Compilation

Analyse Lexicale

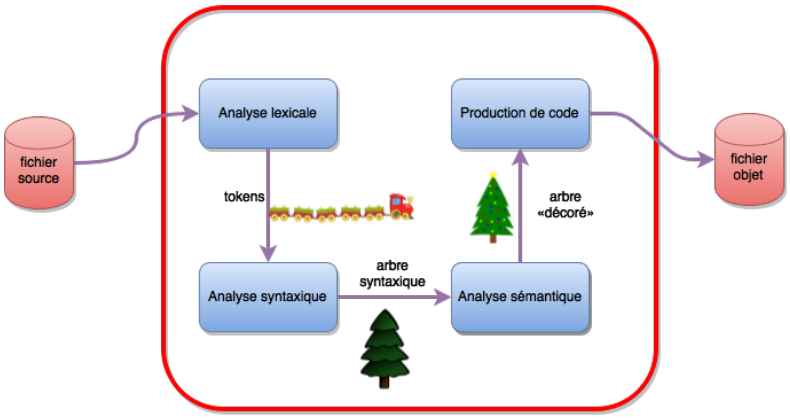
Lex: un générateur d'analyseurs lexicaux

SI4 — 2018-2019

Erick Gallesio

Fonctionnement d'un compilateur

Principe de fonctionnement:



Partie avant:

- Analyseur lexical fournit des «tokens»
- Analyseur syntaxique construit un arbre
- Analyseur sémantique décore l'arbre

Partie arrière:

- Facultatif: optimisation (non représentée ici)
- Production de code parcourt l'arbre décoré

Analyse lexicale

L'analyse lexicale consiste à

- découper le texte en *lexèmes* (ou *tokens*)
- éliminer (éventuellement) certaines unités inutiles (espaces, commentaires)

Dans la phrase en français:

```
La ligne comporte des mots
```

nous avons 5 lexèmes: "La", "ligne", "comporte", "des" et "mots".

Dans le programme C suivant,

```
if (a1 == a2) a1 = a2++ + 12;
```

nous pouvons reconnaître les lexèmes suivants (dans le désordre)

- le mot réservé "if"
- les identificateurs "a1" et "a2"
- la constante entère 12
- les symboles "(", ")" et ";"
- les opérateurs "=" et "+"
- l'opérateur "==" (qui n'est pas vu comme deux "=" successifs)
- l'opérateur "++" (qui n'est pas vu comme deux "+" successifs)

Utilité de l'analyse lexicale

- Les programmes sources contiennent beaucoup d'informations inutiles à l'analyse
  - *commentaires*
  - *indentations*
  - *certains espaces non nécessaires* `x = y ++ + 3` *est équivalent à* `x=y+++3`
  - *pour certains langages la casse n'est pas significative ("foo" ⇔ "FoO")*
- La classe des grammaires utilisées pour l'analyse lexicale est généralement plus simple que celles nécessaires pour l'analyse syntaxique (langages rationnels reconnaissables par des automates finis)

L'analyse lexicale va donc **permettre de simplifier** l'écriture de l'**analyse syntaxique**.

Lexèmes et expression régulières

- Les lexèmes (*tokens*) des langages de programmation sont souvent représentés par des expressions régulières.
- L'ensemble des langages qui peuvent être exprimés sous la forme d'expressions régulières est appelé l'ensemble des *langages réguliers* (ou *rationnels*).
- Un langage régulier peut être reconnu par un automate fini.
- On peut facilement produire automatiquement un analyseur reconnaissant un langage régulier à partir d'une expression régulière.

Expression régulière → DFA (1/4)

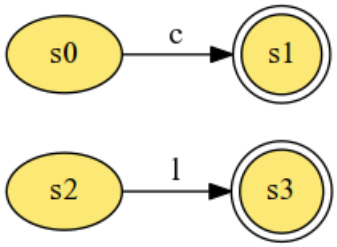
Considérons l'expression régulière suivante

```
l(l|c)*
```

Cette expression régulière est une version simplifiée de l'expression régulière permettant de reconnaître les identificateurs des langages de programmation (une lettre suivie d'une suite de lettres ou de chiffres).

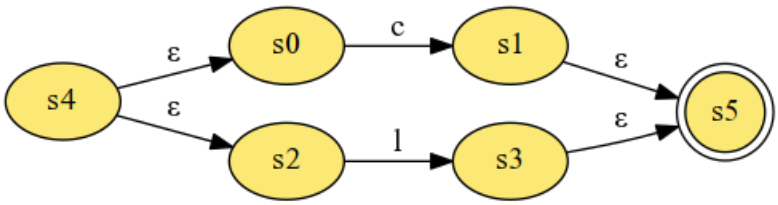
Comment construire l'automate non déterministe à partir de cette expression régulière?

- Pour reconnaître `(l | c)`, nous devons tout d'abord reconnaître les langages `l` et `c`.

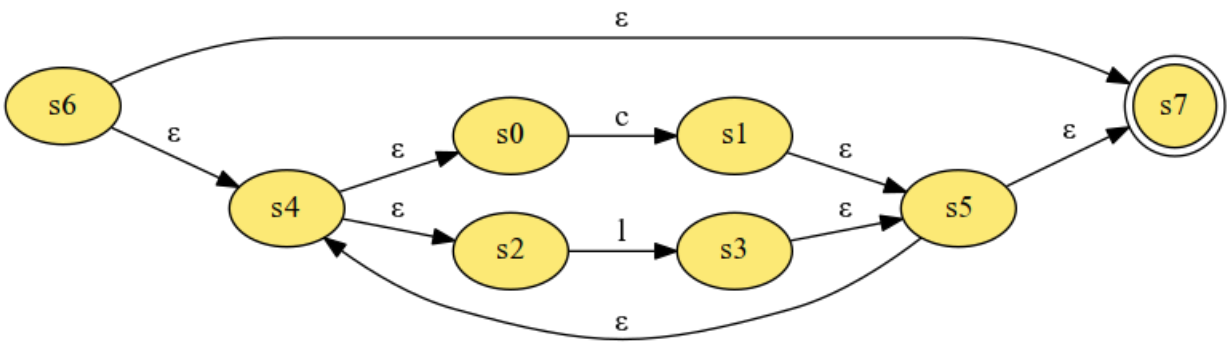


Expression régulière → DFA (2/4)

- Construisons maintenant le NFA pour `(l | c)` :

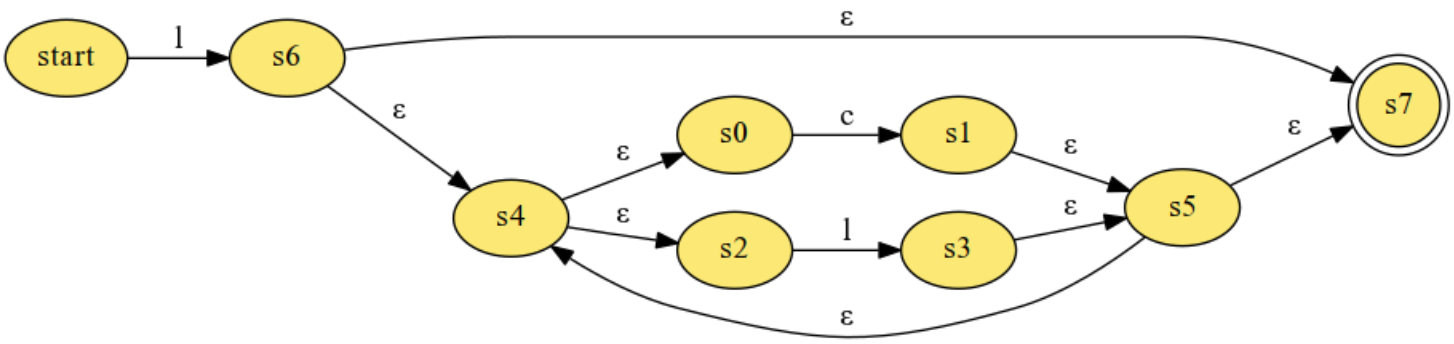


- Puis celui de `(l | c)*` :



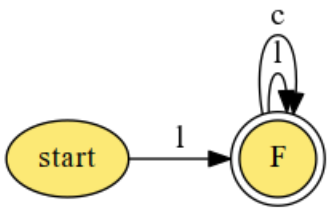
Expression régulière → DFA (3/4)

- On peut maintenant ajouter `l` devant `(l | c)*`:



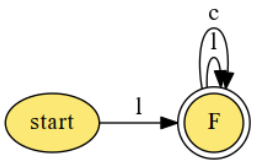
Expression régulière → DFA (4/4)

Après *déterminisation* et *minimisation* de cet automate non déterministe (cf début du cours LFA), on obtient:



Cet automate est donc **l'automate fini déterministe** correspondant à **l'expression régulière** `l(l |c)*`

Automate représenté par une table



peut être représenté en mémoire par la table suivante:

	l	c	autres
start	F	#	#
F	F	F	#

Si on tombe sur ' # ' $\Longrightarrow$ erreur

Analyse lexicale à partir d'une table

En partant de la table:

	l	c	autres
start	F	#	#
F	F	F	#

On peut construire un **analyseur lexical** avec l'algorithme suivant

```
state ← start
c ← nextchar();
while (c ≠ EOF) {
    state ← T[state, c];
    if (state is "#") reject;
    c = nextchar();
}
if (state ∈ Final)
    accept;
else
    reject;
```

Bilan:

- à partir de `l (l | c)*` → NFA → DFA → table d'analyse.
- On peut donc construire automatiquement un analyseur lexical à partir d'une (ou plusieurs) expression(s) régulière(s).

Analyseur lexical

L'analyseur lexical d'un langage de programmation:

- est un automate fini déterministe (DFA)
- il reconnaît l'union des langages des lexèmes
- un état terminal de cet automate correspond à la reconnaissance d'un mot d'un des langages des lexèmes
- quand un lexème peut être le préfixe d'un autre, il reconnaît le **lexème le plus long** (*e.g.* '--', '-=', '-')
- certaines classes de lexèmes nécessitent la construction d'un **attribut**.

Classes de lexèmes (1/2)

- Un lexème reconnu par l'analyseur appartient à une *classe* de lexèmes.
- La classe du lexème est la **seule chose utile** pour l'analyse syntaxique.
- Classes de lexèmes courantes:
  - *constantes numériques (entiers, flottants, …)*
  - *constantes chaînes*
    - identificateurs
  - *chaque mot clé du langages (' `if` ', ' `else` ',..)*
  - *séparateurs (' `,` ', ' `;` ', ' `(` ', …)*

Classes de lexèmes (2/2)

Soit le code suivant:

```
if (a <= 0) {  /* On a un problème */
   printf("Ouch: a = %d\n", a);
}
```

Les lexèmes reconnus ici sont, dans l'ordre:

```
KIF LPAREN [ident] LE [integer] RPAREN LBRACE
[ident] LPAREN [str] COMMA [ident] RPAREN SEMICOLON
RBRACE
```

- Les commentaires sont éliminés
- Chaque mot clé et chaque séparateur constitue sa propre classe de lexèmes
- Tous les identificateurs sont dans la classe `[ident]`
- De même, toutes les chaînes (resp. entiers) sont dans la classe `[str]` (resp. `[integer]`)

Attributs de lexèmes

- Les classes de lexèmes suffisent pour l'analyse syntaxique
- Pour l'analyse sémantique, par contre, on a besoin de connaître la valeurs des lexèmes.
- L'analyseur ajoute donc un **attribut** à certain lexèmes:

```
KIF LPAREN [ident, "a"] LE [integer, 0] RPAREN LBRACE
[ident, "printf"] LPAREN [str, "Ouch ..."] COMMA [ident, "a"] RPAREN SEMICOLON
RBRACE
```

Le type de l'attribut dépend du lexème:

- chaîne pour les identificateurs ou les chaînes,
- entier pour les constantes entières
- float (ou double) pour les constantes réelles
- …

Générateur d'analyseur lexical

Les générateurs d'analyseurs lexicaux utilisent les *expressions rationnelles* pour définir les lexèmes du langage.

- *mots clés*: '`if`', '`else`', '`while`', ...
- *identificateurs*: `[a-zA-Z_][a-zA-Z_0-9]*`
- *entiers*: `([0-9]+)|(0x[0-9A-Fa-f]+)`
- *chaînes*: `\"[^"]*\"`
- *opérateurs*: '`-`', '`--`', '`-=`', ...
- *symboles*: '`;`', '`(`', '`)`', ...

Ils permettent aussi de construire et passer des attributs aux phases suivantes.

Lex

Un générateur d’analyseurs lexicaux

Lex / Flex

Lex est un générateur d'analyseurs lexicaux.

- générateur *standard* Unix
- version originale définie en 1975 par Mike Lesk et Eric Schmidt
- prend en entrée la spécification de l'analyseur lexical
- produit un programme qui implemente l'analyseur en C

Flex est une version alternative de *lex*

- capable de produire du C et du C++
- développé à partir de 1987 et encore (un peu) développé

Autres langages de programmation

La plupart des langages de programmation disposent de quelque chose équivalent à *lex*:

- PLY en Python
- Golex en Go
- Jison en Javascript (lex + syntaxe)
- Silex en Scheme
- …

Structure d'un fichier lex

Un fichier source est divisé en trois sections.

```
<définitions>

%%
<règles>

%%
<code>
```

Définitions:

Cette partie contient

- des macros pour *lex* et
- éventuellement du code C qui est recopié tel quel **au début** l'analyseur produit.

Règles:

association *regexp* → code C.

Code:

permet de mettre du code C qui recopié tel quel **à la fin** de l'analyseur produit.

Exemple 1 (1/3)

```
lettres  [A-Za-z]
chiffres [0-9]
%option noyywrap

%{ /* Code C */
   int cpt_mots = 0, cpt_nombres = 0;
%}

%%       /* début des régles */

{lettres}+           { cpt_mots += 1; }
{chiffres}+          { cpt_nombres += 1; }

.|\n                 { /* Ne rien faire */ }

%%      /* début de la section code */
int main() {
    yylex();
    printf("Mots: %d Nombres: %d\n", cpt_mots, cpt_nombres);
    return 0;
}
```

Exemple 1 (2/3)

Partie définitions

```
lettres  [A-Za-z]
chiffres [0-9]
%option noyywrap
%{int cpt_mots = 0, cpt_nombres = 0; %}
```

- `lettres` et `chiffres` sont des **définitions**
- utilisation de l'option `noyywrap` qui indique que l'on a pas besoin de la fonction `yywrap` (fonction appelée à la fin de l'analyse).
- ajout de code C entre les balises '`%{`' et '`%}`'

Partie règles:

```
{lettres}+          { cpt_mots += 1; }
{chiffres}+         { cpt_nombres += 1; }
.|\n                { /* Ne rien faire */ }
```

- 3 règles ici avec leur code associé

Partie code:

- définition de la fonction `main`
- appel de la fonction `yylex` (fonction principale de l'analyseur produit)

Exemple 1 (3/3)

Construction de l'analyseur

```
$ lex -o simple_lex.c simple_lex.l
$ gcc -o simple simple.c
```

1. Appeler lex (ou flex) pour passer de *simple_lex.l* → *simple_lex.c*
   (l'option `-o` permet de spécifier le nom du fichier de sortie, sinon c'est `lex.yy.c`)
2. Appeler le compilateur C sur l'analyseur produit par lex

Exemple 2 (1/3)

Dans les actions de l'analyseur:

- **yytext** contient le texte du lexème qui vient d'être reconnu
- **yyleng** contient la longueur de ce lexème.
- **yyin** représente le fichier d'entrée (*stdin* par défaut)

```
%{
  #include <math.h>  /* pour atof() */
%}

DIGIT    [0-9]
ID       [a-z][a-z0-9]*

%%
{DIGIT}+             printf("An integer: %s (%d)\en", yytext, atoi(yytext));
{DIGIT}+"."{DIGIT}*  printf("A float: %s (%g)\en", yytext, atof(yytext));
```

... / ...

Exemple 2 (2/3)

... / ...

```
if|then|begin|end|function      printf("A keyword: %s\n", yytext);

{ID}                            printf("An identifier: %s\n", yytext);

"+"|"-"|"*"|"/"                 printf("An operator: %s\en", yytext);

[ \t\n]+                        /* Eat up white space. */

.                               printf("Unrecognized character: %s\n", yytext);

%%

int main(int argc, char *argv[]) {
    ++argv, --argc;  /* Skip over program name. */
    if (argc > 0)
        yyin = fopen(argv[0], "r");

    return yylex();
}
```

Exemple 2 (3/3)

L'ordre des règles est important:

- si deux règles peuvent reconnaître un même lexème, celle qui est déclarée en premier est appliquée.

Ainsi,

```
if        printf("A if keyword\n", yytext);
{ID}      printf("An identifier: %s\n", yytext);
```

- permet bien de reconnaître `if` comme un mot-clé (même si la règle `{ID}` marcherait aussi)
- si on inverse ces deux règles, `if` est vu comme un identificateur standard.

Notion de contexte gauche

Dans certains cas, il est plus simple de décrire une règle en limitant la règle à un certain contexte.

Supposons que l'on veuille supprimer des commentaires C et recopier tout le reste.

- Inutile de construire la chaîne **yytext**
- on recopie tous les caractères si on est en dehors d'un commentaire
- on ne recopie pas les caractères si on est dans un commentaire.

Au départ, on est dans le contexte pré-défini `INITIAL`

```
%x IN_COMMENT /* déclaration d'un nouveau contexte */

%%
"/*"                    BEGIN IN_COMMENT;
<IN_COMMENT>"*/"        BEGIN INITIAL;
<IN_COMMENT>.|\n        /* rien */;
.|\n                    ECHO;
%%
```

Remarque:

BEGIN permet de changer de contexte gauche

Propriétés de l'analyseur produit (1/2)

L'analyseur lexical produit par *lex*:

- linéaire en temps sur le nombre de caractères du texte d'entrée.
- sa taille peut parfois être source de problème (si embarqué par exemple).
  - *une source d'optimisation consiste à reconnaître les mots-clés comme des identificateurs et à discriminer ensuite.*
  - *comme les mots-clés sont connus à l'avance, on les range dans une table*

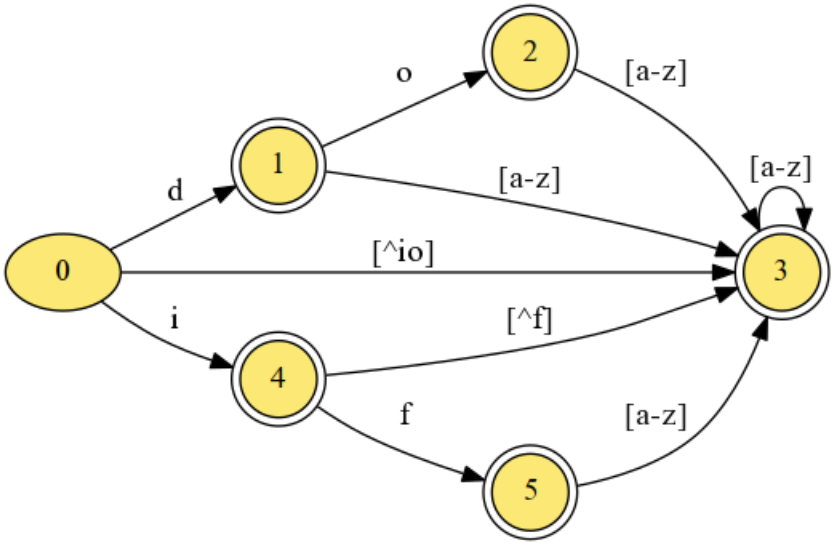
```
if yytext ∈ tokens
    return tokens[yytext]
else
    return <ident, yytext>
```

Propriétés de l'analyseur produit (2/3)

Si on a

```
"do"        printf("Keyword: do\n");
"if"        printf("Keyword: if\n");
[a-z]+      printf("Ident: %s\n", yytext);
```

L'automate est (avec seulement 2 mots-clés)



Propriétés de l'analyseur produit (3/3)

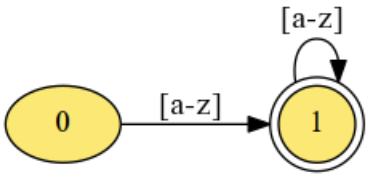
Par contre, le code

```
char* key[] = { "if", "do", "while", ...}

char* find_keyword(char *id) {
    ....
}

%%
[a-z]+      { char *k = find_keyword(yytext);
              if (k)
                 printf("Keyword: %s\n", k);
              else
                 printf("Ident: %s\n", yytext);
            }
```

permet d'obtenir un automate beaucoup plus simple:



La table permettant d'implementer cet automate est donc plus petite.

La fonction yylex()

L'utilisation faite jusqu'à présent de **yylex()** ne correspond pas à son utilisation "classique".

- dans nos exemples, elle ne se termine que lorsqu'on a parcouru tout le fichier
- dans un compilateur, elle permet de renvoyer le prochain lexème du fichier d'entrée (utilisation de **return**).

Généralement, le code *lex* ressemble plutôt à:

```
%%
"if"                    return KIF;
[a-z][a-z0-9]*          { curid = strdup(yytex); return KIDENT; }
"("                     return KLPAREN;
```

La fonction *yylex()* est donc appelée par l'analyseur syntaxique lorsqu'il a besoin de la prochaine unité lexicale.