît */
{
    < CONSTANT: ( <DIGIT> )+ >
    | < #DIGIT: ["0" - "9"] >
```

```
TOKEN : /* ce qu'on reconnaît */
{
    < IDENT: ( <LETTER> )+ >
    | < #LETTER: ["a" - "z"] >
        /* comme l'option IGNORE_CASE est vraie
           on reconnaîtra aussi bien abc que ABC ou AbC */
}
```


Règles lexicales en *JavaCC*

Parties du texte à ignorer

- la catégorie la plus courante est SKIP : les portions de texte conformes à ces définitions sont ignorées

```
SKIP : { /* on saute les espaces */  
        " " | "\n" | "\t"  
    }
```

```
SKIP : { /* et les commentaires a la C */  
        < "/*" (~["*"])* "*" (~["/"] (~["*"])* "**")* "/" >  
    }
```

- la catégorie SPECIAL_TOKEN est aussi ignorée, mais accessible par l'attribut specialToken du lexème suivant
 - permet par exemple de raccrocher un commentaire `/** ... */` pour une application comme *JavaDoc*

Écriture à la main de l'analyseur lexical

- l'écriture directe de l'analyseur lexical peut notablement accroître les performances du compilateur
 - travail plus fastidieux que difficile : il s'agit simplement de câbler en dur un automate fini
 - permet aussi des « ruses » que ne peut pas toujours permettre un générateur d'analyseurs
- il faut bien sûr éviter de construire un automate non déterministe, car personne ne viendra le déterminer (c'est la faiblesse de l'approche)
 - par exemple, ne pas programmer l'automate non déterministe sous-jacent à $+|+=$, mais celui (déterministe) correspondant à $+ =?$ (reconnaissance du +, suivi éventuellement d'un =)

Quelques principes d'écriture manuelle de l'analyseur lexical

- on suppose que `car` est le caractère suivant la portion de texte reconnue, et `carsuiv()` la fonction qui lit le prochain caractère du texte d'entrée
- quelques schémas de codage des expressions régulières
 - `a` : `if (car == 'a') car = carsuiv(); else erreur();`
 - `a e1 | b e2` :

```
if (car == 'a') {
    car = carsuiv(); // + code pour analyser e1
} else if (car == 'b') {
    car = carsuiv(); // + code pour analyser e2
} else erreur();!
```
 - `a (b e1)*` :

```
if (car == 'a') {
    car = carsuiv();
    while (car == 'b') { car = carsuiv(); // + code pour
                          analyser e2
    }
} else erreur();
```

Fragments d'un analyseur lexical manuel

On suppose qu'un tableau `int classe[256]` donne la classe de chaque caractère (par exemple `classe['a']=LETTRE` et `classe['+']='+'`)

```
int lexSuiv() {
    InfoIdent* i; int cla;
    for (;;) {
        switch (classe[car]) {
            case ESPACE : car = carsuiv(); break;
            case LETTRE : do { /* ident ou mot-cle */
                            /* memoriser car dans une chaine str */
                            car = carsuiv(); cla = classe[car];
                        } while (cla == LETTRE || cla == CHIFFRE);
                        i = chercher(str); return i->code;
            case '+' : car = carsuiv();
                      if (car == '=') { car = carsuiv(); return PLUSEGAL; }
                      else return '+';
            case "/" : /* distinguer entre le simple / ==> return '/'
                       et le debut de commentaire ==> break */
                      ...
        }
    }
}
```