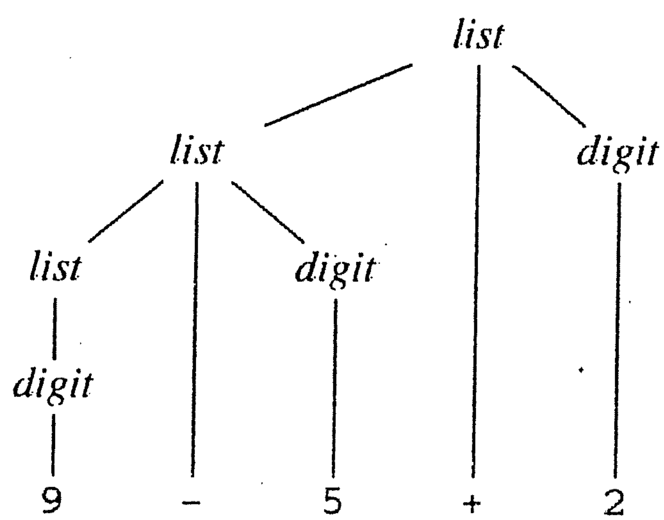

Fig. 2.5 Parse tree for 9-5+2 according to the grammar in Example 2.1.

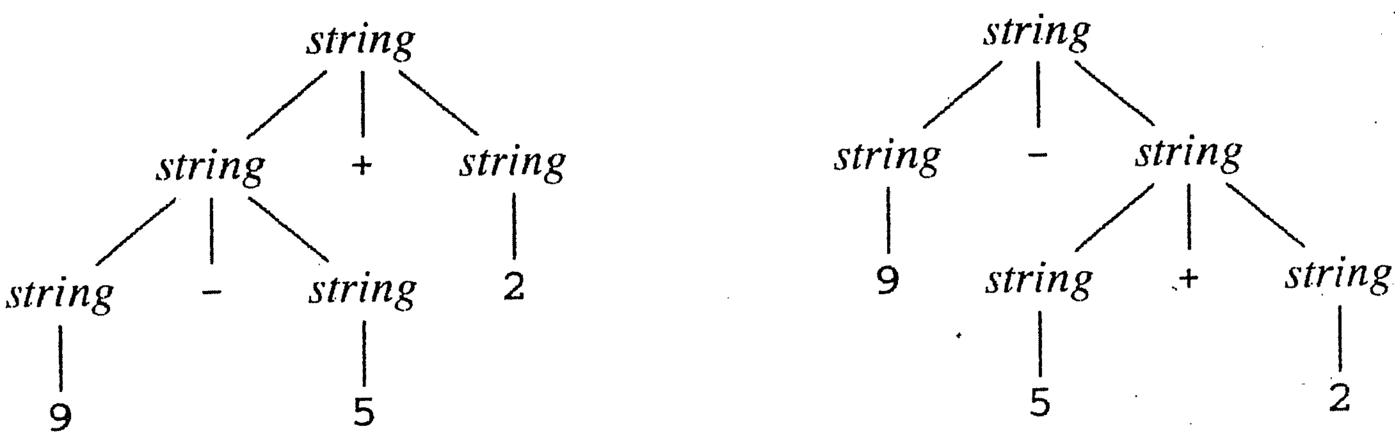

Fig. 2.6. Two parse trees for 9-5+2.

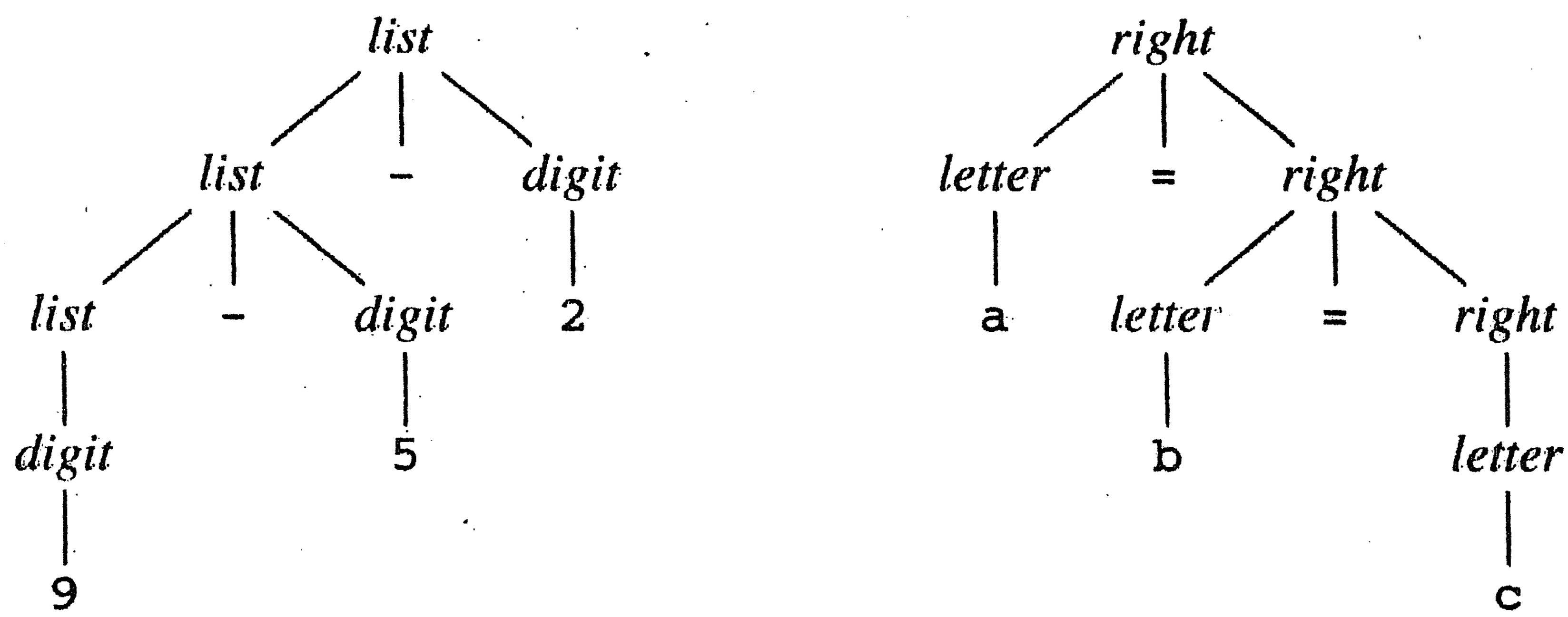

Fig. 2. 7. Parse trees for left- and right-associative operators.

Production	Semantic Rule
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \,\|\, term.t \,\|\, '+'$
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \,\|\, term.t \,\|\, '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
$\cdots$	$\cdots$
$term \rightarrow 9$	$term.t := '9'$

Fig. 2.10. Syntax-directed definition for infix to postfix translation.

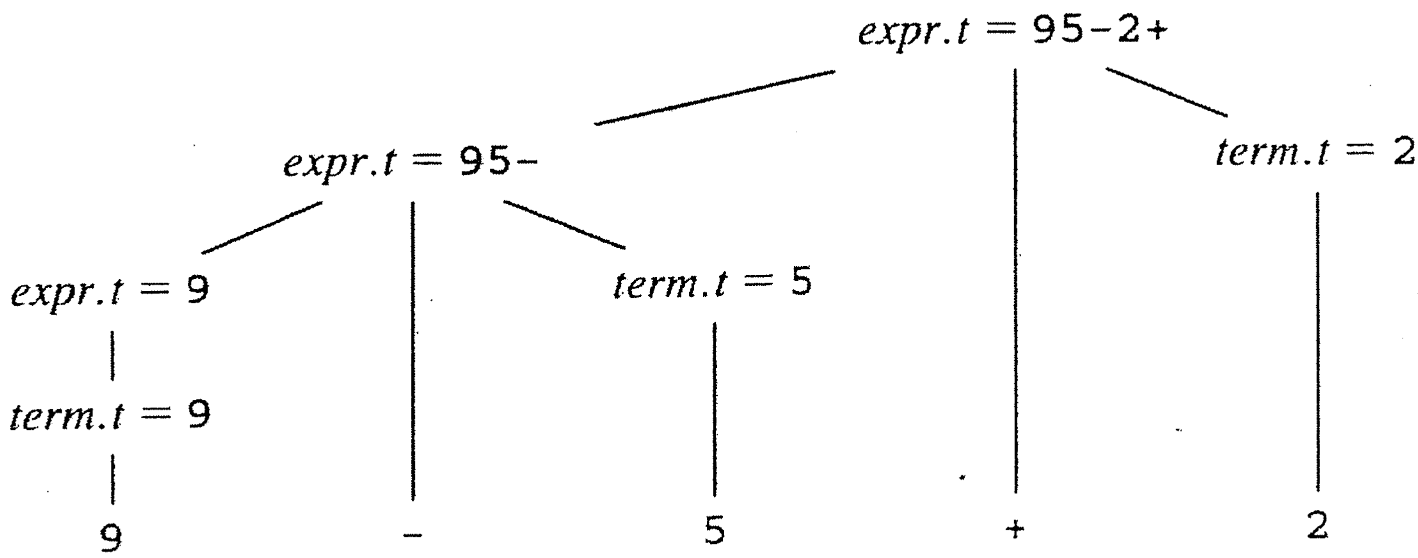

Fig. 2.9. Attribute values at nodes in a parse tree.

PRODUCTION	SEMANTIC RULES
$seq \rightarrow$ **begin**	$seq.x := 0$ $seq.y := 0$
$seq \rightarrow seq_1\ instr$	$seq.x := seq_1.x + instr.dx$ $seq.y := seq_1.y + instr.dy$
$instr \rightarrow$ **east**	$instr.dx := 1$ $instr.dy := 0$
$instr \rightarrow$ **north**	$instr.dx := 0$ $instr.dy := 1$
$instr \rightarrow$ **west**	$instr.dx := -1$ $instr.dy := 0$
$instr \rightarrow$ **south**	$instr.dx := 0$ $instr.dy := -1$

Fig. A. Syntax-directed definition of the robot's position.

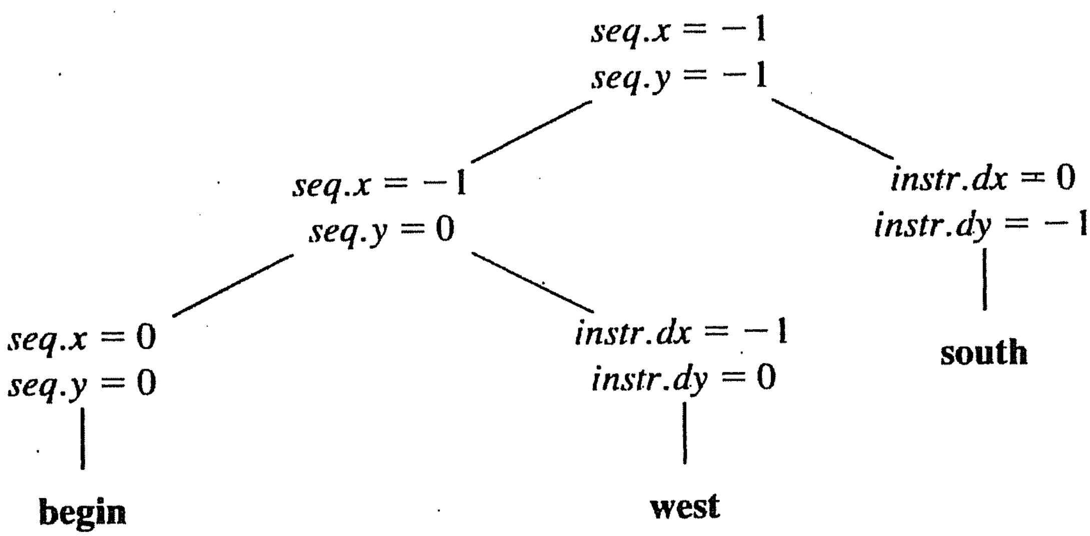

Fig. B. Annotated parse tree for **begin west south**.

procedure *visit* (n : node);
begin
    for each child m of n, from left to right **do**
        visit (m);
    evaluate semantic rules at node n
end

Fig. 2.11. A depth-first traversal of a tree.

expr	$\rightarrow$	*expr*	+	*term*	{ *print*('+') }
expr	$\rightarrow$	*expr*	−	*term*	{ *print*('−') }
expr	$\rightarrow$	*term*			
term	$\rightarrow$	0			{ *print*('0') }
term	$\rightarrow$	1			{ *print*('1') }
	. . .				
term	$\rightarrow$	9			{ *print*('9') }

Fig. 2.15. Actions translating expressions into postfix notation.

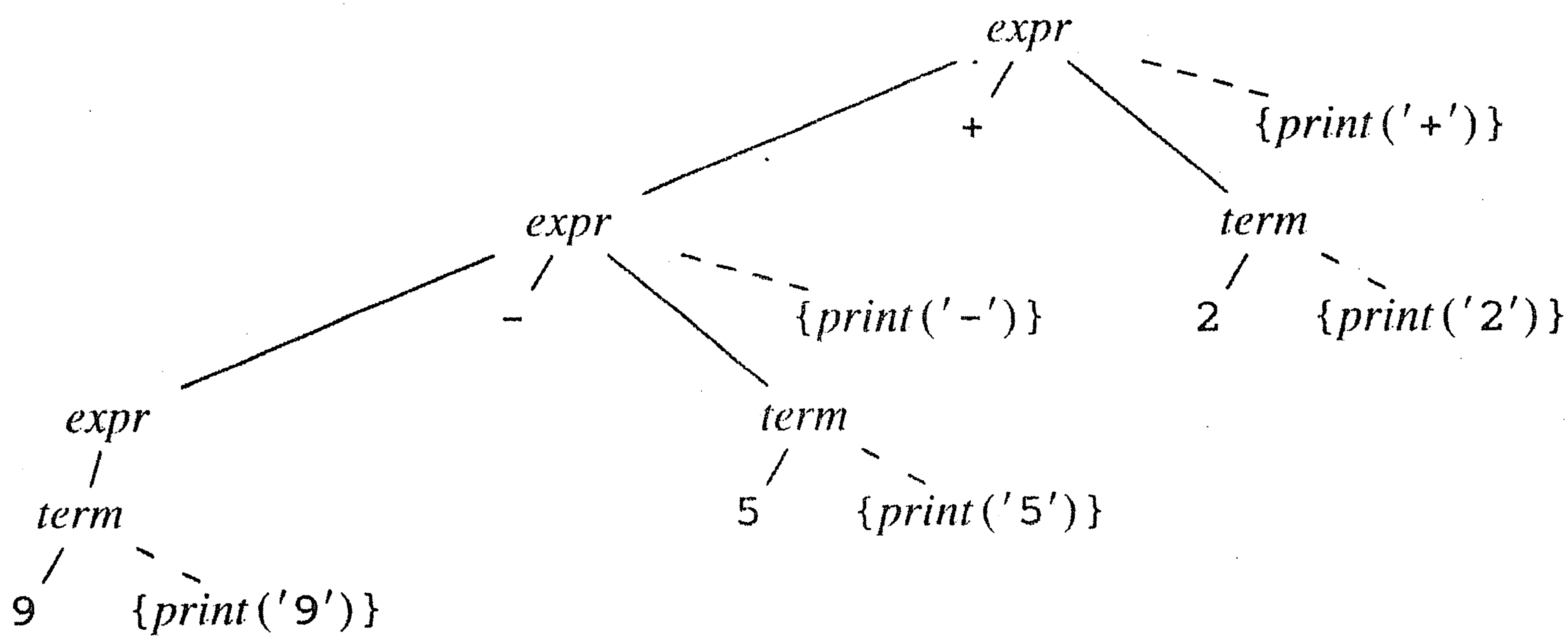

Fig. 2.14. Actions translating 9-5+2 into 95-2+.

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{expr} \; ; \\
 & | & \textbf{if} \; (\; \textbf{expr} \;) \; \mathit{stmt} \\
 & | & \textbf{for} \; (\; \mathit{optexpr} \; ; \; \mathit{optexpr} \; ; \; \mathit{optexpr} \;) \; \mathit{stmt} \\
 & | & \textbf{other} \\
 & & \\
\mathit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

Figure 2.16: A grammar for some statements in C and Java

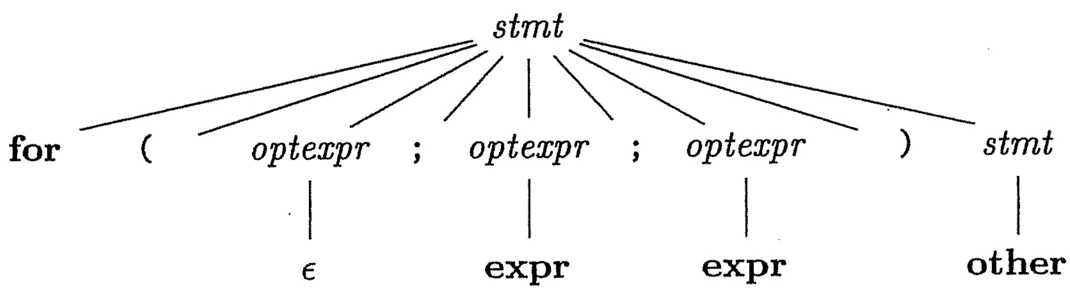

Figure 2.17: A parse tree according to the grammar in Fig. 2.16

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{expr} \ ; \\
 & | & \textbf{if} \ (\ \textbf{expr} \ ) \ \mathit{stmt} \\
 & | & \textbf{for} \ (\ \mathit{optexpr} \ ; \ \mathit{optexpr} \ ; \ \mathit{optexpr} \ ) \ \mathit{stmt} \\
 & | & \textbf{other} \\
 & & \\
\mathit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

Figure 2.16: A grammar for some statements in C and Java

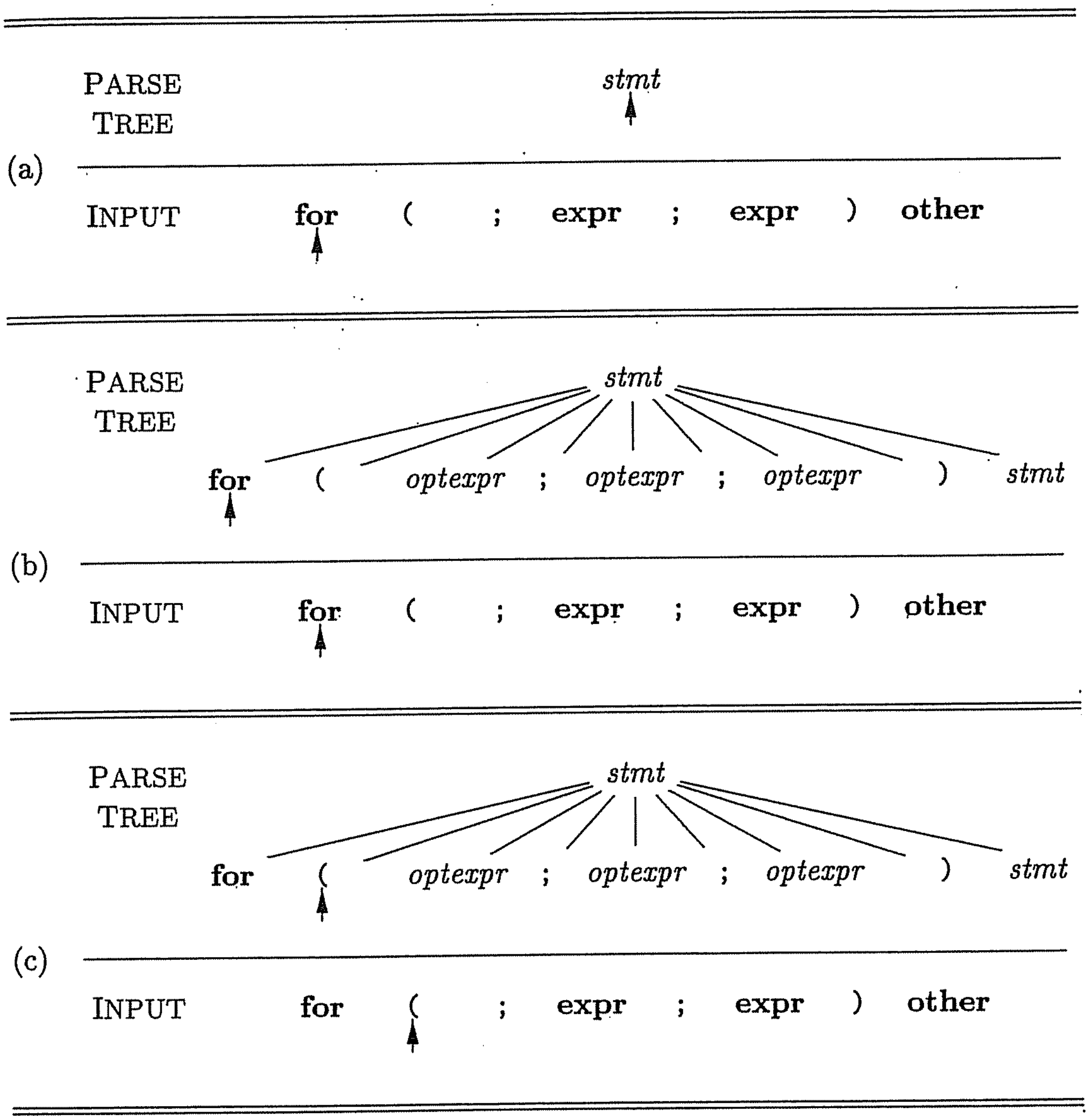

Figure 2.18: Top-down parsing while scanning the input from left to right

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

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

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

Fig. C. Functions for the nonterminals *expr*, *rest*, and *term*.

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

Fig. D.

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

Fig. E. Replacement for functions expr and rest of Fig.

```
#include <ctype.h> /* loads file with predicate isdigit */
int lookahead;

main()
{
     lookahead = getchar();
     expr();
     putchar('\n'); /* adds trailing newline character */
}

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

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

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

error()
{
     printf("syntax error\n"); /* print error message */
     exit(1);      /* then halt */
}
```

Fig. F. C program to translate an infix expression into postfix form.

```
#include <stdio.h>
#include <ctype.h>
int  lineno = 1;
int  tokenval = NONE;

int  lexan()
{
        int t;
        while(1) {
            t = getchar();
            if (t == ' ' || t == '\t')
                    ;     /* strip out blanks and tabs */
            else if (t == '\n')
                    lineno = lineno + 1;
            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 {
                    tokenval = NONE;
                    return t;
            }
        }
}
```

Fig. 6. C code for lexical analyzer eliminating white space and collecting num

Example 2.15: The following pseudocode uses subscripts to distinguish among distinct declarations of the same identifier:

$\{\quad$ **int** x_1; **int** y_1;
$\quad\quad\{\quad$ **int** w_2; **bool** y_2; **int** z_2;
$\quad\quad\quad\quad \cdots w_2 \cdots; \cdots x_1 \cdots; \cdots y_2 \cdots; \cdots z_2 \cdots;$
$\quad\quad\}$
$\quad\quad \cdots w_0 \cdots; \cdots x_1 \cdots; \cdots y_1 \cdots;$
$\}$

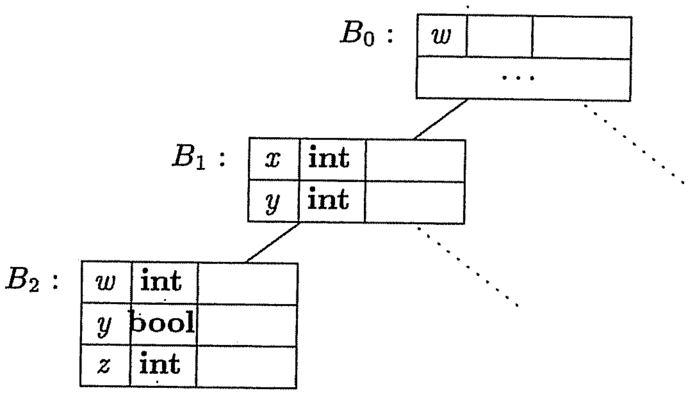

Figure 2.36: Chained symbol tables for Example 2.15

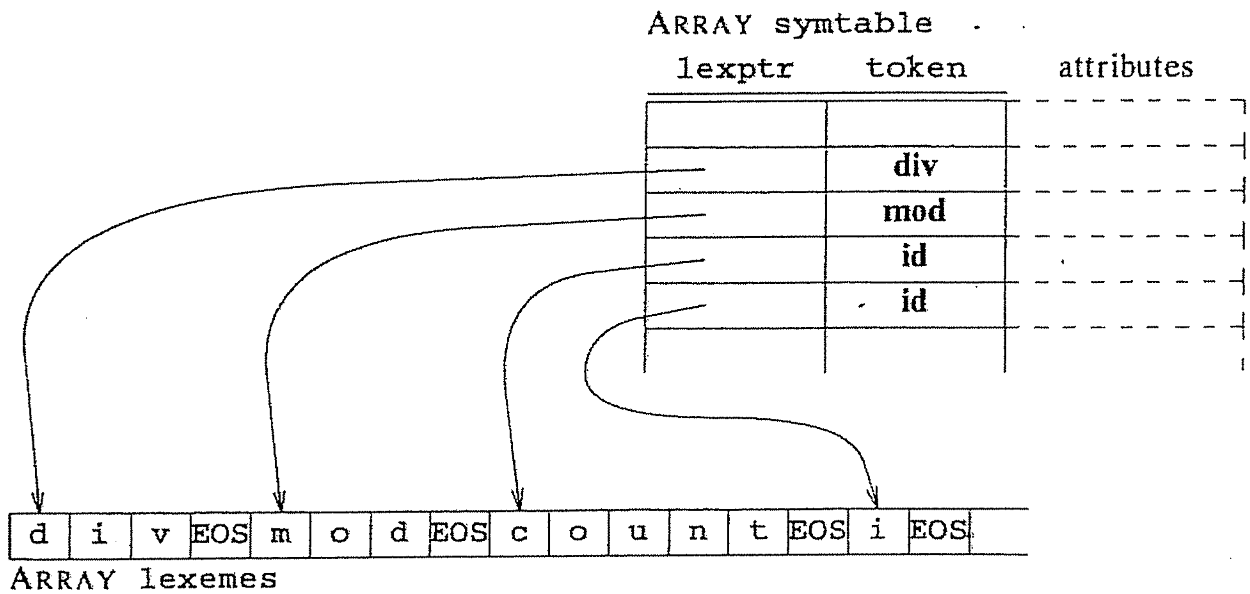

Fig. H. Symbol table and array for storing strings.

```
function lexan: integer;
var     lexbuf :    array [0..100] of char;
        c :         char;
begin
        loop begin
                read a character into c;
                if c is a blank or a tab then
                        do nothing
                else if c is a newline then
                        lineno := lineno + 1
                else if c is a digit then begin
                        set tokenval to the value of this and following digits;
                        return NUM
                end
                else if c is a letter then begin
                        place c and successive letters and digits into lexbuf;
                        p := lookup(lexbuf);
                        if p = 0 then
                                p := insert(lexbuf, ID);
                        tokenval := p;
                        return the token field of table entry p
                end
                else begin   /* token is a single character */
                        set tokenval to NONE;      /* there is no attribute */
                        return integer encoding of character c
                end
        end
end
```

Fig. I. Pseudo-code for a lexical analyzer.

$$
\begin{array}{rcll}
\mathit{program} & \rightarrow & \mathit{block} & \{\ \text{return } \mathit{block.n};\ \} \\[6pt]
\mathit{block} & \rightarrow & '\{'\ \mathit{stmts}\ '\}' & \{\ \mathit{block.n} = \mathit{stmts.n};\ \} \\[6pt]
\mathit{stmts} & \rightarrow & \mathit{stmts}_1\ \mathit{stmt} & \{\ \mathit{stmts.n} = \textbf{new}\ \mathit{Seq}(\mathit{stmts}_1.n, \mathit{stmt.n});\ \} \\
 & | & \epsilon & \{\ \mathit{stmts.n} = \textbf{null};\ \} \\[6pt]
\mathit{stmt} & \rightarrow & \mathit{expr}\ ; & \{\ \mathit{stmt.n} = \textbf{new}\ \mathit{Eval}(\mathit{expr.n});\ \} \\
 & | & \textbf{if}\ (\ \mathit{expr}\ )\ \mathit{stmt}_1 & \\
 & & & \{\ \mathit{stmt.n} = \textbf{new}\ \mathit{If}(\mathit{expr.n}, \mathit{stmt}_1.n);\ \} \\
 & | & \textbf{while}\ (\ \mathit{expr}\ )\ \mathit{stmt}_1 & \\
 & & & \{\ \mathit{stmt.n} = \textbf{new}\ \mathit{While}(\mathit{expr.n}, \mathit{stmt}_1.n);\ \} \\
 & | & \textbf{do}\ \mathit{stmt}_1\ \textbf{while}\ (\ \mathit{expr}\ ); & \\
 & & & \{\ \mathit{stmt.n} = \textbf{new}\ \mathit{Do}(\mathit{stmt}_1.n, \mathit{expr.n});\ \} \\
 & | & \mathit{block} & \{\ \mathit{stmt.n} = \mathit{block.n};\ \} \\[6pt]
\mathit{expr} & \rightarrow & \mathit{rel} = \mathit{expr}_1 & \{\ \mathit{expr.n} = \textbf{new}\ \mathit{Assign}('=', \mathit{rel.n}, \mathit{expr}_1.n);\ \} \\
 & | & \mathit{rel} & \{\ \mathit{expr.n} = \mathit{rel.n};\ \} \\[6pt]
\mathit{rel} & \rightarrow & \mathit{rel}_1 < \mathit{add} & \{\ \mathit{rel.n} = \textbf{new}\ \mathit{Rel}('<', \mathit{rel}_1.n, \mathit{add.n});\ \} \\
 & | & \mathit{rel}_1 <= \mathit{add} & \{\ \mathit{rel.n} = \textbf{new}\ \mathit{Rel}('\leq', \mathit{rel}_1.n, \mathit{add.n});\ \} \\
 & | & \mathit{add} & \{\ \mathit{rel.n} = \mathit{add.n};\ \} \\[6pt]
\mathit{add} & \rightarrow & \mathit{add}_1 + \mathit{term} & \{\ \mathit{add.n} = \textbf{new}\ \mathit{Op}('+', \mathit{add}_1.n, \mathit{term.n});\ \} \\
 & | & \mathit{term} & \{\ \mathit{add.n} = \mathit{term.n};\ \} \\[6pt]
\mathit{term} & \rightarrow & \mathit{term}_1 * \mathit{factor} & \{\ \mathit{term.n} = \textbf{new}\ \mathit{Op}('*', \mathit{term}_1.n, \mathit{factor.n});\ \} \\
 & | & \mathit{factor} & \{\ \mathit{term.n} = \mathit{factor.n};\ \} \\[6pt]
\mathit{factor} & \rightarrow & (\ \mathit{expr}\ ) & \{\ \mathit{factor.n} = \mathit{expr.n};\ \} \\
 & | & \textbf{num} & \{\ \mathit{factor.n} = \textbf{new}\ \mathit{Num}(\textbf{num}.\mathit{value});\ \}
\end{array}
$$

Figure 2.39: Construction of syntax trees for expressions and statements

$$
\begin{array}{rcll}
\textit{start} & \rightarrow & \textit{list}\ \ \textbf{eof} & \\
\textit{list} & \rightarrow & \textit{expr}\ \ ;\ \ \textit{list} & \\
 & | & \epsilon & \\
\textit{expr} & \rightarrow & \textit{expr}\ \ +\ \ \textit{term} & \{\ \textit{print}('+')\ \} \\
 & | & \textit{expr}\ \ -\ \ \textit{term} & \{\ \textit{print}('-')\ \} \\
 & | & \textit{term} & \\
\textit{term} & \rightarrow & \textit{term}\ \ *\ \ \textit{factor} & \{\ \textit{print}('*')\ \} \\
 & | & \textit{term}\ \ /\ \ \textit{factor} & \{\ \textit{print}('/')\ \} \\
 & | & \textit{term}\ \ \textbf{div}\ \ \textit{factor} & \{\ \textit{print}(\texttt{'DIV'})\ \} \\
 & | & \textit{term}\ \ \textbf{mod}\ \ \textit{factor} & \{\ \textit{print}(\texttt{'MOD'})\ \} \\
 & | & \textit{factor} & \\
\textit{factor} & \rightarrow & (\ \ \textit{expr}\ \ ) & \\
 & | & \textbf{id} & \{\ \textit{print}(\textbf{id}.\textit{lexeme})\ \} \\
 & | & \textbf{num} & \{\ \textit{print}(\textbf{num}.\textit{value})\ \}
\end{array}
$$

Fig. 2.35. Specification for infix-to-postfix translator.

$$
\begin{array}{rcl}
\mathit{start} & \rightarrow & \mathit{list}\ \textbf{eof} \\
\mathit{list} & \rightarrow & \mathit{expr}\ \ ;\ \ \mathit{list} \\
& | & \epsilon \\
\mathit{expr} & \rightarrow & \mathit{term}\ \ \mathit{moreterms} \\
\mathit{moreterms} & \rightarrow & +\ \ \mathit{term}\ \ \{\ \mathit{print}('+')\ \}\ \ \mathit{moreterms} \\
& | & -\ \ \mathit{term}\ \ \{\ \mathit{print}('-')\ \}\ \ \mathit{moreterms} \\
& | & \epsilon \\
\mathit{term} & \rightarrow & \mathit{factor}\ \ \mathit{morefactors} \\
\mathit{morefactors} & \rightarrow & *\ \ \mathit{factor}\ \ \{\ \mathit{print}('*')\ \}\ \ \mathit{morefactors} \\
& | & /\ \ \mathit{factor}\ \ \{\ \mathit{print}('/')\ \}\ \ \mathit{morefactors} \\
& | & \textbf{div}\ \ \mathit{factor}\ \ \{\ \mathit{print}(\texttt{'DIV'})\ \}\ \ \mathit{morefactors} \\
& | & \textbf{mod}\ \ \mathit{factor}\ \ \{\ \mathit{print}(\texttt{'MOD'})\ \}\ \ \mathit{morefactors} \\
& | & \epsilon \\
\mathit{factor} & \rightarrow & (\ \ \mathit{expr}\ \ ) \\
& | & \textbf{id}\ \ \{\ \mathit{print}(\textbf{id}.\mathit{lexeme})\ \} \\
& | & \textbf{num}\ \ \{\ \mathit{print}(\textbf{num}.\mathit{value})\ \}
\end{array}
$$

Fig. 2.38. Syntax-directed translation scheme after eliminating left-recursion.

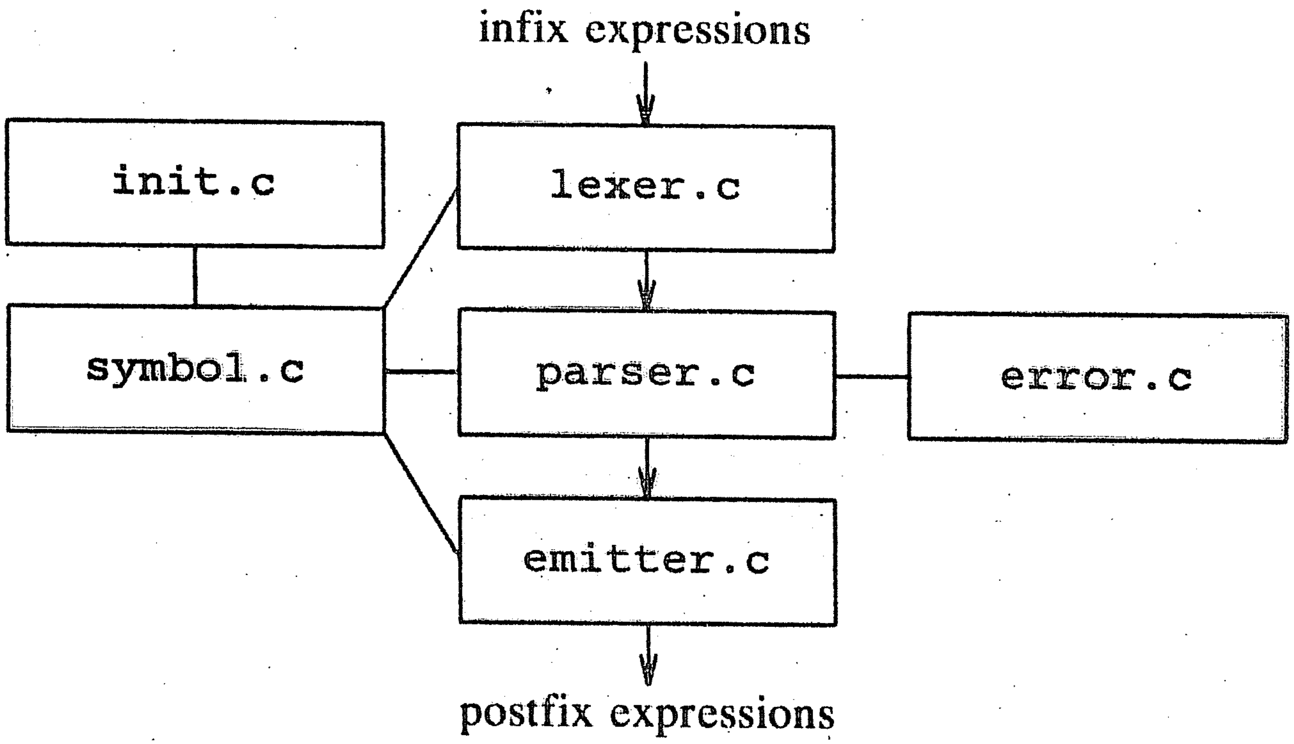

Fig. 2.36. Modules of infix-to-postfix translator.