```
/* ---------------------------------------------------------------
   LALR(1) grammar to parse:  S -> C C '\n'    C -> 'd' | 'c' C
--------------------------------------------------------------- */
%{
  #define YYSTYPE double  /* YYSTYPE is the type of yyval */
  #include <math.h>
  #include <stdio.h>
  int yylex (void);
  void yyerror (char const *);
%}

/* Bison declarations.  */

%% /* The grammar follows.  */

S:         C  C  '\n'  { printf ("   result: %.5g\n", $1 + $2); }
;

C:        'd'          { $$ = 1;      }
        | 'c'  C       { $$ = 1 + $2; }
;
%%
/* ----- Lexical Analyzer ------ */
/* The lexical analyzer returns the numeric code
   of the character read.  It skips all blanks
   and tabs, and returns 0 for end-of-input.  */

#include <ctype.h>

int yylex (void)
{
  int c;

  /* Skip white spaces.  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* Return end-of-input.  */
  if (c == EOF)
    return 0;
  /* Return a single char.  */
   return c;
}
/* ----- ERROR Routine ------ */

#include <stdio.h>

/* Called by yyparse on error.  */

void yyerror (char const *s)
{
  printf ("%s\n", s);
}

/* ----- MAIN program ------ */

int main (void)
{
  return yyparse ();
}
/* ------------------------------------------------------------------
The grammar:          S ->  C  C '\n'
                      C -> 'c' C | 'd
is an LALR(1) grammar. The initial lookahead symbol is '\n', not '$'.
---------------------------------------------------------------------
$ bison -g g.y
$ cc -o g g.tab.c
$ g
ccdcccd
   result: 7
```

```
$ g
cdcdd
syntax error
-------------------------------
The option -g produces a VCG graph description of the parse table.
-------------------------------------------------------------------- */

Graph corresponding to the LALR(1) parse table:
-----------------------------------------------

graph: {
     display_edge_labels:     yes
     priority_phase:     yes
     port_sharing:     no
     crossing_weight:     median
     splines:     yes
     node: {
          title:     "0"
          label:     "state  0\n $accept -> . S $end"
     }
     edge: {
          sourcename:     "0"
          targetname:     "1"
          label:     "'d'"
          color:     blue
          textcolor:     black
     }
     edge: {
          sourcename:     "0"
          targetname:     "2"
          label:     "'c'"
          color:     blue
          textcolor:     black
     }
     edge: {
          sourcename:     "0"
          targetname:     "3"
          label:     "S"
          color:     green
          textcolor:     black
     }
     edge: {
          sourcename:     "0"
          targetname:     "4"
          label:     "C"
          color:     green
          textcolor:     black
     }
     node: {
          title:     "1"
          label:     "state  1\n C -> 'd' ."
     }
     node: {
          title:     "2"
          label:     "state  2\n C -> 'c' . C"
     }
     backedge: {
          sourcename:     "2"
          targetname:     "1"
          label:     "'d'"
          color:     blue
          textcolor:     black
     }
     edge: {
          sourcename:     "2"
          targetname:     "2"
          label:     "'c'"
          color:     blue
          textcolor:     black
```

```
    }
    edge: {
        sourcename:     "2"
        targetname:     "5"
        label:     "C"
        color:     green
        textcolor:     black
    }
    node: {
        title:     "3"
        label:     "state  3\n $accept -> S . $end"
    }
    edge: {
        sourcename:     "3"
        targetname:     "6"
        label:     "$end"
        color:     blue
        textcolor:     black
    }
    node: {
        title:     "4"
        label:     "state  4\n S -> C . C '\\n'"
    }
    backedge: {
        sourcename:     "4"
        targetname:     "1"
        label:     "'d'"
        color:     blue
        textcolor:     black
    }
    backedge: {
        sourcename:     "4"
        targetname:     "2"
        label:     "'c'"
        color:     blue
        textcolor:     black
    }
    edge: {
        sourcename:     "4"
        targetname:     "7"
        label:     "C"
        color:     green
        textcolor:     black
    }
    node: {
        title:     "5"
        label:     "state  5\n C -> 'c' C ."
    }
    node: {
        title:     "6"
        label:     "state  6\n $accept -> S $end ."
    }
    node: {
        title:     "7"
        label:     "state  7\n S -> C C . '\\n'"
    }
    edge: {
        sourcename:     "7"
        targetname:     "8"
        label:     "'\\n'"
        color:     blue
        textcolor:     black
    }
    node: {
        title:     "8"
        label:     "state  8\n S -> C C '\\n' ."
    }
}
```

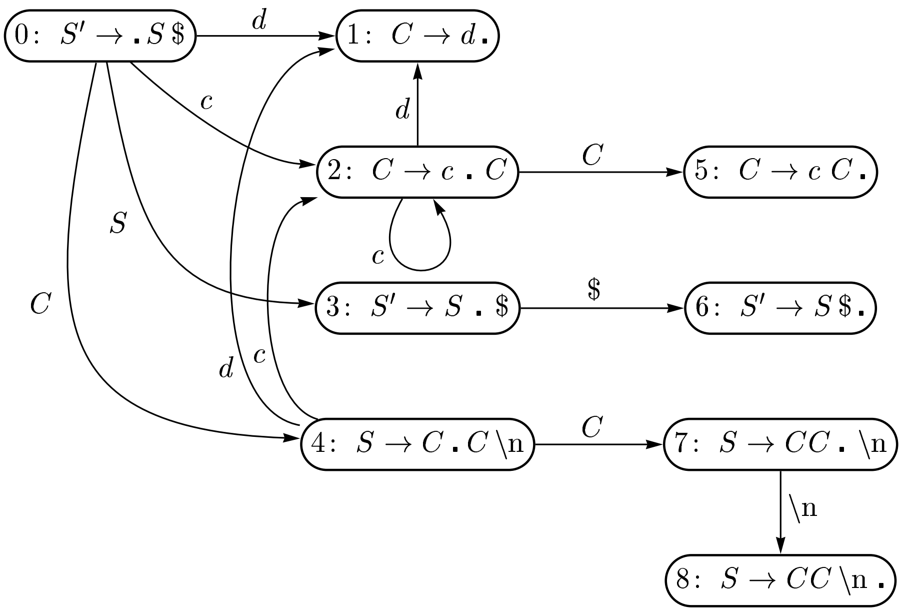

Figure 1: LALR(1) parsing automaton for the grammar with productions: $S' \to S\,\$$, $S \to C\ C\ \backslash n$, $C \to d \mid c\,C$.

```
/* Infix notation calculator.  */

%{
  #define YYSTYPE double /* YYSTYPE is the type of yyval */
  #include <math.h>
  #include <stdio.h>
  int yylex (void);
  void yyerror (char const *);
%}

/* Bison declarations.  */
/* precedence of operators is determined by the ordering
   of lines. For instance, ^ ties more than NEG */
%token NUM
%left '-' '+' /* left associative: 2-1-1=0*/
%left '*' '/' /* left associative */
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation */

%% /* The grammar follows.  */
input:    /* empty */
        | input line
;

line:     '\n'
        | exp '\n' { printf ("result: %.10g\n", $1);}
;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1,$3);}
        | '(' exp ')'        { $$ = $2;         }
     // | error              { yyerror;         }    // 1
     /* uncomment 1 if parsing should continue upon error */
;
%%
```

```
/* ----- Lexical Analyzer ------ */

/* The lexical analyzer returns a double floating point
   number on the stack and the token NUM, or the numeric code
   of the character read if not a number.  It skips all blanks
   and tabs, and returns 0 for end-of-input.  */

#include <ctype.h>

int yylex (void)
{
  int c;

  /* Skip white space.  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* Process numbers.  */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* Return end-of-input.  */
  if (c == EOF)
    return 0;
  /* Return a single char.  */
  return c;
}
/* ----- ERROR Routine ------ */

#include <stdio.h>

/* Called by yyparse on error.  */

void yyerror (char const *s)
{
  printf ("%s\n", s);
}

/* ----- MAIN program ------ */

int main (void)
{
  return yyparse ();
}
/* ------------------------------------------------------------------
$ bison -g calc.y

   produces calc.tab.c

$ cc -lm -o calc calc.tab.c

   produces  calc  (executable)
   The option -lm links to the mathematical library for "pow".

$ calc
3+2*7
        17
2^4*3
        48
$
------------------------------
The option -g produces a VCG graph description of the parse table.
------------------------------------------------------------------ */
```