Minicompilateur

1. Syntaxe
2. Traduction dirigée par la syntaxe
3. Analyse syntaxique
4. Analyse lexicale
5. Intégration des techniques

Aperçu

Construction d'un traducteur d'expressions arithmétiques en notation postfixe= code intermédiaire.

On décrit la syntaxe par une *grammaire*.

On emploie la méthode de *traduction dirigée par la syntaxe*.

On aura (dans un deuxième temps) des identificateurs traités dans une table des symboles.

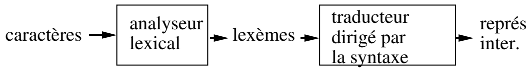

Figure 1: Partie frontale

Syntaxe

On spécifie la syntaxe par une *grammaire*. Une *règle* est de la forme

$$inst \rightarrow \textbf{if } (exp) inst \textbf{ else } inst$$

On aura par exemple une grammaire pour les listes de chiffres séparées par des $+$ ou $-$:

$$\begin{aligned} list &\rightarrow list + chiffre \\ list &\rightarrow list - chiffre \\ list &\rightarrow chiffre \\ chiffre &\rightarrow 0|1|2|3|4|5|6|7|8|9 \end{aligned}$$

ou encore pour les blocs d'instructions:

$$\begin{aligned} bloc &\rightarrow \textbf{begin } opt\_insts \textbf{ end} \\ opt\_insts &\rightarrow inst\_list \mid \epsilon \\ inst\_list &\rightarrow inst\_list;\ inst \end{aligned}$$

Arbre d'analyse

On utilise les grammaires pour construire des arbres d'analyse. Par exemple:

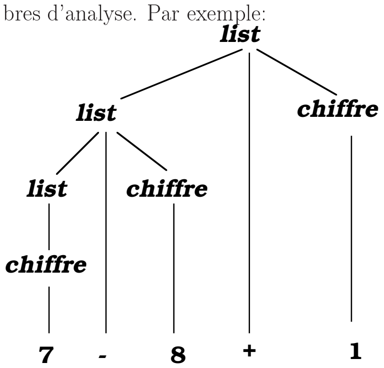

Figure 2: Un arbre d'analyse

Une notion importante est l'ambiguité des grammaires. Par exemple, la grammaire

$$\begin{aligned} string &\rightarrow string + string | string - string \\ &\quad |0|1|2|3|4|5|6|7|8|9 \end{aligned}$$

est ambigue.

Une grammaire non-ambigue

On utilise trois niveaux de priorité pour exprimer

1. L'associativité de gauche à droite
2. La priorité de $*$ et $/$ sur $+$ et $-$.

On aura d'abord

$$factor \rightarrow \textbf{chiffre} \mid (expr)$$

pour les expressions de base. Puis

$$\begin{aligned} term &\rightarrow term * factor \\ &\mid term/factor \\ &\mid factor \end{aligned}$$

pour le deuxième niveau. Et enfin

$$\begin{aligned} expr &\rightarrow expr + term \\ &\mid expr - term \\ &\mid term \end{aligned}$$

pour le troisième.

Traduction dirigée par la syntaxe

On calcule des *attributs* sémantiques associés aux constructions syntaxiques. Par exemple: la valeur d'une expression, l'adresse d'une variable,... Le calcul se fait sur l'arbre d'analyse.

Les attributs peuvent être:

1. synthétisés: calculés en remontant dans l'arbre d'analyse.
2. hérités: en descendant (ou en allant horizontalement)

Attributs synthétisés

Calcul de la valeur d'une expression:

REGLE	ACTION
$L \rightarrow E$ **n**	$print(E.val)$
$E \rightarrow E + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T * F$	$T.val := T_1.val \times F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$**chiffre**	$F.val :=$**chiffre**$.lexval$

Schémas de traduction

On ajoute aux règles de la grammaire des *actions* qui calculent les attributs. On fixe de plus un ordre de visite de l'arbre d'analyse: l'ordre de l'exploration en profondeur (*'depth-first search'*).

Par exemple, le schéma:

$$\begin{array}{lll} expr \rightarrow & expr + term & \{print('+')\} \\ expr \rightarrow & expr - term & \{print('-')\} \\ expr \rightarrow & term & \\ term \rightarrow & 0 & \{print('0')\} \\ term \rightarrow & 1 & \{print('1')\} \\ \ldots & & \\ term \rightarrow & 9 & \{print('9')\} \end{array}$$

effectue la traduction en forme suffixe.

Analyse syntaxique

L'analyse syntaxique est la construction de l'arbre d'analyse à partir de la suite de symboles. Il existe deux grandes classes de méthodes:

1. descendante: c'est la méthode la plus facile.
2. ascendante: permet de traiter plus de cas.

Principe de la méthode descendante (*'top-down-parsing'* ou *'recursive descent'*):

1. Associer une fonction à chaque non-terminal de la grammaire.
2. Utiliser une variable globale pour explorer le texte.
3. Executer les productions de la grammaire.

Premier programme

Réalise la traduction infixe/suffixe limitée à des expressions additives.

Les lexèmes sont constitués d'un seul caractère et donnés par la fonction standard **`getchar`**.

La fonction **`match`** vérifie les lexèmes et lit le suivant. Elle appelle **`error`** si la lecture est incorrecte.

```
#include <ctype.h> /*charge isdigit */
int lookahead;

main()
{
   lookahead = getchar();
   expr();
   putchar('\n'); /* ajoute newline */
}
```


Les expressions

On transforme une expression de la forme

$$term + term + \ldots + term$$

en

$$term\; term\; + \ldots term +$$

d'où le programme:

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


Les termes

La reste est de l'analyse lexicale:

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

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

Traitement des erreurs

Il est réduit à la plus simple expression:

```
error()
{
   printf("syntax error\n");
      /*ecrit un message d'erreur*/
   exit(1);     /*arrete l'execution */
}
```


Analyse lexicale

On souhaite éliminer les blancs, lire des constantes, des identificateurs.

Paire *producteur-consommateur* avec l'analyseur syntaxique.

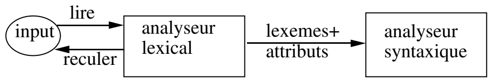

Figure 3: Insertion de l'analyse lexicale

L'analyseur lexical passe le couple du type de lexème et de son attribut à l'analyseur syntaxique. Ainsi

$$12 + 45 - 8$$

est transformé en

$$< \mathbf{num}, 12 >< +, >< \mathbf{num}, 45 >< -, >< \mathbf{num}, 8 >$$

Implémentation

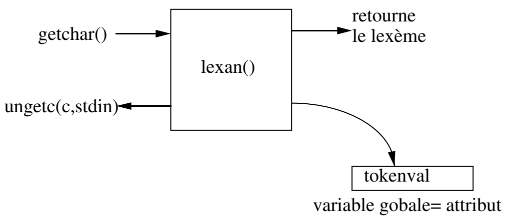

Figure 4: Interactions

Le type du lexème est un entier représenté comme une constante symbolique:

```
#define NUM 256
```


Programme

Code C pour éliminer les blancs et rassembler les nombres.

```
#include <stdio.h>
#include <ctype.h>

int lineno = 1;
int tokenval = NONE;

int lexan()   /*analyseur lexical*/
{

 int t;
 while(1) {
    t = getchar();
    if (t == ' ' || t == '\t')
        ; /*effacer les blancs*/
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit(t)) {  /* t est un chiffre*/
      tokenval = t - '0';
      t= getchar();
      while (isdigit(t)) {
         tokenval = tokenval*10 + t-'0';
```

```
         t = getchar();
      }
      ungetc(t,stdin);
      return NUM;
    }
    else {
       tokenval = NONE;
       return t;
    }
  }
}
```


Ajout d'une table des symboles

La table des symboles utilise deux fonctions:

`insert(s,t)`:retourne une nouvelle entrée pour la chaine s, correspondant au lexème t.

`lookup(s)`: retoune un indice pour la chaine s ou 0 si s n'apparait pas.

On peut ainsi traiter les *mots clé* :

`insert("div",` **div**)

L'analyseur utilise un tampon `lexbuf`

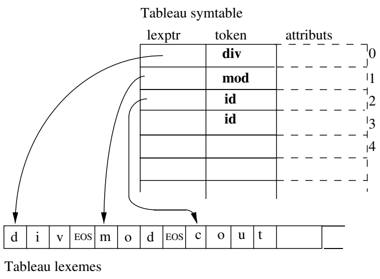

Figure 5: Structure de données

Schema

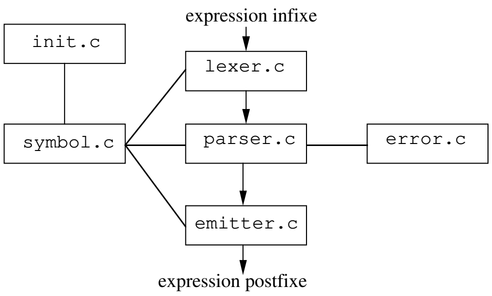

Figure 6: Modules du traducteur

Le programme

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

#include "global.h"

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

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

#include "global.h"

main()
{
     init();
     parse();
```

```
    exit(0);    /*terminaison normale*/
}


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

#include "global.h"

char lexbuf[BSIZE];
int  lineno=1;
int tokenval = NONE;
extern char lexemes[];
int lexan()   /*analyseur lexical*/
{

 int t;
 while(1) {
    t = getchar();
    if (t == ' ' || t == '\t')
        ; /*effacer 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 i,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;

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

expr()
{
   int t;
   term();
   while(1)
     switch (lookahead) {
     case '+': case '-':
```



```
       t = lookahead;
       match(lookahead); term(); emit(t, NONE);
       continue;
     default:
       return;
     }
}

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

factor()
{
```

```
     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");
     }
}

match(t)
     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*/
```



```
char lexemes[STRMAX];
int lastchar = -1; /*derniere position */
                  /* utilisee dans lexemes*/
struct entry symtable[SYMMAX];
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 positi
                    /* 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"

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

#include "global.h"
```



```
emit(t, tval)   /*engendre l'output*/
     int t, tval;
{
     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);
     }
}
```