Prädiktiver Parser (Prinzip)

```
token lookahead;

match(token t)
{
  if (lookahead==t)
    lookahead = nexttoken();
  else
   error ("Syntax error");

}/*match*/

type()
{
  switch(lookahead) {

  case INTEGER:
  case CHAR:
  case NUM:
    simple();
    break;

  case PTR:
    match(PTR); /* '^' */
    match(ID);
    break;

  case ARRAY:
    match(ARRAY);
    match(LB);  /* '[' */
    simple();
    match(RB);  /* ']' */
    match(OF);
    type();
    break;

  default:
    error ("Syntax error");
    break;
  }

}/*type*/
```

```
simple()
{

  if (lookahead==INTEGER)
    match(INTEGER);
  else if (lookahead==CHAR)
    match(CHAR);
  else if (lookahead==NUM) {
    match(NUM);
    match(DDOT);  /* '..' */
    match(NUM);
  }
  else error ("Syntax error");

}/*simple*/
```

Übersetzung einfacher Infix-Ausdrücke (1)

Produktion: E $\rightarrow$ TR

```
expr()
{
  term();
  rest();

}/*expr*/
```

Produktion: R $\rightarrow$ + T {print('+')} R | **-** T {print('**-**')} R | ε

```
rest()
{
  if (lookahead=='+') {
    match('+');
    term();
    putchar('+');
    rest();
  }
  else if (lookahead=='-') {
    match('-');
    term();
    putchar('-');
    rest();
  }

}/*rest*/
```

Produktion: T $\rightarrow$ **0** {print('**0**')} | ... | **9** {print('**9**')}

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

}/*term*/
```

Übersetzung einfacher Infix-Ausdrücke (2)

Laufzeitverbesserung für `rest()`

```
rest()
{

L:
  if (lookahead=='+') {
    match('+');
    term();
    putchar('+');
    goto L;
  }
  else if (lookahead=='-') {
    match('-');
    term();
    putchar('-');
    goto L;
  }
  else ;

}/*rest*/
```

Übersetzung einfacher Infix-Ausdrücke (3)

Zusammenfassung von `expr()` und `rest()`

```
expr()
{

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

}/*expr*/
```

Rekursiver-Abstieg-Compiler (1)

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

int lookahead;

main()
{

  lookahead = getchar();
  expr();
  putchar('\n');

}/*main*/


expr()
{

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

}/*expr*/
```

Rekursiver-Abstieg-Compiler (2)

```
term()
{

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

}/*term*/


match(int t)
{

  if (lookahead==t)
    lookahead = getchar();
  else
    error();

}/*match*/


error()
{

  printf("Syntax error\n");
  exit(1);

}/*error*/
```

Scanner: Erkennung von Zahlen

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

#define NONE -1
#define NUM 256

int  lineno = 1;
int  tokenval = NONE;

int lexan()
{
  int t;

  while(1) {
    t = getchar();
    if (t==' ' ||t=='\t') {
      /* Eliminate blanks and tab skips */
      ;
    }
    else if (t=='\n') {
      /* Increment linenumber */
      lineno++;
    }
    else if (isdigit(t)) {
      tokenval = t - '0';
      t = getchar();
      while (isdigit(t)) {
        tokenval = tokenval*10 + t - '0':
        t = getchar();
      }
      ungetc(t,stdin);
      return NUM;
    }
    else {
      /* No token recognized */
      tokenval= NONE;
      return t;
    }
  }

}/*lexan*/
```

Infix-Postfix-Compiler

```
/*=====================================================*/
/* 1-Pass-Compiler Infix -> Postfix                    */
/*=====================================================*/

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

/*--- Token types ---*/

#define NONE  -1

#define NUM   256
#define DIV   257
#define MOD   258
#define ID    259
#define SEM   260
#define DONE  261

/*--- Symbol table ---*/

/* Size of table */
#define SYMMAX 100

/* Table entry */
struct entry {
     char *lexptr;
     int  token ;
};

/* Table */
struct entry symtable [SYMMAX];

/* Last used table entry */
int  lastentry;

/* Reserved words */
struct entry keywords[] = {
  "div",DIV,
  "mod",MOD,
  0,0
};
```

```
/*--- Lexem array ---*/

/* End of string */
#define EOS   '\0'

/* Size of lexem array */
#define STRMAX 999

/* Lexem array */
char lexemes[STRMAX];

/* Last used lexem position */
int  lastchar;

/*--- Token buffer ---*/

/* Size of buffer */
#define BSIZE 128

/* Buffer */
char lexbuf[BSIZE];

/*========= Management of symbol table =========*/

/* Initialize symbol table */
init()
{
  struct entry *p;

  lastentry = 0;
  lastchar = -1;
  for (p=keywords; p->token; p++)
          insert(p->lexptr, p->token);

}/*init*/
```

```
/* Check whether symbol is in table */
int
lookup(char s[])
{
  int p;

  for (p=lastentry; p>0; p=p-1) {
    if (strcmp(symtable[p].lexptr,s)==0) {
      return p;
    }
  }

  return 0;

}/*lookup*/

/* Insert new symbol into table */
int
insert(char s[], int tok)
{
  int len;

  len = strlen(s);

  if (lastentry+1>=SYMMAX) {
    error ("Symbol table overflow!");
  }

  if (lastchar+len+1>=STRMAX) {
    error ("Lexem array overflow!");
  }

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

  return lastentry;

}/*insert*/
```

```
/*================ Scanner ======================*/

/* Lookahead symbol */
int lookahead;

/* Line number */
int lineno;

/* Token value */
int tokenval;

/* Get next token */
int lexan()
{
  int t;

  while(1) {
    t = getchar();
    if (t==' ' ||t=='\t') {
      /* Eliminate blanks */
      ;
    }
    else if (t=='\n') {
      /* Increment linenumber */
      lineno++;
    }
    else if (isdigit(t)) {
      /* t is digit */
      ungetc(t,stdin);
      scanf("%d", &tokenval);
      return NUM;
    }
    else if (isalpha(t)) {
      /* t is letter */
      int p, b = 0;
      while (isalnum(t)) {  /* t is alphanumeric */
        lexbuf[b] = t;
        t = getchar();
        b++;
        if (b >= BSIZE) {
          error("Compiler error!");
        }
      }
```

```
      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) {
      /* End of file */
      return DONE;
    }
    else if (t==';') {
      /* End of statement */
      tokenval= SEM;
      return t;
    }
    else {
      /* No token recognized */
      tokenval= NONE;
      return t;
    }
  }

}/*lexan*/

/* Match token and get next one */
match(int t)
{
  if (lookahead==t) {
    lookahead =lexan();
  }
  else
    error ("Syntax error");

}/*match*/
```

```
/*================== Parser =====================*/

/* Parse input */
parse()
{
  lookahead = lexan();
  while (lookahead!=DONE) {
    expr();
    match (';');
    printf("\n");
  }

}/*parse*/

/* Parse nonterminal expr */
expr()
{
  int t;

  term();
  while(1) {
    switch(lookahead) {
    case '+':
    case '-':
      t = lookahead;
      match(lookahead);
      term();
      emit(t,NONE);
      break;
    default:
      return;
    }
  }

}/*expr*/
```

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

/* Parse nonterminal factor */
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");
  }
}/*factor*/
```

```
/*============ Compiler output ===================*/

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

}/*emit*/

/*=============== Error messages =================*/

error(char *m)
{

  printf("ERROR: line %d:%s\n", lineno, m);
  printf("\nInterrupted.\n");
  exit(1);

}/*error*/
```

```
/*=============== Main program ===================*/

main()
{
  lineno = 1;
  tokenval = NONE;

  printf("\nInfix-Postfix-Compiler");
  printf("\n----------------------\n\n");

  init();
  parse();

  printf("\nTerminated.\n");
  exit(0);

}/*main*/
```

Beispiel

```
(9+33) mod 4 -5+xx-7+8-9+7;
88*(a*br);
88*a*br;
8-7*8 div 3;
8-((7*8) div 3);
7*((4+xx)-abc1) mod 4;
7 MOD 3;
```

Programmausgabe

```
Infix-Postfix-Compiler
----------------------

9 33 + 4 MOD 5 - xx + 7 - 8 + 9 - 7 +
88 a br * *
88 a * br *
8 7 8 * 3 DIV -
8 7 8 * 3 DIV -
7 4 xx + abc1 - * 4 MOD
7 ERROR: line 6:Syntax error

Interrupted.
```