Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

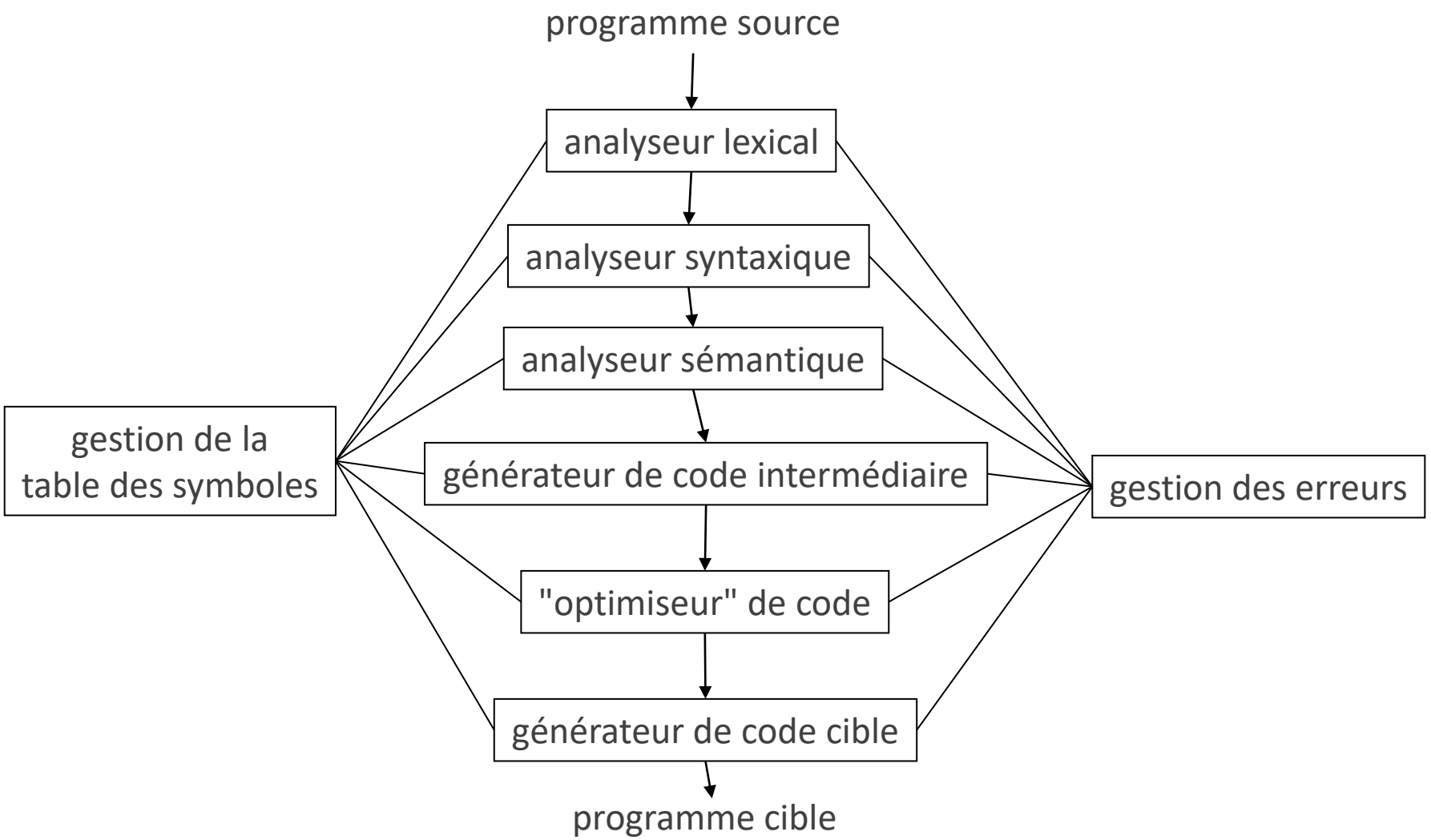
Organisation d'un compilateur

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Les phases de la compilation



Université Gustave Eiffel

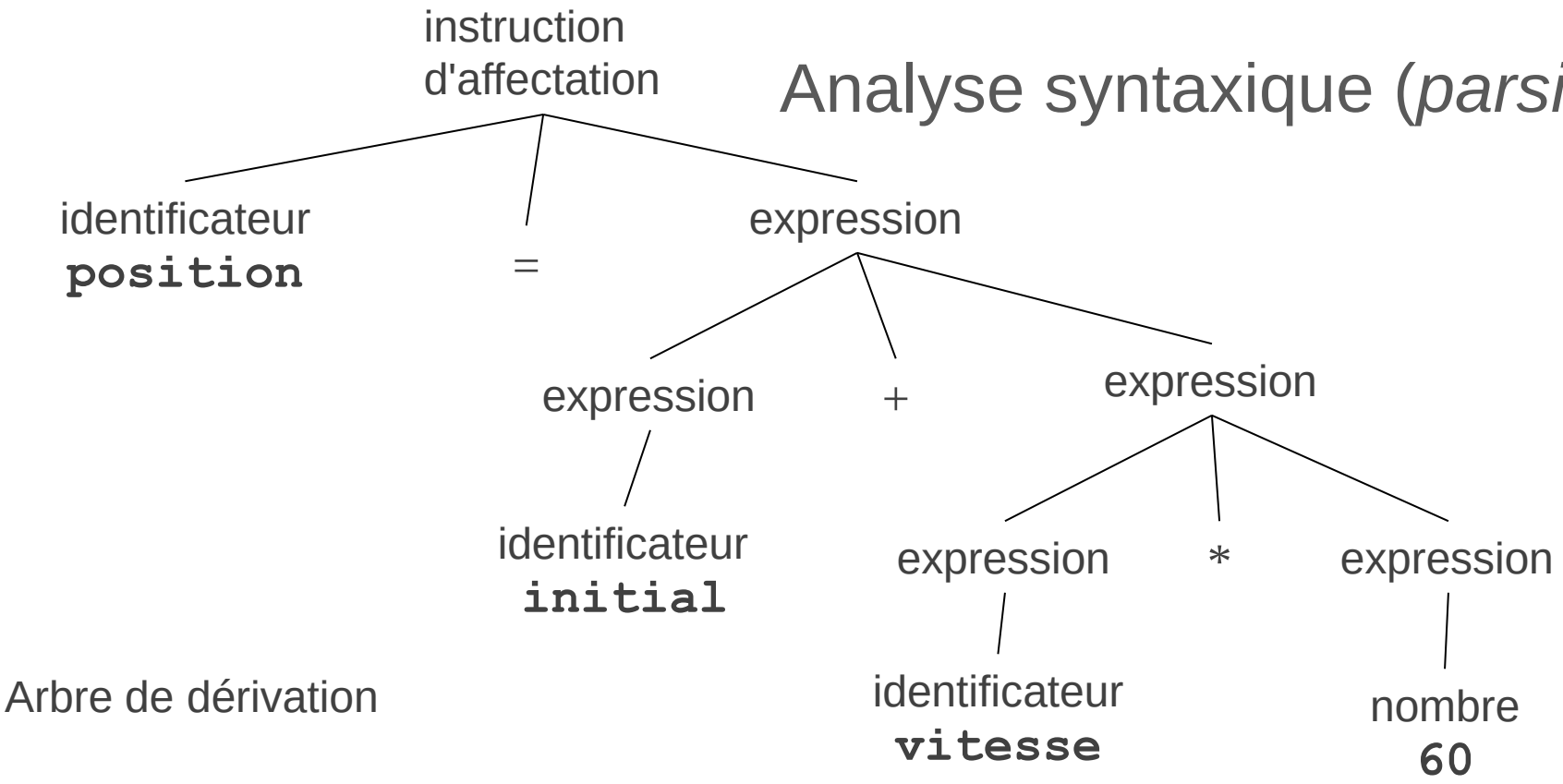
INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Analyse lexicale

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

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

Analyse syntaxique (*parsing*)



Arbre de dérivation

On reconstruit la structure syntaxique de la suite de lexèmes fournie par l'analyseur lexical

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Génération de code intermédiaire

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Programme pour une machine abstraite

Représentations utilisées

- Code à trois adresses
- Arbres syntaxiques

"Optimisation" de code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```



```
temp2 := id3 * 60.0
id1 := id2 + temp2
```

- Elimination des opérations inutiles pour produire du code plus efficace
- La constante est traduite en réel flottant à la compilation, et non à l'exécution
- La variable **temp3** est éliminée

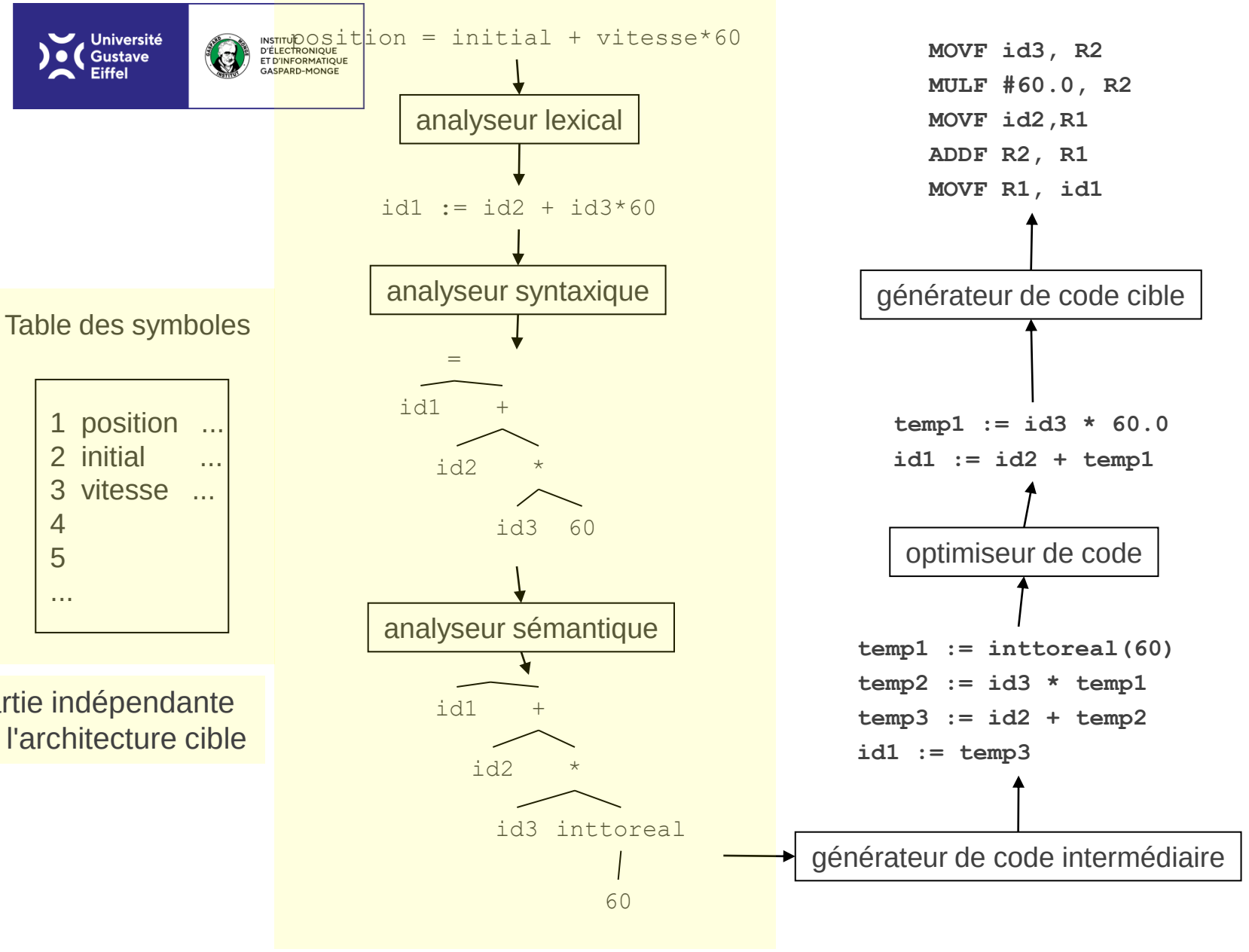
Génération de code cible

```
temp2 := id3 * 60.0
id1 := id2 + temp2
```



```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

- La dernière phase produit du code en langage d'assemblage
- Un point important est l'utilisation des registres
- F = flottant
- La première instruction transfère le contenu de **`id3`** dans le registre R2
- La seconde multiplie le contenu du registre R2 par la constante **`60.0`**


position = initial + vitesse*60
analyseur lexical
id1 := id2 + id3*60
analyseur syntaxique
=
id1 +
id2 *
id3 60
analyseur sémantique
id1 +
id2 *
id3 inttoreal
60
générateur de code intermédiaire
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
optimiseur de code
temp1 := id3 * 60.0
id1 := id2 + temp1
générateur de code cible
MOVF id3, R2
MULF #60.0, R2
MOVF id2,R1
ADDF R2, R1
MOVF R1, id1
Table des symboles
1 position ...
2 initial ...
3 vitesse ...
4
5
...
Partie indépendante de l'architecture cible

Préprocesseurs

```
#include <stdio.h>
#define PI 3.14159
#define MIN(a,b) a<b?a:b
```

```
int a=0, b=1, c;
c=MIN(a,b)+10; /* c vaut 0 */

/* solution : */
#define MIN(a,b) (a<b?a:b)
```

```
int a=0, b=1, c=2, d;
d=MIN(a&&c,b); /* d vaut 1 */

/* solution : */
#define MIN(a,b) (a)<(b)?(a):(b)
```

- Les définitions de macros permettent l'utilisation de paramètres
- Si le préprocesseur est intégré au compilateur, il fait partie de l'analyse lexicale et non de l'analyse syntaxique
- Remplacement simpliste

Préprocesseurs

Si on écrit

```
\JACM 17;4;715-728
```

on doit voir

J.ACM **17**:4, pp. 715-728

On définit :

```
\define\JACM #1;#2;#3.
  {{\sl J.ACM} {\bf #1}:#2, pp. #3}
```

Les définitions de macros existent aussi dans les logiciels de composition de documents

En TeX, une définition de macro a la forme

```
\define <nom> <modèle>{<corps>}
```

Si on écrit

```
\JACM 17;4;715-728
```

on obtient

```
{\sl J.ACM} {\bf 17}:4, pp. 715-728
```

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Analyse lexicale et syntaxique
Partage des tâches

```
float perimeter= 3.1416 ;
if(min[n] <= max(1)){
   return ;
}
```

```
pic_list *s;
if( min [n]<= max (1)){
    return;
}
```

Reconnaitre un nombre à virgule flottante : analyseur lexical ou syntaxique ?

Reconnaitre un opérateur à 2 caractères

Différencier un mot-clé d'un identificateur

Différencier un nom de variable d'un nom de fonction ou d'un nom de type

Analyse lexicale et syntaxique
Partage des tâches

```
float perimeter=3.1416;
if(min[n]<=max(1)){
    return;
}
```

```
float perimeter = 3.1416 ;
if ( min [ n ] <= max ( 1 ) ) {
    return ;
}
```

~~float perimeter=3 . 1416;~~
~~if(min[n]< =max(1)){~~
~~   return;~~
~~}~~

Un analyseur lexical élimine les espaces blancs inutiles

Si les règles du langage permettent de séparer deux éléments par des espaces, ce sont deux lexèmes différents, sinon c'est un seul lexème

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Analyse lexicale et syntaxique
Partage des tâches

```
pic_list *s;
if(min[n]<=max(1)){
    return;
}
```

~~
```
while *s;
if(struct[n]<=switch(1)){
    return;
}
```
~~

Mots-clés

Un analyseur lexical reconnait facilement les mots-clés

Classification des identificateurs

Un identificateur peut être un nom de variable, de fonction, de type ou de constante, suivant le contexte

Un analyseur lexical explore difficilement le contexte

C'est l'analyseur syntaxique qui fait la différence

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Groupement des phases

Partie amont (*front end*)

Regroupe tout ce qui ne dépend pas de l'architecture cible

On peut utiliser la même partie amont sur une machine différente

Partie aval (*back end*)

Regroupe le reste

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Groupement des phases

Passes

Plusieurs phases peuvent être regroupées dans une même passe consistant à lire un fichier et en écrire un autre

Analyse lexicale, syntaxique, sémantique et génération de code intermédiaire peuvent être regroupées en une seule passe

La réduction du nombre de passes accélère le traitement

Outils logiciels d'aide à la construction de compilateurs

Flex, Lex

Générateurs d'analyseurs lexicaux

Engendrent un analyseur lexical (*lexer, scanner*) sous forme d'automate fini à partir d'une spécification sous forme d'expressions rationnelles

Bison, Yacc

Générateurs d'analyseurs syntaxiques

Engendrent un analyseur syntaxique (*parser*) à partir d'une grammaire

Bison, Yacc

Générateurs de traducteurs

Engendrent un traducteur à partir d'un schéma de traduction (grammaire + règles sémantiques)

Compilation d'un programme C

Si on compile par **`gcc -S bonjour.c`** le fichier suivant :

```
#include <stdio.h>

  float radius=6.4,circum;
  circum=6.28*radius;
  return 0;
 }
```

on obtient de l'assembleur

- **LFB** pour *label function begin*
- **ebp** pour *base pointer*
- **esp** pour *stack pointer*
- **pushl** pour *push long*

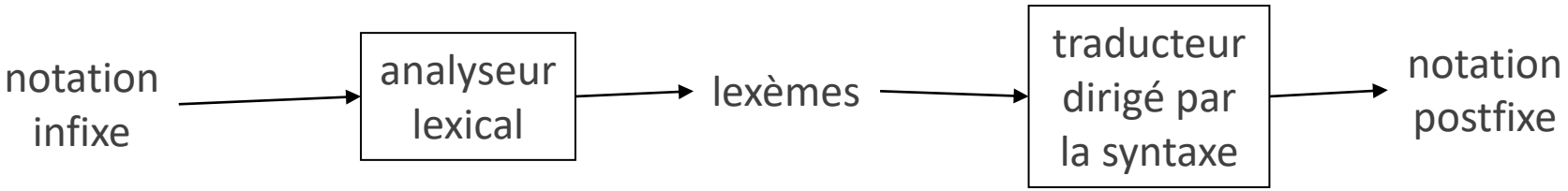
etc.

```
#include <stdio.h>

int main(void) {
  float radius=6.4,circum;
  circum=6.28*radius;
  return 0;
 }
```

```
    .file   "arith.c"
    .def    ___main;    .scl    2;  .type   32; .endef
    .text
    .globl  _main
    .def    _main;  .scl    2;  .type   32; .endef
_main:
LFB7:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    andl    $-16, %esp
    subl    $32, %esp
    call    ___main
    movl    $LC0, %eax
    movl    %eax, 28(%esp)
    flds    28(%esp)
    fldl    LC1
    fmulp   %st, %st(1)
    fstps   24(%esp)
    movl    $0, %eax
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
LFE7:
    .section .rdata,"dr"
    .align 4
LC0:
    .long   1087163597
    .align 8
LC1:
    .long   1374389535
    .long   1075388088
    .ident  "GCC: (GNU) 4.9.3"
```

Un mini-compilateur



(12+27)*5-7

12 27 + 5 * 7 -

12+27*(5-7)

12 27 5 7 - * +

Construction d'un traducteur d'expressions arithmétiques en notation postfixe

On décrit la syntaxe par une grammaire

On emploie la méthode de traduction dirigée par la syntaxe

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Introduction à la syntaxe

bloc --> **begin** *opt_insts* **end**
opt_insts --> *inst_list* | ε
inst_list --> *inst_list* **;** *inst*
inst_list --> *inst*

list --> *list* + *chiffre*
list --> *list* - *chiffre*
list --> *chiffre*
chiffre --> **0|1|2|3|4|5|6|7|8|9**

On spécifie la syntaxe par une **grammaire**

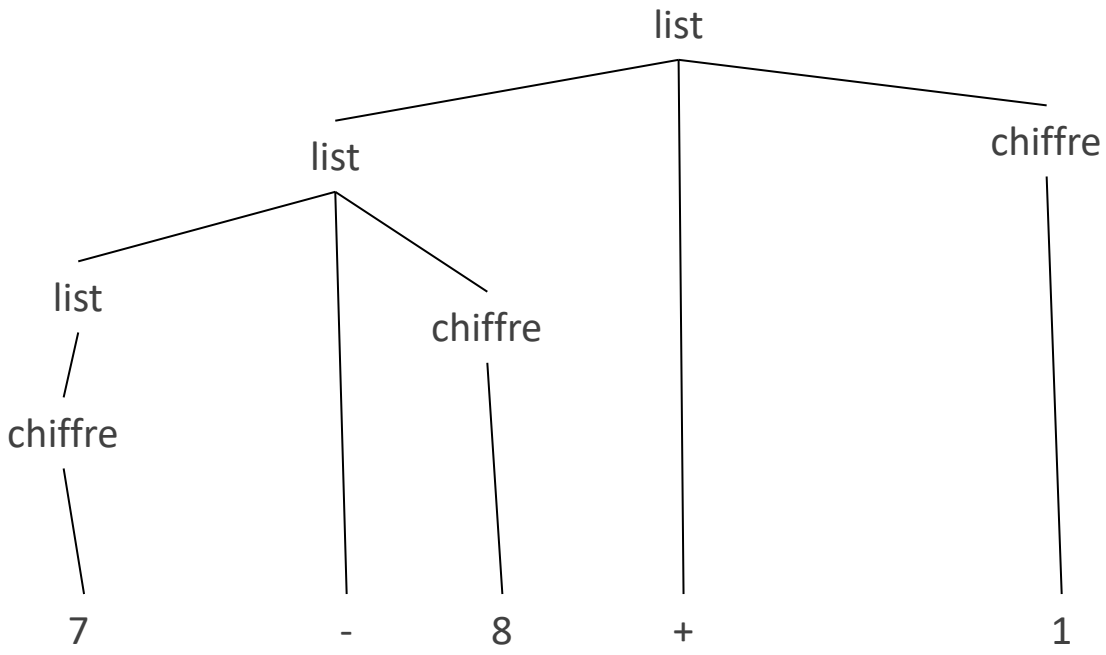
Une **règle** est de la forme

inst --> **if (** *exp* **)** *inst* **else** *inst*

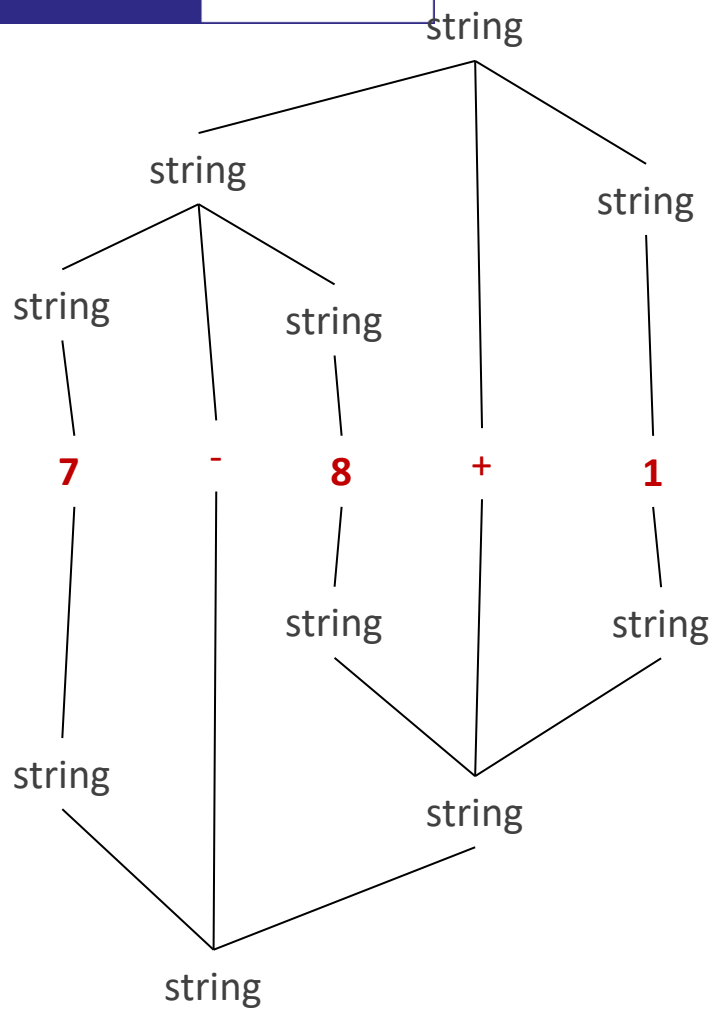
On aura par exemple une grammaire pour les blocs d'instructions

Une grammaire pour les listes de chiffres séparés par des + ou des -

Arbre de dérivation



On utilise les grammaires pour construire des arbres de dérivation



Ambiguïté

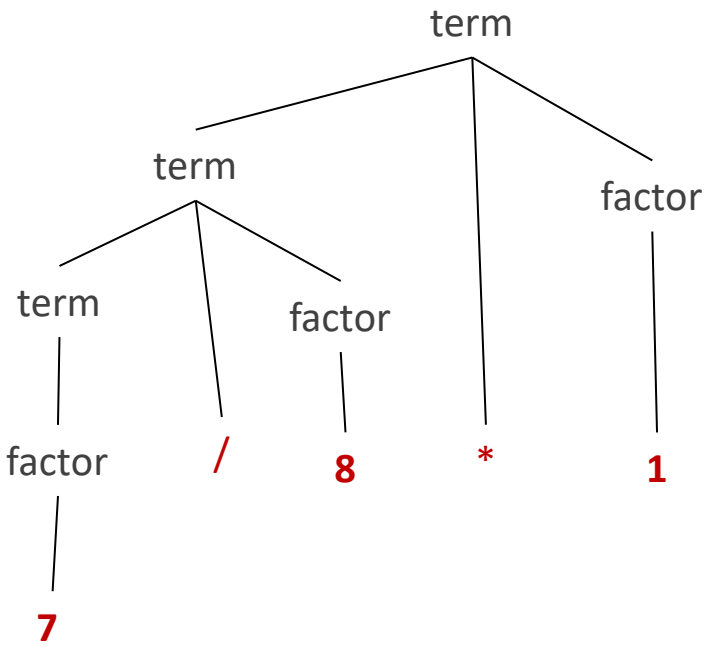
La grammaire

string --> *string* + *string* | *string* - *string*
|**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**

est ambiguë

Associativité :

quand (*a* op *b*) op *c* = *a* op (*b* op *c*)



Une grammaire non ambiguë

On utilise trois niveaux de priorité pour forcer
- l'associativité de gauche à droite
- la priorité de * et / sur + et -

Pour les expressions les plus simples :

factor --> **chiffre** | **(** *expr* **)**

Pour le deuxième niveau :

term --> *term* ***** *factor*
| *term* / *factor*
| *factor*

Pour le troisième niveau :

expr --> *expr* **+** *term*
| *expr* - *term*
| *term*

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Grammaire attribuée

On ajoute des actions et on fixe un ordre de visite de l'arbre d'analyse : l'ordre en profondeur (*depth-first*)

expr -->	*expr* **+** *term*	{ print("+") ; }
expr -->	*expr* **-** *term*	{ print("-") ; }
expr -->	*term*	
term -->	**0**	{ print("0") ; }
term -->	**1**	{ print("1") ; }
...		
term -->	**9**	{ print("9") ; }

Résultat : traduction en forme postfixe

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Introduction à l'analyse syntaxique

L'analyse syntaxique est la construction de l'arbre de dérivation à partir de la suite de symboles

Analyse par descente récursive : la plus facile

1. Faire une fonction pour chaque non-terminal de la grammaire
2. Ranger le prochain lexème dans une variable globale
3. Suivre les règles de la grammaire

Analyse ascendante : permet de traiter plus de cas

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Version 0

Traduction infixe-postfixe des expressions additives

Les lexèmes sont constitués d'un seul caractère

On aura dans la version 2

- des identificateurs rangés dans une table des symboles

- les expressions multiplicatives

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

main()

```
#include <ctype.h>   /* charge isdigit() */
int lookahead ;      /* contient le prochain caractère */

int main() {
   lookahead = getchar() ;
   expr() ;
   putchar('\n') ;   /* ajoute une fin de ligne */
   return 0;
   }
```

Les lexèmes sont donnés par la fonction getchar()

La fonction match() vérifie les lexèmes et lit le suivant. Elle appelle error() si ce qu'elle lit n'est pas conforme

Les expressions

```
void expr() {
   term() ;
   while(1)
        if (lookahead == '+') {
               match('+') ; term() ; putchar('+') ; }
        else if (lookahead == '-') {
               match('-') ; term() ; putchar('-') ; }
        else break ; }
```

On transforme une expression de la forme

term + term + ... + term

en

term term + ... term +

Les termes

```
void term() {
   if (isdigit(lookahead)) {
       putchar(lookahead) ;
       match(lookahead) ; }
   else error() ; }

void match(int t) {
   if (lookahead == t)
       lookahead = getchar() ;
   else error() ; }
```

Les termes sont obtenus par l'analyse lexicale

On a changé de grammaire

Rappel : la grammaire de départ

expr --> *expr* **+** *term*
expr --> *expr* **-** *term*
expr --> *term*
term --> **0**
term --> **1**
...
term --> **9**

Ce premier programme revient à utiliser une grammaire un peu différente :

expr --> *term expr'*
expr' --> **+** *term expr'*
expr' --> **-** *term expr'*
expr' --> *ε*
term --> **0**
term --> **1**
...
term --> **9**

Le langage engendré est le même

Avec la grammaire précédente, la première instruction de la fonction **`expr()`** aurait été un appel à elle-même

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Traitement des erreurs

On signale simplement l'erreur

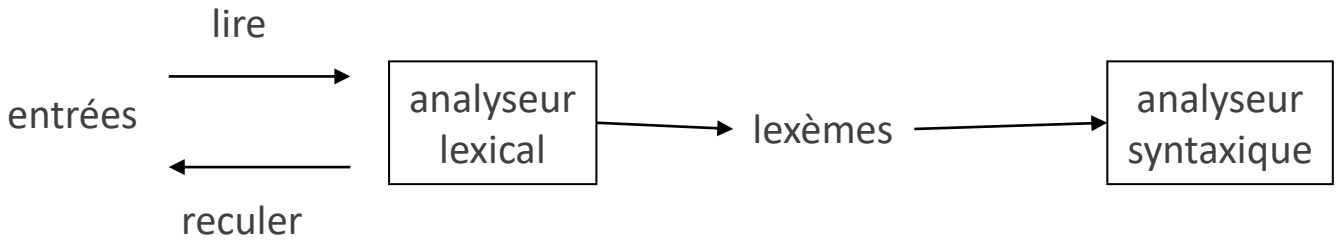
```
void error() {
   printf("syntax error\n") ;
   exit(1) ; }        /* arrête
   l'exécution */
```

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Analyse lexicale



`12 + 45 - 8`

est transformé en :

<num, 12> <+,> <num, 45> <-,> <num, 8>

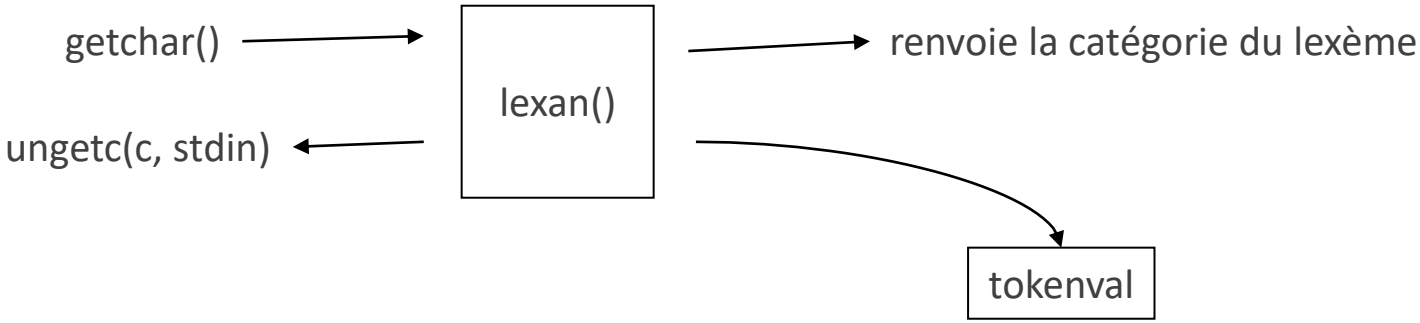
On prépare la version 1

Éliminer les espaces, lire les constantes

L'analyseur lexical est appelé par l'analyseur syntaxique et lui fournit des informations

L'analyseur lexical renvoie à l'analyseur syntaxique des couples (catégorie du lexème, attribut)

Réalisation




```
#define NUM 256
```

La catégorie du lexème est représentée par un entier déclaré comme constante symbolique

Les autres informations sont transmises par l'attribut

Version 1

```
int lineno=1;
int tokenval = NONE;
int lexan(void) {   /*analyseur lexical*/
 int t;
 while(1) {
    t = getchar();
    if (t == ' ' || t == '\t')
        ; /* sauter les espaces blancs */
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit(t)) {  /* t est un chiffre*/
      ungetc(t, stdin);
      scanf("%d", &tokenval);
      return NUM; }
```

Code C pour éliminer les espaces et rassembler les chiffres

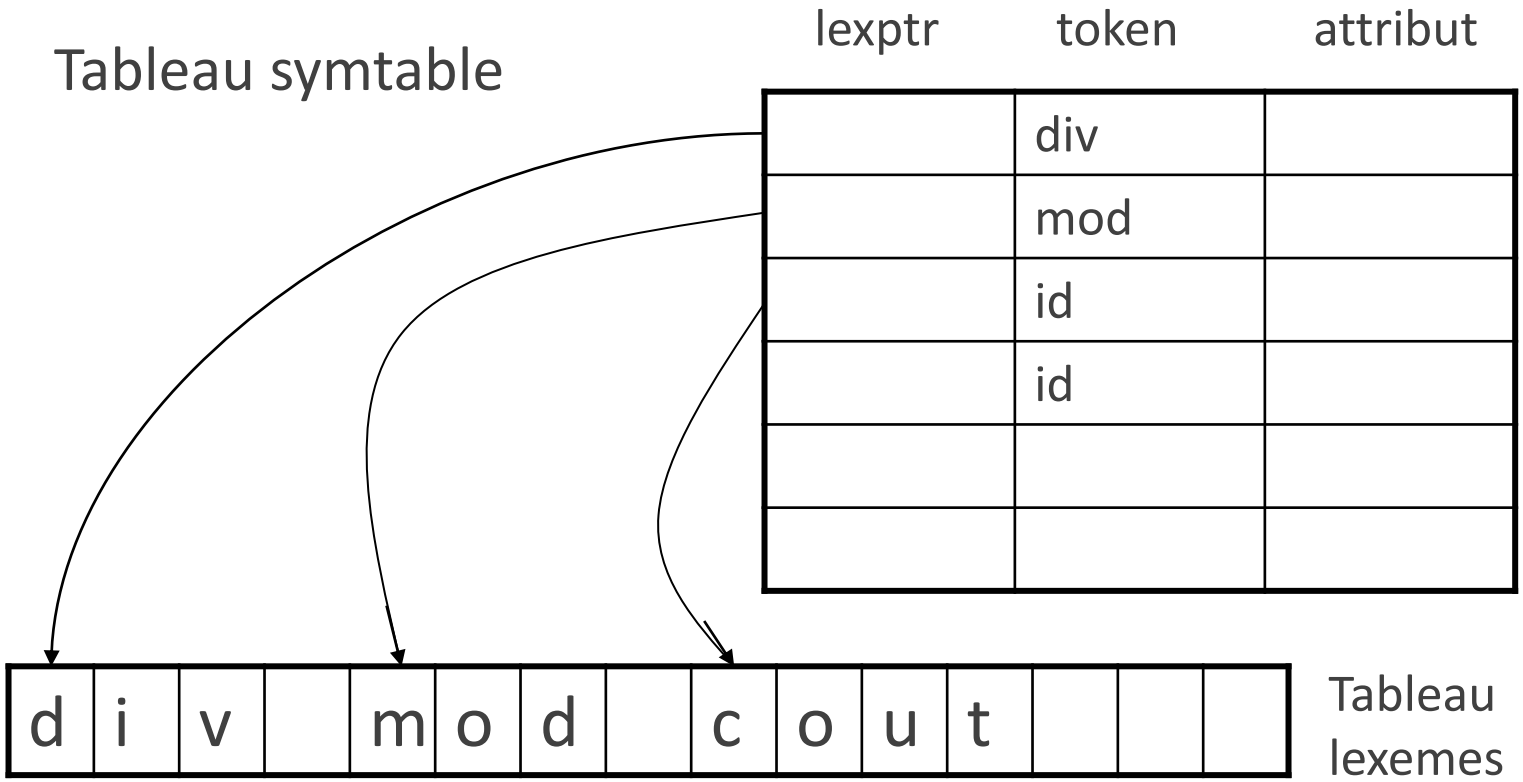
Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Version 1

```
else {
        tokenval = NONE;
        return t;
    }
  }
}
```

Ajout d'une table des symboles



Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Ajout d'une table des symboles

La table des symboles utilise deux fonctions

insert(s, t) crée et renvoie une entrée pour la chaîne s et le lexème t

lookup(s) renvoie l'indice de la chaîne s, ou 0 si elle n'y est pas

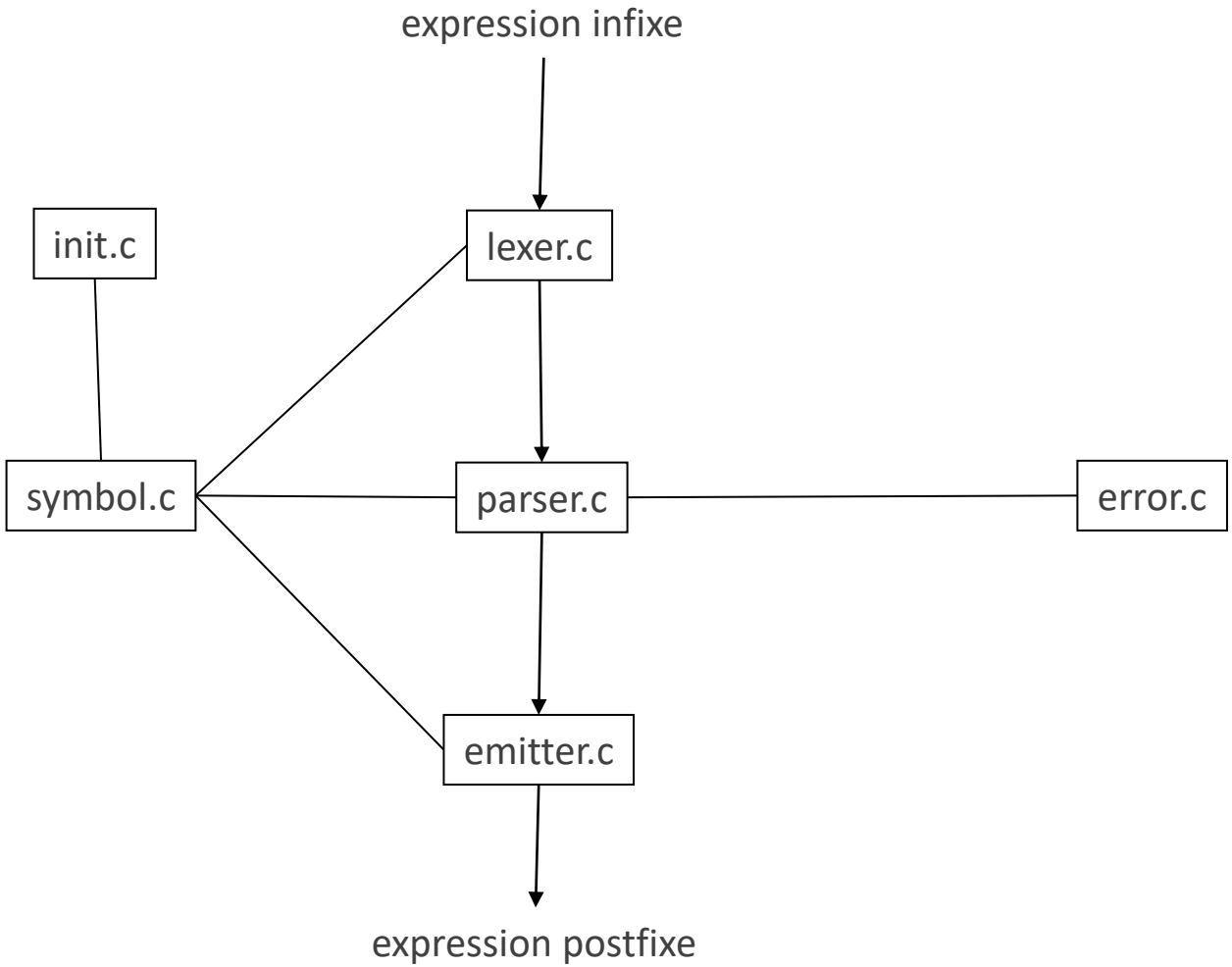
On peut ainsi traiter les mots-clés : insert("div", div)

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

Sommaire

Architecture du compilateur



Version 2

```
/******* global.h ************/

#include <stdio.h>    /*charge des routines i/o*/
#include <ctype.h>    /*charge les routines de */
                      /*test de caractères*/
#include <string.h>

#define BSIZE  128  /*taille du tampon*/
#define NONE    -1
#define  EOS   '\0'

#define NUM    256
#define DIV    257
#define MOD    258
#define ID     259
#define DONE   260

extern int  tokenval;  /*valeur de l'attribut du lexeme*/
extern int  lineno;

struct entry {  /*structure des elements de la */
  char *lexptr;    /*table des symboles*/
  int token;
};
```

```
extern struct entry symtable[];  /*table des symboles*/

void init(void);

void error(char *m);

void emit(int t, int tval);

int insert(char s[], int tok);

void parse(void);

void expr(void);

void term(void);

void factor(void);

void match(int t);

int lexan(void);

int lookup(char s[]);
```

```
/************** init.c *********/

#include "global.h"

static struct entry keywords[] = {
     {"div", DIV},
     {"mod", MOD},
     {0,   0}
};

void init(void)   /* charge les mots-cles dans la table */
{
     struct entry *p;
     for (p = keywords; p->token; p++)
        insert(p->lexptr, p->token);
}

/************ main.c **************/

#include "global.h"

int main(void)
{
     init();
     parse();
     return 0;    /*terminaison normale*/
}
```

```
/********** lexer.c ************/

#include "global.h"

int lineno=1;
int tokenval = NONE;
extern char lexemes[];
static char lexbuf[BSIZE];

int lexan(void)   /*analyseur lexical*/
{

 int t;
 while(1) {
    t = getchar();
    if (t == ' ' || t == '\t')
        ; /*sauter les blancs*/
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit(t)) {  /* t est un chiffre*/
      ungetc(t, stdin);
      scanf("%d", &tokenval);
      return NUM;
    }
```

```
else if (isalpha(t)) { /*t est une lettre*/
     int p, b =0;
     while (isalnum(t)){ /*t est alphanum. */
        lexbuf[b] = t;
        t = getchar();
        b = b+1;
        if (b >= BSIZE)
           error("erreur de compilation");
     }
     lexbuf[b] = EOS;
     if (t!= EOF)
         ungetc(t, stdin);
     p = lookup(lexbuf);
     if (p == 0)
          p=insert(lexbuf, ID);
     tokenval = p;
     return symtable[p].token;
  }
  else if (t == EOF)
    return DONE;
  else {
     tokenval = NONE;
     return t;
  }
 }
}
```

```
/********** parser.c ************************/

#include "global.h"
int lookahead;

void parse(void)    /* analyse et traduit la liste */
{                   /* d'expressions*/
   lookahead = lexan();
   while (lookahead != DONE ) {
     expr(); match(';');
   }
}

void expr(void)
{
   int t;
   term();
   while(1)
     switch (lookahead) {
     case '+': case '-':
       t = lookahead;
       match(lookahead); term(); emit(t, NONE);
       continue;
     default:
       return;
     }
}
```

```
void term(void)

    int t;
    factor();
    while(1)
      switch(lookahead) {
      case '*': case '/': case DIV: case MOD:
        t = lookahead;
        match(lookahead); factor(); emit(t, NONE);
        continue;
      default:
        return;
      }
}

void factor(void)
{
    switch(lookahead) {
        case '(':
           match('('); expr(); match(')'); break;
        case NUM:
           emit(NUM, tokenval); match(NUM); break;
        case ID:
           emit(ID, tokenval); match(ID); break;
        default:
           error("syntax error");
    }
}
```

```
void match(int t)
{
     if (lookahead == t)
       lookahead = lexan();
     else error("syntax error");
}



/******** symbol.c ************/

#include "global.h"

#define STRMAX 999  /*taille de la table lexemes*/
#define SYMMAX 100   /* taille de symtable */
struct entry symtable[SYMMAX];
char lexemes[STRMAX];
static int lastchar = -1; /*derniere position */
                  /* utilisee dans lexemes*/
static int lastentry= 0; /*derniere position */
                  /* utilisee dans symtable*/
```

```
int lookup(char s[])   /*retourne la position */
                /* d'une entree pour s */
{
   int p;
   for (p= lastentry; p > 0; p = p-1)
      if (strcmp(symtable[p].lexptr, s) == 0)
          return p;
   return 0;
}

int insert(char s[], int tok)   /*retourne la position */
                    /* d'une entree pour s */
{
    int len;
    len = strlen(s);   /* strlen calcule la */
                       /*  longueur de s */
    if (lastentry + 1 >= SYMMAX)
        error( "table pleine");
    if (lastchar + len + 1 >= STRMAX)
        error("tableau des lexemes plein");
```

```
    lastentry = lastentry + 1;
    symtable[lastentry].token = tok;
    symtable[lastentry].lexptr
                = &lexemes[lastchar + 1];
    lastchar = lastchar + len + 1;
    strcpy(symtable[lastentry].lexptr, s);
    return lastentry;
}


/************* error.c *************/

#include "global.h"

void error(char *m)    /* engendre les messages d'erreur */
{
     fprintf(stderr, "line %d: %s\n", lineno, m);
     exit(1);   /*terminaison anormale*/
}
```

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

```
/*************** emitter.c ****************/

#include "global.h"

void emit(int t, int tval)   /*engendre les sorties*/
{
     switch(t) {
      case '+': case '-': case '*': case '/':
        printf("%c\n",t); break;
      case DIV:
        printf("DIV\n"); break;
      case MOD:
        printf("MOD\n"); break;
      case NUM:
        printf("%d\n", tval); break;
      case ID:
        printf("%s\n", symtable[tval].lexptr);
        break;
      default:
        printf("token %d, tokenval %d\n", t, tval);
     }
}
```