



# Analyse lexicale

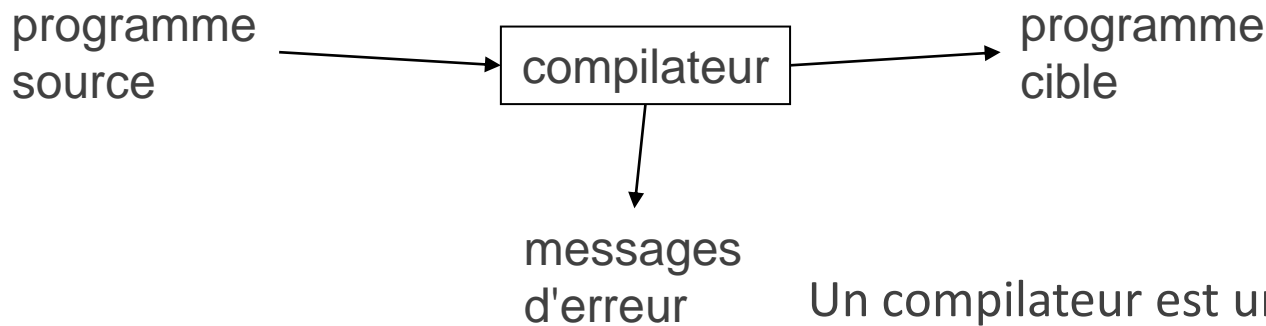
# Sommaire

Présentation du cours

Analyse lexicale

Le logiciel Flex

# Qu'est-ce qu'un compilateur ?



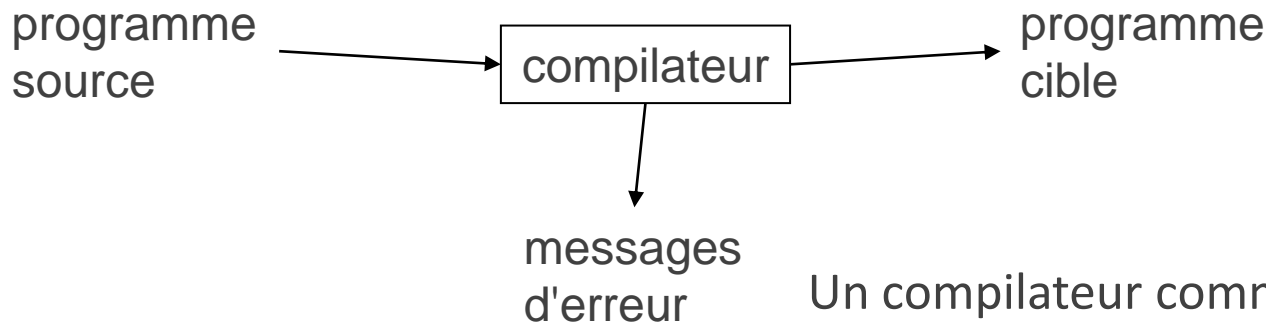
Un compilateur est un programme qui traduit un autre programme

Le programme source est dans un langage de haut niveau : C, C++, Java...

Le programme cible est dans un langage de bas niveau : assembleur, binaire, code à octets (*bytecode*) Java...

Premiers compilateurs :  
écrits par Alick Glennie et  
Grace Hopper (1952)

# Qu'est-ce que l'analyse syntaxique (*parsing*) ?



Un compilateur commence par analyser le programme qu'il va traduire

Analyser = identifier chaque partie

## **Analyse lexicale**

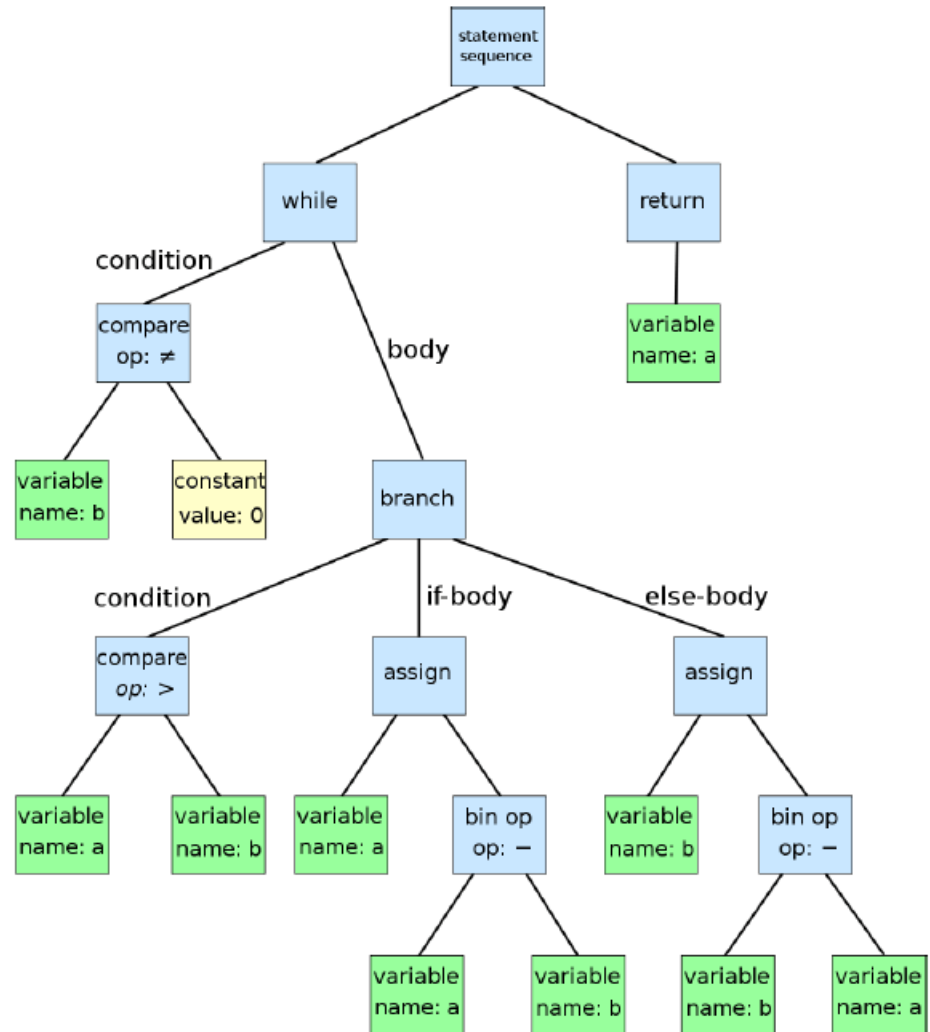
Mots-clés, identificateurs, opérateurs...

## **Analyse syntaxique**

Instructions, expressions, fonctions...

# Qu'est-ce que l'analyse syntaxique ?

```
while b != 0
  if a > b
    a := a - b
  else
    b := b - a
return a
```



# Objectifs du cours

> gcc toto.c

toto.c:1: erreur d'analyse syntaxique before ',' token

/tmp/ccImHOOB.o(.text+0x27): In function 'main' : undefined reference to 'get'

toto.c: Dans la fonction "main" :

toto.c:5: attention : suggest explicit braces to avoid ambiguous 'else'

Savoir écrire des analyseurs syntaxiques, de  
fichiers journaux (*log files*) par exemple

Savoir utiliser les logiciels Flex et Bison

Développer l'analyseur syntaxique pour le projet  
de compilation (2<sup>e</sup> semestre)

Comprendre les messages d'erreur des  
compilateurs

Savoir lire et écrire des grammaires

# Planning

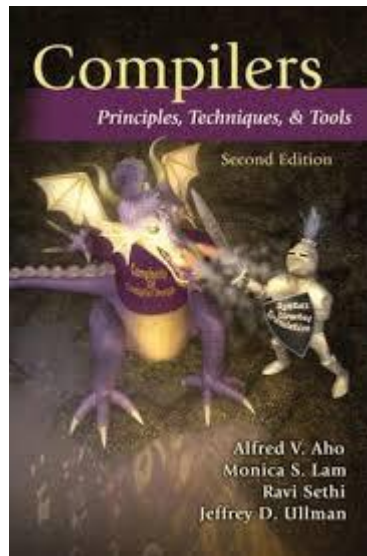
12 cours, 12 TD sur machine

1 projet de programmation 33 %

1 examen final 67 %

Le projet de programmation est obligatoire

# Bibliographie



Aho, Sethi, Ullman, 1986/2007. *Compilateurs. Principes, techniques et outils*, Pearson Education France

Levine, Mason, Doug, 1990. *Lex & Yacc*, O'Reilly.

Ce support de cours est inspiré de Aho *et al.* (1986)



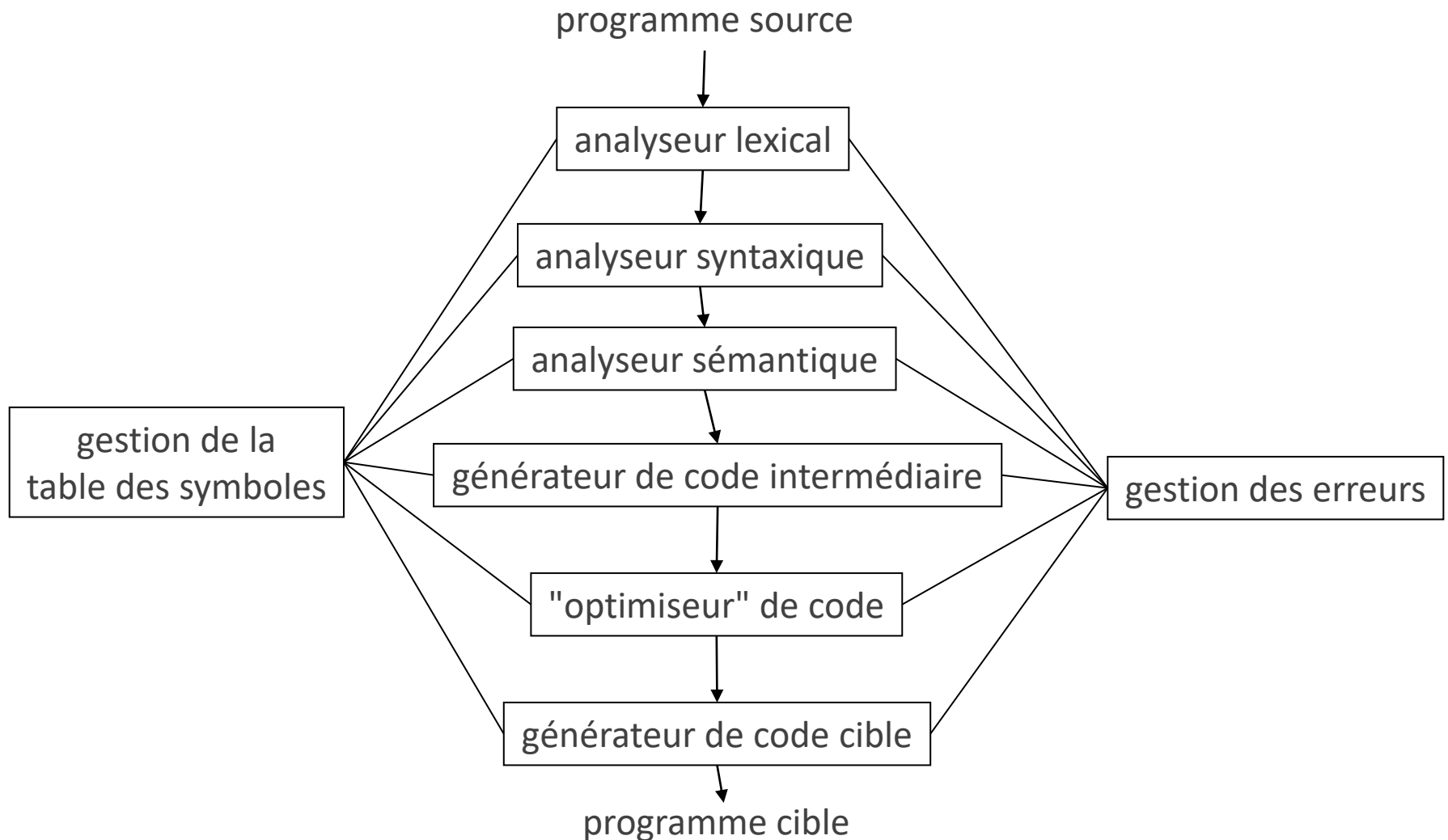
# Sommaire

Présentation du cours

Analyse lexicale

Le logiciel Flex

# Les phases de la compilation



# Analyse lexicale

position = initial + vitesse \* 60



[id, 1] [=] [id, 2] [+] [id, 3] [\*] [60]

Analyse du programme source en **lexèmes**  
(*tokens*)

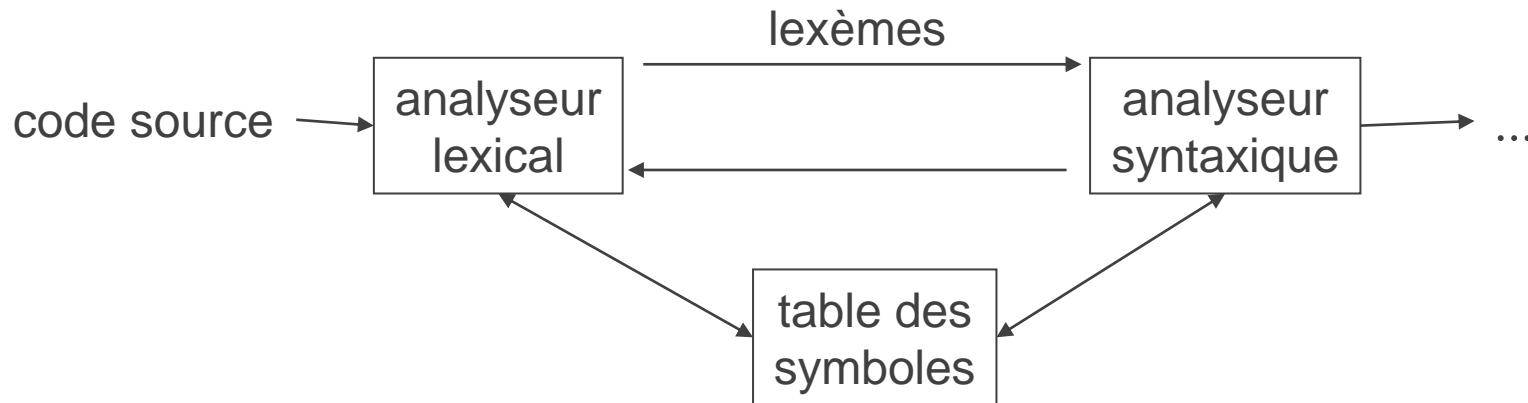
**Identificateurs** = noms des  
variables, des fonctions...

Les **identificateurs** rencontrés sont placés dans la  
table des symboles

Les blancs et les commentaires sont éliminés

À quoi servent les espaces dans le code source ?

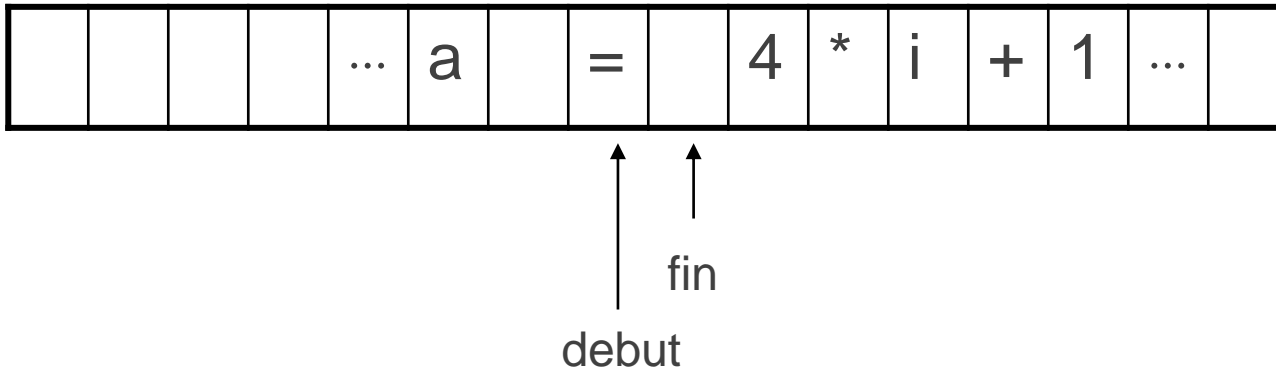
# Rôle d'un analyseur lexical (*lexer*)



Réduire la complexité de la  
conception et de  
l'implémentation  
Augmenter la flexibilité, la  
portabilité, la maintenabilité

Transformer un flot de caractères en flot de  
lexèmes  
Séparer l'analyse lexicale de l'analyse syntaxique  
Modulariser

## Deux méthodes de construction



Utiliser un générateur d'analyseurs lexicaux (Gcc  
contient Flex)

Écrire l'analyseur lexical dans un langage évolué

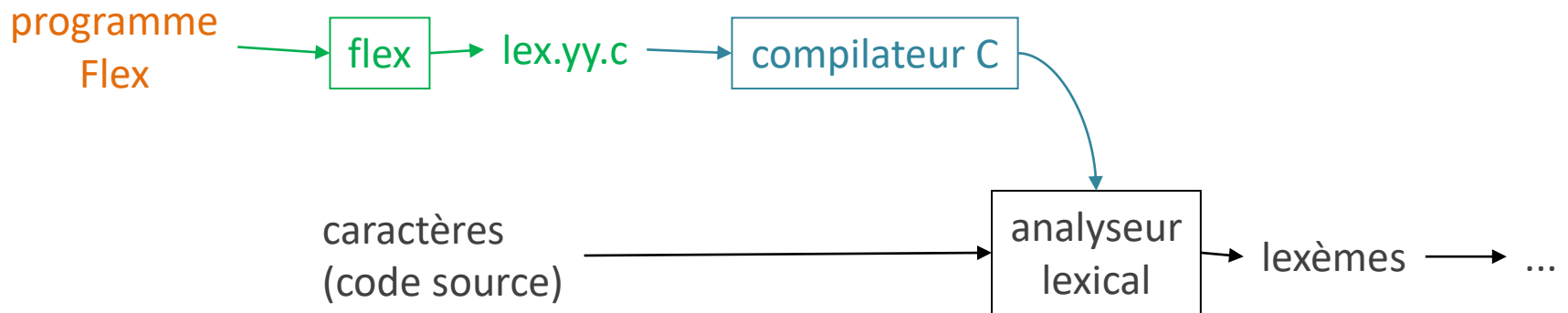
# Sommaire

Présentation du cours

Analyse lexicale

Le logiciel Flex

# Utilisation de Flex




4 étapes :

- créer avec un éditeur un programme Flex (expressions régulières)
- traiter cette spécification par la commande flex
- compiler le programme source C obtenu
- exécuter le programme exécutable obtenu

**Il faut un makefile**

# Que fait un analyseur lexical ?

`position = initial + vitesse * 60`



## Un analyseur lexical

- parcourt le code source
- reconnaît des lexèmes
- pour chaque lexème, lance une action

## L'analyseur lexical produit par Flex

pour chaque lexème reconnu, exécute un bout de code en C



# Exemple de programme Flex

```
%{  
int line_count=1;  
%}  
%%  
.* { line_count++; }  
%%
```

# Découpage du code source par l'analyseur lexical (1/3)

```
position = initial + vitesse * 60
```



Si *pirate* est reconnu, *rat* n'est pas reconnu

Un programme Flex minimal

L'analyseur lexical produit par Flex

- commence à chercher les lexèmes au début du code source ;
- après chaque lexème reconnu, recommence à chercher les lexèmes **juste après**

**Si aucun lexème n'est reconnu à partir d'un point du code source**

l'analyseur affiche le premier caractère sur la sortie standard et recommence à chercher à partir du caractère suivant

# Découpage du code source par l'analyseur lexical (2/3)

**Si plusieurs lexèmes sont reconnus à partir d'un  
point du code source (conflit)**

## **1. Deux lexèmes de longueurs différentes**

C'est le **plus long** qui gagne

Exemple 1 : les identificateurs

`[a-zA-Z_][a-zA-Z_0-9]*`

Cela permet d'aller jusqu'à la fin d'un  
identificateur

Exemple 2 : **a+++b**

parti

# Découpage du code source par l'analyseur lexical (3/3)

`while`

Exemple : conflit entre  
`[a-zA-Z_] [a-zA-Z_0-9]*`  
et *while*

`while`  
`[a-zA-Z_] [a-zA-Z_0-9]*`

## 2. Deux lexèmes de même longueur

Ils commencent au même point et terminent au même point : s'il y a conflit, c'est qu'ils sont issus de deux règles différentes. C'est la règle qui **apparaît la première** dans la spécification Flex qui gagne

Mettre l'exception avant la règle

En dehors de ce cas, l'ordre des règles ne joue pas

# Expressions régulières en Flex

- . N'importe quel caractère (octet) sauf retour à la ligne
- [xyz] Un caractère parmi x, y et z
- [abj-oZ] Un caractère parmi a, b, Z et n'importe lequel entre j et o
- [^A-Z] N'importe quel caractère autre que ceux entre A et Z
- r\* Zéro, une ou plusieurs fois l'expression r
- r+ Un ou plusieurs r
- r? Zéro ou un r
- r{2,5} Entre deux et cinq r
- r{2,} Au moins deux r
- r{4} Quatre r

Source : la page man de la commande flex

# Expressions régulières en Flex

- {nom} Comme l'expression définie comme « nom » dans les déclarations
- "[xyz]"glb" La chaîne de caractères [xyz]"glb"
- (r) Un r. Les parenthèses imposent un ordre d'application des opérateurs
- rs Un r suivi d'un s
- r|s Un r ou un s
- r/s Un r, mais uniquement s'il est suivi d'un s. La longueur prise en compte pour résoudre les conflits est celle de rs, mais la partie reconnue (yytext) est uniquement r
- ^r Un r, mais uniquement s'il est en début de ligne
- r\$ Un r, mais uniquement s'il est en fin de ligne
- <<EOF>> Fin de fichier

Les expressions régulières ci-dessus sont groupées en fonction de leur priorité, la plus haute au sommet et la plus basse en bas.

# Exemples

```
#include <stdio.h>
int yylex();
void yyerror(char *);
#line 74 "instr-comm.c" /* yacc.c:339 */
# ifndef YY_NULLPTR
#   if defined __cplusplus && 201103L <= __cplusplus
#     define YY_NULLPTR nullptr
#   else
#     define YY_NULLPTR 0
#   endif
# endif
/* Enabling verbose error messages. */
#ifdef YYERROR_VERBOSE
# undef YYERROR_VERBOSE
# define YYERROR_VERBOSE 1
#else
# define YYERROR_VERBOSE 0
#endif
```

1. Extraire d'un fichier en C les `#define`, `#ifdef`, `#ifndef` pour les compter
2. Remplacer les `sport` par des `activite` dans du code en C

# Programmes Flex

Un programme Flex est fait de trois parties :

**déclarations**

%%

**règles de traduction**

%%

**fonctions auxiliaires en C**

Les règles de traduction sont de la forme

$p_1$                       { *action*<sub>1</sub> }

$p_2$                       { *action*<sub>2</sub> }

...

$p_n$                       { *action*<sub>n</sub> }

où chaque  $p_i$  est une expression rationnelle et chaque action une suite d'instructions en C.



# Que fait un analyseur lexical avec un analyseur syntaxique ?

```
position = initial + vitesse * 60
```

## Un analyseur lexical

- parcourt le code source
- reconnaît des lexèmes
- pour chaque lexème, envoie des informations à l'analyseur syntaxique

## L'analyseur lexical produit par Flex

pour chaque lexème reconnu, renvoie des informations à l'analyseur syntaxique par un **return**

# Exemple

```
%{
/* Partie en langage C : définitions de constantes,
déclarations de variables globales, commentaires... */
%}
letter [a-zA-Z]
```

```
%%
```

```
[ \t\n]*      { /* pas d'action */ }
if             { return IF ; }
then          { return THEN ; }
else          { return ELSE ; }
{letter}({letter}|[0-9])* { yylval = install_id() ;
                        return ID ; }
([0-9]+(\.[0-9]*)?|\.[0-9]+)((E|e)(\+|-)?[0-9]+)? {
    yylval = install_num() ; return NUMBER ; }
```

## Exemple

```
"<"          { yylval = LT ; return RELOP ; }
"<="         { yylval = LE ; return RELOP ; }
```

```
%%
```

```
int install_id() {
/* fonction installant dans la table des symboles le
   lexème vers lequel pointe yytext et dont la longueur
   est yyleng. Renvoie l'indice de l'entrée dans la table
   */
}

int install_num() {
/* calcule la valeur du lexème et renvoie son indice dans
   une table */
}
```

# Syntaxe de Flex et Lex

Flex est une version de Lex (1975)

On utilise toujours l'extension .lex

Les **conventions d'écriture** des spécifications Flex sont d'époque

- Balises bizarres
- Espaces blancs significatifs
- Plusieurs parties avec des conventions différentes

Le **style logiciel** est d'époque aussi  
Beaucoup de variables globales

```
% {
. . .
% }
. . .
%%
. . .
%%
```

# Espaces blancs

```
%%
[ \t\n]*      { /* pas d'action */ }
if             { return IF ; }
then          { return THEN ; }
```

```
%%
[ \t\n]*      { /* pas d'action */ }
if             { return IF ; }
then          { return THEN ; }
```

!

```
%{
...
}%
...
%%
...
%%
```

Les espaces blancs sont significatifs  
Les balises doivent être en début de ligne

# Espaces blancs

```
%%
[ \t\n]*      { /* pas d'action */ }
if             { return IF ; }
then          { return THEN ; }
{letter}({letter}|[0-9])* { yylval = install_id() ;
                        return ID ; }
```

! →

```
%%
[ \t\n]*      { /* pas d'action */ }
if             { return IF ; }
then          { return THEN ; }
{letter}({letter}|[0-9])* { yylval = install_id() ;
                        return ID ; }
```

!

# Programmes Flex

Les commentaires `/* ... */` ne peuvent être insérés que dans une portion en C :

- dans la partie déclaration, seulement entre `%{` et `%}` ou dans des lignes commençant par un espace blanc ;
- dans la partie règles, seulement dans les actions ;
- dans la partie fonctions auxiliaires, n'importe où.

Dans les règles

$p_i$                       `{ actioni }`

les expressions rationnelles  $p_i$  ne peuvent pas contenir d'espaces blancs (ou alors dé-spécialisés).

Au début de la partie règles, si une ou plusieurs lignes sont entre `%{` et `%}` ou commencent par un espace blanc, elles sont interprétées comme du langage C et insérées dans `lex.yy.c` au début de la fonction qui reconnaît les motifs (utilisable pour déclarer des variables locales et ajouter des commentaires).

# Expressions régulières en Flex (suite)

## Séquences d'échappement

`\X` Si X est a, b, f, n, r, t ou v, le caractère `\X` en C ANSI. Si X est 0, le caractère de code ASCII 0. Sinon, un X déspecialisé

`\123` Le caractère de valeur octale 123

`\x2a` Le caractère de valeur hexadécimale 2a

Pour Flex, un « retour à la ligne » est le `'\n'` du compilateur C utilisé pour compiler Flex. En Windows, on doit parfois filtrer soi-même les `\r` de l'entrée ou utiliser explicitement `r/\r\n` pour « r\$ ».



# Expressions régulières en Flex

## À l'intérieur des crochets ([xyz])

À l'intérieur des crochets, tous les opérateurs d'expressions régulières perdent leur signification spéciale sauf « \ », « - », « ] » (sauf juste après le crochet ouvrant), et, juste après le crochet ouvrant, « ^ », donc les caractères suivants ne sont **pas** des caractères spéciaux :

. \* + ? { } " ( ) | / \$

et l'espace non plus.

Les crochets peuvent également contenir des expressions de classes de caractères. Ce sont des expressions entourées par des délimiteurs [: et :] (qui doivent elles-mêmes apparaître entre le '[' et le ']' extérieurs ; d'autres éléments peuvent aussi être présents dans les mêmes crochets). Les expressions valides sont :

[ :alnum: ] [ :alpha: ] [ :blank: ] [ :cntrl: ] [ :digit: ] [ :graph: ] [ :lower: ] [ :print: ] [ :punct: ] [ :space: ]  
[ :upper: ] [ :xdigit: ]

Ces expressions désignent chacune un groupe de caractères équivalent à la fonction C standard correspondante isXXX.

```
#include <stdio.h>
int yylex();
void yyerror(char *);
#line 74 "instr-comm.c" /* yacc.c:339 */
# ifndef YY_NULLPTR
#   if defined __cplusplus && 201103L <= __cplusplus
#     define YY_NULLPTR nullptr
#   else
#     define YY_NULLPTR 0
#   endif
# endif
/* Enabling verbose error messages. */
#ifdef YYERROR_VERBOSE
# undef YYERROR_VERBOSE
# define YYERROR_VERBOSE 1
#else
# define YYERROR_VERBOSE 0
#endif
```

## Exemples

- Extraire d'un fichier en C les `#define`, `#ifdef`, `#ifndef` et les copier dans un autre fichier
- avec la ligne entière
  - avec les extensions sur une ou plusieurs autres lignes

# Un peu de vocabulaire

Différence entre lettre et caractère

**Caractères**

Lettres

aàbcçdé...AÀBCÇDÉ...

Chiffres

0123...

Espaces blancs (*whitespace*)

Autres

% = " ' . \* + ? { } " ( ) | / \$...

# Un peu de vocabulaire

Entre apostrophes (*in single quotes*)

' r '

Entre guillemets (*in double quotes*)

" r "

Entrecôte (*rib steak*)



Photo by Dcollard - Own work, CC BY-SA 3.0  
<https://commons.wikimedia.org/w/index.php?curid=21614585>

## Exemple : chaines de caractères entre guillemets

`\("[^"]*" \)`

Reconnait

`"bonjour"` `""`

Ne reconnait pas

`"bonjo"` `"bon"jour"`

`"%s\n\"%s\""\n"`

`\("[^"\\]|\\.|\\\\n)*\)`

Reconnait

`"%s\n\"%s\""\n"`

Ne reconnait pas

`"bonjo"` `"bon"jour"`

# Fonctions et variables définies par Flex

## **yylex()**

Reconnait les motifs et exécute les actions  
correspondantes

Si une action contient un return, il termine `yylex()`

## **yytext**

Chaine de caractères contenant la partie reconnue

## **yylen**

Longueur de `yytext` (sans le 0 final)

## **ECHO;**

Écrit `yytext`

## main() dans Flex

```
%%  
int main() {  
    return yylex();  
}
```

```
gcc lex.yy.o -Wall -lfl
```

On peut écrire un `main()` dans un programme Flex  
Sinon, la **bibliothèque de Flex** (option **-lfl**) fournit  
un `main()` qui appelle seulement `yylex()`  
L'option **-lfl** doit apparaître à la fin et non au début  
de la ligne de commande

# Options Flex

```
%{  
/* Partie en langage C */  
%}  
letter [a-zA-Z]  
  
%option nounput  
%option noinput  
%option noyywrap  
  
%%  
[ \t\n]*      ;  
if             { return IF ; }  
then          { return THEN ; }  
else          { return ELSE ; }
```

**%option nounput**

**%option noinput**

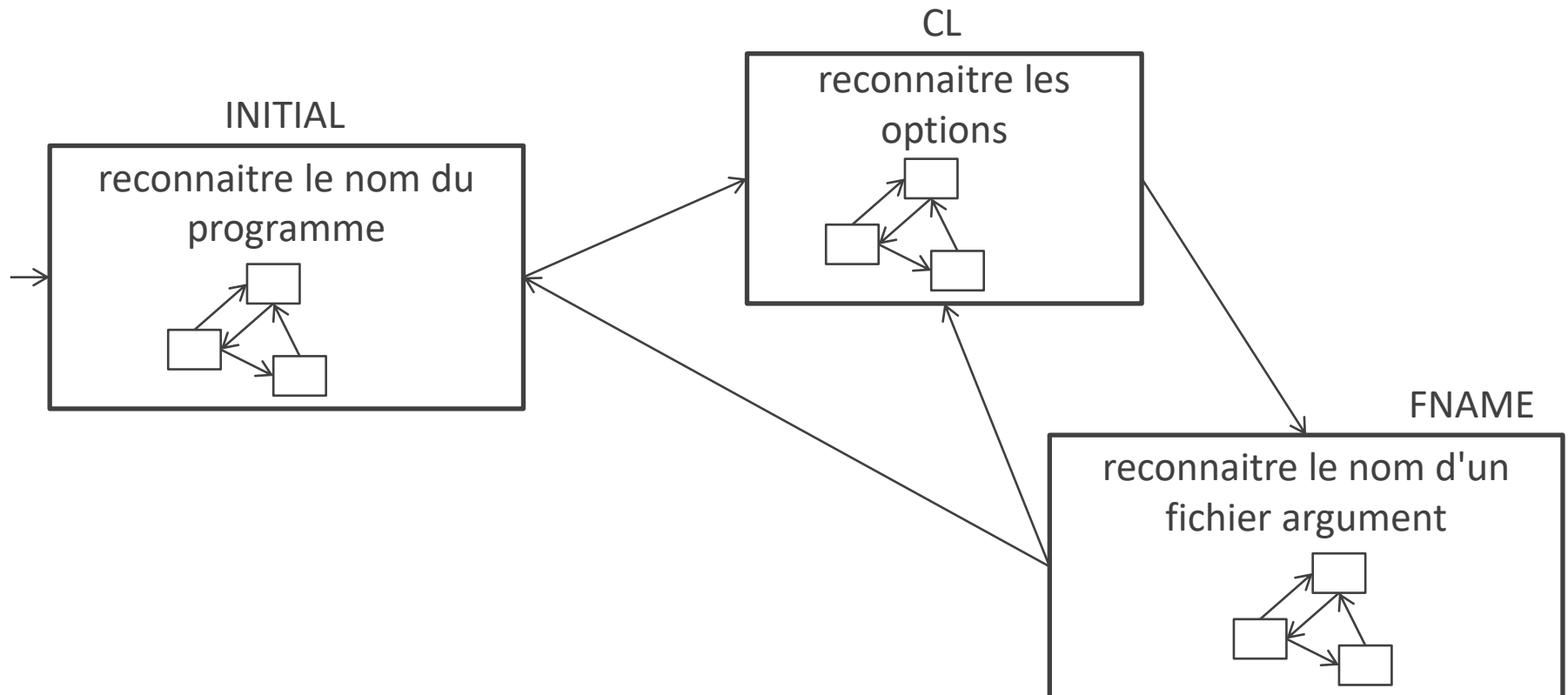
Supprime des avertissements du  
compilateur sur des fonctions  
définies et non utilisées

**%option noyywrap**

Utile si on n'utilise pas la bibliothèque  
de Flex (-lfl)



# Conditions de démarrage : exemple



## Conditions de démarrage exclusives (déclarées avec %x)

<code>&lt;c&gt;r</code>	Un r, seulement si on est dans la condition de démarrage c
<code>&lt;c1,c2,c3&gt;r</code>	Un r, dans une des conditions de démarrage c1, c2 ou c3
<code>&lt;*&gt;r</code>	Un r, dans n'importe quelle condition de démarrage

Au départ, on est automatiquement dans la condition de démarrage INITIAL. Les expressions régulières sans `<...>` sont reconnues.

Pour passer dans la condition de démarrage c, utiliser dans une action :

BEGIN c;

Les expressions régulières sans `<...>` ne sont plus reconnues.

Pour retourner dans la condition de démarrage INITIAL, utiliser :

BEGIN INITIAL;

Les expressions régulières sans `<...>` sont à nouveau reconnues.

# Conditions de démarrage exclusives (déclarées avec %x)

déclarations

%%

règles de traduction

%%

fonctions auxiliaires en C

On ne peut pas déclarer la condition de  
démarrage INITIAL

Il faut déclarer les autres dans la partie  
déclarations

```
%x c1 c2 c3  
%%
```

```
%x c1  
%x c2  
%x c3  
%%
```

# Conditions de démarrage exclusives (déclarées avec %x)

```
%x CL FNAME
```

```
%%
```

```
^[a-zA-Z][-a-zA-Z0-9_]* {
    strcpy(progName, yytext);
    BEGIN CL; verbose=fname=0; }
<CL>[ ]+ /* ignore spaces */
<CL>-h|-\?|-help {
    printf("Usage: (...) \n",
        progName);
    BEGIN INITIAL; }
<CL>-v|-verbose { verbose=1; }
<CL>-f|-filename { BEGIN FNAME; }
<CL>\n { call();
    BEGIN INITIAL; }
<FNAME>[ ]+ /* ignore spaces */
<FNAME>[^ \n]+ { strcpy(argName, yytext);
    BEGIN CL; fname=1; }
<FNAME>\n { call();
    BEGIN INITIAL; }
```

```
%%
```

Il ne peut pas y avoir de conflit entre une règle <c1>r et une règle <c2>r

Il ne peut pas y avoir de conflit entre une règle <c>r et une règle sans <...> (sauf si c=INITIAL)

On peut regrouper les règles <c>r (sauf si c=INITIAL) :

- l'ordre des règles entre <c>r et <c2>r ne joue pas

- l'ordre des règles entre <c>r et r ne joue pas

```
%{
#define MAXNAME 256
char progName[MAXNAME], argName[MAXNAME]; int verbose, fname; void call();
}%
%x CL FNAME
%option nounput
%option noinput
%%
^[a-zA-Z][-a-zA-Z0-9_.]* { strcpy(progName, yytext); BEGIN CL; verbose=fname=0; }
<CL>[ ]+ /* ignore spaces */
<CL>-h|-?|-help { printf("Usage: %s [-help | -h | -?] [-verbose | -v]"
                        " [(-file | -f) filename]\n", progName);
                  BEGIN INITIAL; }
<CL>-v|-verbose { verbose=1; }
<CL>-f|-filename { BEGIN FNAME; }
<CL>\n { call(); BEGIN INITIAL; }
<FNAME>[ ]+ /* ignore spaces */
<FNAME>[^ \n]+ { strcpy(argName, yytext); BEGIN CL; fname=1; }
<FNAME>\n { call(); BEGIN INITIAL; }
%%
void call() {
    printf("v%d f%d ", verbose, fname);
    if (fname)
        printf("%s", argName);
    printf("\n");}
```

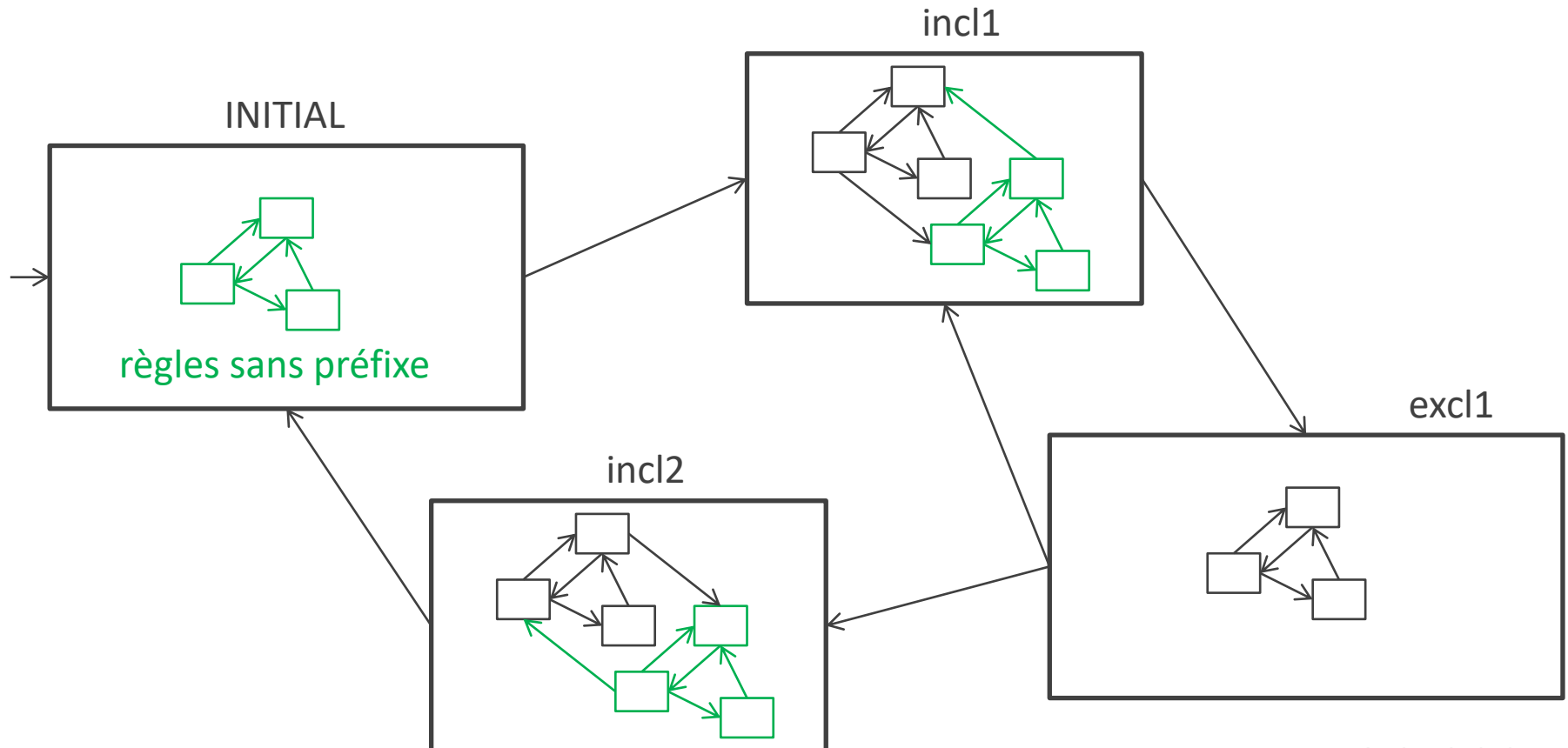
<<EOF>>

<<EOF>>

Fin de fichier

<<EOF>> n'est compatible avec aucun opérateur  
d'expressions régulières sauf <...>

# Conditions de démarrage inclusives



# Conditions de démarrage inclusives (déclarées avec %s)

Comme les conditions de démarrage exclusives sauf



%s c1 c2 c3  
 %%

r	Un r, si on est dans INITIAL ou une condition de démarrage inclusive
<c>r	Un r, seulement si on est dans la condition de démarrage c
<c1,c2,c3>r	Un r, dans une des conditions de démarrage c1, c2 ou c3
<*>r	Un r, dans n'importe quelle condition de démarrage

La condition de démarrage INITIAL se comporte comme une condition de démarrage inclusive, mais:

- on ne la déclare pas
- on est automatiquement dedans au départ.

Il peut y avoir conflit entre une règle <c>r et une règle sans <...>

On ne peut pas toujours regrouper les règles <c>r