5 Darkroom Software

If you have a home computer, some way to display a graphics image, and maybe even a way to capture images from a digitizer or a video recorder, you can easily build your own digital darkroom tools. To help you get started I will discuss a small interactive image editor named *popi* (pronounced “po-pee”). It is a portable version of the editor *pico* (“pee-ko”) that was used to generate the pictures in this book. (*Popi* is short for *portable pico*.)

The most noticeable difference between *pico* and *popi* is speed. *Pico* has a built-in optimizing compiler that translates transformation expressions into machine code for a DEC VAX computer. This compiler dramatically improves the performance of the editor, but of course, it works for only one specific target machine. Instead, *popi* translates the transformation expressions into programs that are interpreted by a little portable stack machine. This reduces the efficiency, but makes it possible to use the editor on any machine that can compile *C* programs.

Popi

The discussion that follows explains how *popi* works, how you can use it to create and edit images, and how user commands are parsed and executed. The editor is for black-and-white images, stored in disk files in raw format: one byte per pixel in scanline order, top scanline first. There are a few other restrictions to *popi*’s command language that you may or may not want to remove once you have this software running on your system. This version of the editor, for instance, does not know any trigonometric functions and does not know about polar coordinates. It will be relatively easy to make the extensions. Some hints on how to do that are given at the end of this chapter.

The resolution of the images you can process depends only on the amount of memory available on your system. The more memory, the better. The version discussed here works with 248×248 images, for which you will at least need 200 kilobytes of main memory in your system.

Command Language

Popi accepts five types of commands. The most important one is the image transformation command

```
    new = expression
```

that was used throughout this book. The other commands are

```
    r file
```

to read an image from a disk file into a read-only buffer,

```
    w file
```

to write the result of the last transformation into a file,

```
    f
```

to show which files are currently open, and

```
    q
```

to quit the editor.

Let us ignore the precise structure of the transformation expressions for a while. We return to that in the section titled *Grammar Rules*. We look first at the global structure of the editor. Commands can be typed on a single line separated by semicolons, or on separate lines. An edit session with *popi*, for instance, may go as follows.

```
    $ popi
    -> r rob          read image 'rob' into a buffer
    -> r pjw          read image 'pjw' into a buffer
    -> f              check which files are open
    $1 = rob
    $2 = pjw
    -> new=rob-pjw    subtract pjw from rob
    -> w prob         write result in a file 'prob'
    -> q              quit
    $                 operating system prompt
```

The arrow `->` is the editor's prompt: it tells the user that the program is ready for a new command.

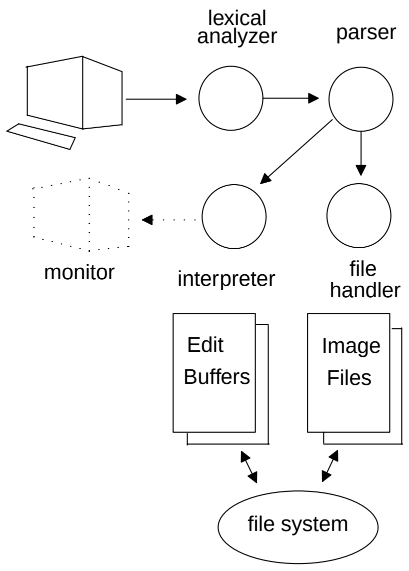
Program Structure

The image editor is written in the programming language *C*. It has four parts:

- a lexical analyzer,
- a recursive-descent parser,
- a file handler, and
- an interpreter, built as a stack-machine.

The figure illustrates how these parts fit together.

Consider what happens when the user (at the upper left-hand side of the figure) enters the command *new* $= x{*}{*}y$. The command is first read by the lexical analyzer. The job of the lexical analyzer is to recognize some predefined character sequences in the input, such as the image name *new* and the operator $**$. Each predefined sequence is passed as a single *token* to the next program module: the parser. The eight-character sequence *new* $= x{*}{*}y$ is passed as a sequence of five tokens (*new*, =, *x*, *POW*, and *y*) from the lexical analyzer to the parser.



The Structure of Popi

The parser looks at the sequence of tokens and decides what type of command it is. In this case it will decide that this is an image transformation command since the first token is *new* and not, for example, *r* or *w*. The parser then starts building a little program for the interpreter to perform the transformation. For our example the program should instruct the interpreter to calculate a new value x^y for each pixel (dot) of the image. The execution loop over all pixels is a fixed part of the interpreter. Note that the interpreter evaluates the complete transformation expression once for every pixel in the image being edited.

If the user types the command *r picture*, the parser will invoke the file handler to read the image stored in file *picture*. The image is read into a read-only buffer and can be used as a source for image edit operations. The only images that ever change during an edit session are the ones in the edit buffers. There are two edit buffers: one is called *new* and the other is called *old*. Buffer *old* holds the result of the last edit operation performed. Initially it is an all-zero, or black, image. The other edit buffer, *new*, is the target of the current edit operation. After the completion of each edit operation the two buffers are swapped; the result of the last edit operation becomes the start for a next one. (Note that this makes it trivial to add a one-level *undo* operation.)

You will almost certainly want to extend the editor with a display routine, dotted in the figure, to see the transformations on a monitor either while they are being performed by the interpreter, or after the interpreter completes. The specific display routine you will need, however, depends on the hardware you use. A sample routine for bi-level displays is included at the end of this chapter.

Throughout the program, library routines are used that can in one form or another be found on most systems. All library routines used are part of the proposed *ANSI* standard *C* language definition, so it should be fairly easy to find or to provide for equivalent routines on most systems. The software should run without change on well-known systems such as UNIX and MS-DOS.

Let's take a closer look now at the different parts of the editor. At the end of this chapter the complete program is listed.

The Lexical Analyzer

The task of the lexical analyzer is to recognize keywords such as *new* and *old*, and identify operators that consist of more than one character, such as >=, ! =, and &&. It must also find out when the name of an image file is used, and it must recognize numbers and convert the corresponding character strings into integers. White space (space and tab characters) is ignored. Everything that is not recognized by the lexical analyzer is passed on to the parser untouched. Single characters which are not part of a predefined sequence are treated as tokens with a code that equals their ASCII value. All other tokens are written in capitals, e.g., *POW*, and are defined as integer constants with a value greater than 255, the largest possible ASCII value.

The function `getchar()` is a library routine that returns the next available character in the command typed by the user.

The tokens passed from the lexical analyzer to the parser can have two attributes: *lexval* and *text*. They are declared as follows.

```
    int lexval;
    char text[256];
```

Tokens of the type *VALUE*, for example, have a value attribute stored in the integer variable *lexval*. Tokens of type *NAME* have a string attribute that is stored in array *text*. Tokens of type *FNAME* have both a value and a string attribute. The *FNAME* token refers to an open image file. Its string attribute gives the file name, and its value attribute is the number of the buffer into which the image was read.

This is what the lexical analyzer looks like:

```
lex()
{       int c;

        do      /* ignore white space */
                c = getchar();
        while (c == ' ' || c == '\t');

        if (isdigit(c))
                c = getnumber(c);
        else if (isalpha(c) || c == '_')
                c = getstring(c);

        switch (c) {
        case EOF:  c = 'q'; break;
        case '*':  c = follow('*', POW, c); break;
        case '>':  c = follow('=', GE,  c); break;
        case '<':  c = follow('=', LE,  c); break;
        case '!':  c = follow('=', NE,  c); break;
        case '=':  c = follow('=', EQ,  c); break;
        case '|':  c = follow('|', OR,  c); break;
        case '&':  c = follow('&', AND, c); break;
        case 'Z':  c = VALUE; lexval = 255; break;
        case 'Y':  c = VALUE; lexval = DEF_Y-1; break;
        case 'X':  c = VALUE; lexval = DEF_X-1; break;
        default :  break;
        }
        return c;
}
```

The function *follow*(*tok*, *ifyes*, *ifno*) looks at the next character typed. If it matches *tok*, the value *ifyes* is returned; if it does not match, the character is saved with *pushback*() (more about that below) and *ifno* is returned.

```
follow(tok, ifyes, ifno)
{       int c;

        if ((c = getchar()) == tok)
                return ifyes;
        pushback(c);

        return ifno;
}
```

The library routines *isdigit*(*c*) and *isalpha*(*c*) return nonzero (a boolean value *true* in *C*) when the argument is a digit or a letter, respectively. *getnumber*(*c*) and *getstring*(*c*) are shown below. They scan the user command for a number or a text string starting with character *c*.

```
getnumber(first)
{       int c;

        lexval = first - ’0’;
        while (isdigit(c = getchar()))
                lexval = 10*lexval + c - ’0’;
        pushback(c);
        return VALUE;
}
```

File names are looked up in a structure *src*[] where the I/O handler stores images. *nsrc* slots are in use, but the first two slots (0 and 1) are for the edit buffers and are skipped in the search. We will talk more about this data structure in a little while.

```
getstring(first)
{       int c = first;
        char *str = text;

        do {
                *str++ = c;
                c = getchar();
        } while (isalpha(c) || c == ’_’ || isdigit(c));
        *str = ’\0’;
        pushback(c);

        if (strcmp(text, "new") == 0) return NEW;
        if (strcmp(text, "old") == 0) return OLD;

        for (c = 2; c < nsrc; c++)
                if (strcmp(src[c].str, text) == 0)
                {       lexval = c-1;
                        return FNAME;
                }
        if (strlen(text) > 1)
                return NAME;
        return first;
}
```

A few more library routines are used here. *strlen*(*str*) returns the number of characters in the string, and *strcmp*(*str*1, *str*2) returns zero if the two given strings are equal. Normally, the lines with the function calls to *strcmp*() would be implemented as a search in a symbol table, but since we have only a few predefined symbols in this program, we can easily do without.

The last character read by routines *follow*(), *getnumber*(), and *getstring*() terminates a symbol and may start the next one. It must be saved for later, so it is pushed back onto the input. The function *pushback*(*c*) can be implemented with a one-slot buffer. In a UNIX environment it can be implemented as *ungetc*(*c*, *stdin*).

The Parser

Apart from correctly decoding the transformation expressions, the parser must be able to recognize file handling commands and respond to user inquiries. The parsing routines never call *getchar*() but use *lex*() for all input. The routine *parse*() itself is called from the *main*() procedure repeatedly, until it returns *false* (a value of zero):

```
main()
{
        ...
        do noerr=1; while( parse() );
}
```

The variable *noerr* is used as an error flag that is reset each time the parse routine is called. But more about error handling later. The parser will return zero only on an explicit *quit* command *q* from the user.

```
parse()
{       extern int lat;         /* look ahead token */

        printf("-> ");
        while (noerr)
        {       switch (lat = lex()) {
                case  'q': return 0;
                case '\n': return 1;
                case  ';': break;
                case  'f': showfiles();
                           break;
                case  'r': getname();
                           if (!noerr) continue;
                           getpix(&src[nsrc], text);
                           break;
                case  'w': getname();
                           if (!noerr) continue;
                           putpix(&src[CUROLD], text);
                           break;
                default  : transform();
                           if (noerr) run();
                           break;
        }       }
}
```

The parser indicates its willingness to accept a new command from the user by printing a prompt `->` with the I/O routine *printf*(*string*). The routine *printf*() is used in two flavors here. As shown, it just prints text on the user's screen, possibly with some extra arguments for printing numbers and character strings. We will also use it as *fprintf*(*stderr*, *string*) to print error messages. (If your system has no separate channel for error messages, they can equally well be printed with *printf*(*string*).)

Commands are typed on a single line separated by semicolons, or on separate lines. If the command can be parsed without syntax errors, the transformation program built by the parser is executed by the interpreter. This *program* is in reality a sequence of tokens (numbers) stored in an array called *parsed*[]. Below, this is referred to as the *parse string*. For completeness, here is also the routine *getname*() that is used in *parse*() to fetch a filename argument (a single alphanumeric character, or one of the tokens *FNAME* or *NAME* with a string attribute).

```
getname()
{       int t = lex();

        if (t != NAME && t != FNAME && !isalpha(t))
                error("expected name, bad token: %d\n", t);
}
```

error() is the routine to be called when a syntax error is detected. It will eat up the rest of the input, up to a newline, and set an error flag that will avoid the interpreter from being run on erroneous input.

```
error(s, d)
        char *s;
{
        extern int lat;

        fprintf(stderr, s, d);
        while (lat != '\n')
                lat = lex();
        noerr = 0;      /* noerr is now false */
}
```

Having seen the lexical analyzer and the main routine of the parser, you already know almost everything there is to know about the working of *popi*. We only have to fill in a few details, such as the parsing of expressions, the execution of parse strings and the reading and writing of image files.

Grammar Rules

When the parser starts processing a transformation expression it expects to see a sequence such as

```
new[x,y] = expression.
```

This sequence reaches the parser as a token *NEW* followed by a character [, an expression for the *x* index, a comma, another expression for the *y*, a character =, and a final expression for the values to be assigned. As an added difficulty we will allow for certain standard parts of this sequence to be omitted. The editor should be able to fill in the missing parts with defaults. For instance, if the index to array *new* is missing, as in

```
new = expression
```

the parser may assume a default index [*x*, *y*]. If the token *new* is also missing, and the statement just reads

```
expression
```

the parser can assume that you meant an assignment to image array *new* with the default index [*x*, *y*].

To make more precise what type of expressions are acceptable to the parser, we define a grammar. A transformation, for instance, takes one of three possible forms, which we can describe as follows:

```
    trans   →       NEW ‘[’ index ‘]’ ‘=’ expr
            |       NEW ‘=’ expr
            |       expr
```

This grammar rule is called a *production*. On the left-hand side of the arrow we write the name of the phrase or language fragment we want to define. Here we wrote *trans*, as an abbreviation of “transformation expression.” The right-hand side shows a number of alternative ways in which the phrase can be constructed. The alternatives are separated by vertical bars. Quoted characters are called *literals*, and names in capitals are called *terminals*. Everything in a production except the literals and terminals must be expanded in still other productions, so that eventually everything can be defined recursively in terms of literals and terminals only. Note that the lexical analyzer we discussed before will recognize all the literals and terminals for us, and pass them as tokens to the parser.

The only undefined term in the production above is *expr*. So we will have to define it.

```
    expr    →       term
            |       term ‘?’ expr ‘:’ expr
```

This rule says that an expression is either a conditional expression or something called a *term*. Note that conditional expressions can have other conditionals inside, but not in the term before the question mark.

```
    term    →       factor binop term
```

This time we have two nonterminals in the production: *binop* and *factor*. The expansion for *binop*, or binary operator, is quickly defined.

```
    binop   →       ‘*’ | ‘/’ | ‘%’
            |       ‘+’ | ‘-’
            |       ‘>’ | ‘<’ | GE | LE | EQ | NE
            |       ‘^’ | AND | OR
```

A *factor* is a little more work.

```
    factor  →       ‘(’ expr ‘)’
            |       ‘-’ factor
            |       ‘!’ factor
            |       OLD
            |       fileref
            |       value
            |       ‘x’
            |       ‘y’
            |       factor POW factor
```

A factor, then, can be any expression enclosed in parentheses; it may be preceded by a single minus sign (the unary minus) or a single exclamation mark (a boolean negation). It can be the token *OLD*, the characters *x* and *y*, or a *factor* raised to the power of some other *factor*. It can also be a constant value or a file reference *fileref*. This nonterminal, in turn, can be defined as follows:

```
fileref →       FNAME
        |       FNAME ‘[’ index ‘]’
        |       ‘$’ value
        |       ‘$’ value ‘[’ index ’]’
```

$FNAME$ is the token returned by the lexical analyzer when it recognizes the name of an open image file in the input. We allow for file references to be either symbolic (e.g., *pjw*[*x*,*y*]) or numeric (as in $1[*x*,*y*]). The correspondence between names and numbers is given by the *f* command, discussed earlier. This leaves only the following nonterminals to be expanded.

```
value   →       digit | digit value
index   →       expr ‘,’ expr
```

and trivially:

```
digit   →       ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’
        |       ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
```

Now let’s look at the code for the parser that makes it all happen. The parser’s job is to read the transformation expressions, flag syntax errors, and build a program for the interpreter that encodes the expressions.

The treatment of the defaults requires some attention. Remember that each image buffer takes either an explicit, user-defined index or, if the user omits it, an implicit index [*x*,*y*]. If the index to the destination buffer *new* is omitted, the parser inserts a special symbol @ into the program to tell the interpreter to use the default index and destination, i.e., *new*[*x*,*y*]. The processing of explicit or implicit indexes to image files stored in other image buffers is deferred to a procedure named *fileref*().

Here, first, is the code for routine *transform*(), that is called from *parse*() at the start of a transformation expression.

```
transform()
{       extern int prs;

        prs = 0; /* initial length of parse string */
        if (lat != NEW)
        {       expr();
                emit(’@’);
                pushback(lat);
                return;
        }
        lat = lex();
        if (lat == ’[’)
        {       fileref(CURNEW, LVAL);
                expect(’=’); expr(); emit(’=’);
        } else
        {       expect(’=’); expr(); emit(’@’);
        }
        if (lat != '\n' && lat != ';')
                error("syntax error, separator\n");
        pushback(lat);
}
```

The symbol @ is really an abstract instruction of the stack machine that runs the transformations. An assignment to any other location in array *new* is encoded with an explicit “instruction” =. The program for the interpreter, or the *parse string* as we have called it before, is built in an array called *parsed*. We use a procedure *emit*(*symbol*) to add new symbols to the parse string.

```
emit(what)
{
        if (prs >= MANY)
                error("expression too long\n");
        parsed[prs++] = what;
}
```

The parse string is in standard postfix notation, with a few special operators such as @ and =. In postfix notation the expression $new = x**y$ becomes ‘$x, y, POW, @$.’ The operators simply follow the operands to which they apply, instead of sitting between them. This format greatly simplifies the design of the interpreter. Procedure *expect*(*token*) checks that the look-ahead token *lat* matches the expected value and then reads in the next token from the lexical analyzer.

```
expect(t)
{
        if (lat == t)
                lat = lex();
        else
                error("error: expected token %d\n",t);
}
```

fileref() is a procedure that decodes a reference to an image buffer. It has to check that the image is really available, and it must properly decode the index. A file reference can occur in two different contexts: on the left-hand side or the right-hand side of an assignment. In the first case the interpreter will have to calculate the address of the location in the destination buffer where a new pixel value is to be stored; in the second case it simply needs to read a pixel value from that location and can forget about its address. The parser will distinguish the two different uses by using either the symbol *LVAL* or *RVAL* in the second argument to *fileref*(). As it turns out, we will only allow array *new* to be used as an *LVAL*. Here is the code. The appropriate token value to be issued is passed as in argument *tok* (see, for instance, the usage of *fileref*() in the *transform*() routine above).

```
fileref(val, tok)
{
        if (val < 0 || val >= nsrc)
                error("bad file number: %d\n", val);

        emit(VALUE);
        emit(val);
        if (lat == '[')
        {       lat = lex();
                expr(); expect(',');
                expr(); expect(']');    /* [x,y] */
        } else
        {       emit('x');
                emit('y');
        }
        emit(tok);
}
```

A value is encoded in the parse string by the symbol *VALUE* followed by the actual number seen.

More about Parsing Expressions

The most important part of the parser starts with *expr*(). It is time to worry here about operator precedence rules. An expression like $512 * y + x$ is to be interpreted as $(512 * y) + x$ instead of $512 * (y + x)$. In postfix form this is the difference between parsing 512,*y*,*,*x*,+ and 512,*y*,*x*,+,*. To ensure that the parser gives multiplications, divisions, and modulo operations a higher priority than, for instance, additions or subtractions, we use a lookup table that encodes four precedence levels.

```
int op[4][7] = {
        { '*', '/', '%', 0, 0, 0, 0, },
        { '+', '-',   0, 0, 0, 0, 0, },
        { '>', '<',  GE, LE, EQ, NE, 0, },
        { '^', AND,  OR, 0, 0, 0, 0, },
};
```

Procedure *expr*() uses a generic procedure *level*(*nr*) to parse sub-expressions for precedence level *nr*. It tries to parse the highest-precedence operators first. At the highest level it tries to find a *factor*() such as a number or a variable. At the other levels it checks for the appropriate operators in the table shown above. The lowest level is given to the operators of a conditional expression (a question mark and a colon).

```
expr()
{       extern int prs;
        extern int parsed[MANY];
        int remem1, remem2;

        level(3);
        if (lat == '?')
        {       lat = lex();
                emit('?');
                remem1 = prs; emit(0);
                expr();
                expect(':'); emit(':');
                remem2 = prs; emit(0);
                parsed[remem1] = prs-1;
                expr();
                parsed[remem2] = prs-1;
        }
}
```

Conditional image transformation commands are built from three sub-expressions:

`condition?iftrue:iffalse`.

If, at runtime, the condition is found to be true, the interpreter will execute the instructions in array *parsed* for the *iftrue* part. If, however, the condition is found to be false, the interpreter should be able to skip to the *iffalse* part. To allow the interpreter to do this at runtime, the parser reserves a slot in the parse string for the destination of the jump. The slot is filled with that destination after the colon has been parsed. Similarly, after executing an *iftrue* part, the interpreter must skip to the end of the conditional. Again the proper destination can be patched into the string, but only after the end of the *iffalse* expression has been seen (i.e., a semicolon or a newline).

```
level(nr)
{       int i;
        extern int noerr;

        if (nr < 0)
        {       factor();
                return;
        }
        level(nr-1);
        for (i = 0; op[nr][i] != 0 && noerr; i++)
                if (lat == op[nr][i])
                {       lat = lex();
                        level(nr);
                        emit(op[nr][i]);
                        break;
                }
}
```

A factor, finally, is defined as follows:

```
factor()
{       int n;

        switch (lat) {
        case   '(':     lat = lex();
                        expr();
                        expect(')');
                        break;
        case   '-':     lat = lex();
                        factor();
                        emit(UMIN);
                        break;
        case   '!':     lat = lex();
                        factor();
                        emit('!');
                        break;
        case   OLD:     lat = lex();
                        fileref(CUROLD, RVAL);
                        break;
        case FNAME:     n = lexval;
                        lat = lex();
                        fileref(n+1, RVAL);
                        break;
        case   '$':     lat = lex();
                        expect(VALUE);
                        fileref(lexval+1, RVAL);
                        break;
        case VALUE:     emit(VALUE);
                        emit(lexval);
                        lat = lex();
                        break;
        case 'y':
        case 'x':       emit(lat);
                        lat = lex();
                        break;
        default :       error("expr: syntax error\n");
        }
        if (lat == POW)
        {       lat = lex();
                factor();
                emit(POW);
        }
}
```

In the order listed, a factor can be an expression enclosed in parentheses, a factor preceded by a minus sign (unary minus), or a logical negation. It can also be a symbolic or a numeric file reference, a value, a Cartesian coordinate, or a factor raised to some other factor with a power operator.

So far, we have discussed the lexical analyzer and the parser. The interpreter and the file handler remain to be discussed. Before we discuss the file handler, first a few words about the specific data structures that are used for storing the images.

Data Structure

Images are stored in a structure of the following type:

```
struct SRC {
        char *str;              /* file name    */
        unsigned char **pix;    /* pixel values */
} src[];
```

The format is one byte per pixel, *DEF_X* pixels per scanline, and *DEF_Y* scanlines per image. We will use two of these structures as edit buffers: *src*[0] and *src*[1] with *DEF_Y*×*DEF_X* pixels. The result of the last edit operation is in one of these two arrays and is referred to as *old*. Internally, it is addressed as *src*[*CUROLD*].*pix*. Similarly, the destination of an edit operation is in the other edit buffer named *new*, or *src*[*CURNEW*].*pix*.

The default image resolution is defined by two constants:

```
#define DEF_X   248
#define DEF_Y   248
```

What follows is an aside on the choice of the resolution, which you can skip on a first reading:

> The precise value you can afford to enter for *DEF_X* and *DEF_Y* depends on the amount of memory in your computer and the type of memory allocation performed. On most systems, an image size of 248×248 will allow for a generous number of image files to be open for editing simultaneously. Smaller computers, that is, computers with a wordsize of 16 instead of 32 or 64 bits, can make it hard to allocate more than 2^{16} = 64k bytes at a time. We therefore chose to allocate memory for the images in increments of *DEF_X*, once for each scanline, instead of once for each picture.
>
> A few more hairy details. An allocator sometimes uses a few bytes from each block allocated for its own housekeeping. On a small system, this makes it attractive to pick a value for *DEF_X* that is the same number of bytes smaller than a power of 2 so that we can evenly fill up available memory, without leaving gaps. (The amount of real memory available comes in powers of 2.) In this case we chose 256 − 8 = 248, which turns out to be an good size for an AT&T PC6300+ system running UNIX. On a larger system you need not worry about memory allocation details and just select any frame size that is convenient to work with.

File Handler

We need routines to read and write image files, and to show which files are currently open. The last one is easy:

```
showfiles()
{       int n;

        if (nsrc == 2)
                printf("no files open\n");
        else
                for (n = 2; n < nsrc; n++)
                        printf("$%d = %s\n", n-1, src[n].str);
}
```

It uses *printf*() again, this time with two extra arguments: a number to be printed at the place indicated with *%d* and a name to be printed at the *%s*.

Reading new files, including the required memory allocation, can be done as follows, using the standard I/O routines *fopen*(), *fread*(), and *fclose*(). A call to *fopen*(*str*,"*r*") opens the file named *str* for reading. Similarly *fopen*(*str*,"*w*") opens the file for writing, and creates it if it does not exist. *fread*(*ptr*,*n*,*m*,*fd*) reads *m* chunks of *n* bytes from the file referred to by file descriptor *fd* and places it at the location given by *ptr*.

```
getpix(into, str)
        struct SRC *into;       /* work buffer */
        char *str;              /* file name   */
{
        FILE *fd;
        int i;

        if ((fd = fopen(str, "r")) == NULL)
        {       fprintf(stderr, "no file %s\n", str);
                return;
        }

        if (into->pix == (unsigned char **) 0)
        {       into->pix = (unsigned char **)
                        Emalloc(DEF_Y * sizeof(unsigned char *));
                for (i = 0; i < DEF_Y; i++)
                        into->pix[i] = (unsigned char *)
                                Emalloc(DEF_X);
        }
        into->str = (char *) Emalloc(strlen(str)+1);
        if (!noerr) return;     /* set by Emalloc */

        for (i = 0; i < DEF_Y; i++)
                fread(into->pix[i], 1, DEF_X, fd);
        strcpy(into->str, str);

        fclose(fd);
        nsrc++;
}
```

A separate procedure is used to access the memory allocator, to make it easier to catch errors:

```
char *
Emalloc(N)
{       char *try, *malloc();

        if ((try = malloc(N)) == NULL)
                error("out of memory\n");
        return try;
}
```

Writing files is also simple. *fwrite*(*ptr*, *n*, *m*, *fd*) writes *m* chunks of data, of *n* bytes each, from a location pointed to by *ptr* into the file with descriptor *fd*.

```
putpix(into, str)
        struct SRC *into;       /* work buffer */
        char *str;              /* file name   */
{
        FILE *fd;
        int i;

        if ((fd = fopen(str, "w")) == NULL)
        {       fprintf(stderr, "cannot create %s\n", str);
                return;
        }
        for (i = 0; i < DEF_Y; i++)
                fwrite(into->pix[i], 1, DEF_X, fd);
        fclose(fd);
}
```

The Interpreter

The interpreter is a sensitive piece of code since it is asked to execute the parse string once for every pixel in the image being edited. Even for a small image size of 248×248 pixels the interpreter must run through the parse string 61,504 times. It is important that it is as fast as it can be.

The interpreter is built as a *stack* machine. It maintains a pointer *rr* to a stack of values. Values and variables are pushed onto the stack. Operators and functions pop the stack and replace the values at the top with their result. At the end of each run of the interpreter that stack should be empty. The runtime stack contains *long* values instead of *integers* to allow for the computation of both pixel values and pixel addresses.

For convenience we define a macro *dop*(*OP*) for the frequently occurring operation that pops two values from the stack, applies an operation *OP* to them, and then pushes the result back.

```
#define dop(OP) a = *--rr; tr = rr-1; *tr = (*tr OP (long)a)
```

Variable *rr* points to the first free slot on the stack. The topmost symbol on the stack is at position ($rr - 1$). The *dop* macro can be used for all binary arithmetic and boolean operations. The interpreter will keep a pointer to the default destination of pixel assignments up to date in a pointer *p*.

```
run()
{       long R[MANY];              /* the stack    */
        register long *rr, *tr;    /* top of stack */
        register unsigned char *u; /* explicit destination */
        register unsigned char *p; /* default  destination */
        register int k;            /* indexes parse string */
        int a, b, c;               /* scratch     */
        int x, y;                  /* coordinates */

        p = src[CURNEW].pix[0];
        for (y = 0; y < DEF_Y; y++, p = src[CURNEW].pix[y])
        for (x = 0; x < DEF_X; x++, p++)
        for (k = 0, rr = R; k < prs; k++)
        {       if (parsed[k] == VALUE)
                {       *rr++ = (long)parsed[++k];
                        continue;
                }
                if (parsed[k] == '@')
                {       *p = (unsigned char) (*--rr);
                        continue;
                }
                switch (parsed[k]) {
                case  '+': dop(+);  break;
                case  '-': dop(-);  break;
                case  '*': dop(*);  break;
                case  '/': dop(/);  break;
                case  '%': dop(%);  break;
                case  '>': dop(>);  break;
                case  '<': dop(<);  break;
                case   GE: dop(>=); break;
                case   LE: dop(<=); break;
                case   EQ: dop(==); break;
                case   NE: dop(!=); break;
                case  AND: dop(&&); break;
                case   OR: dop(||); break;
                case  '^': dop(|);  break;
                case  'x': *rr++ = (long)x; break;
                case  'y': *rr++ = (long)y; break;
                case UMIN: tr = rr-1; *tr = -(*tr); break;
                case  '!': tr = rr-1; *tr = !(*tr); break;
                case  '=': a = *--rr;
                           u = (unsigned char *) *--rr;
                           *u = (unsigned char) a;
                           break;
                case RVAL: a = *--rr;
                           b = *--rr;
                           tr = rr-1;
                           c = *tr;
                           *tr = (long) src[c].pix[a][b];
                           break;
                case LVAL: a = *--rr;
                           b = *--rr;
                           tr = rr-1;
                           c = *tr;
                           *tr = (long) &(src[c].pix[a][b]);
                           break;
                case  POW: a = *--rr;
                           *(rr-1) = Pow(*(rr-1),(long)a);
                           break;
                case  '?': a = *--rr; k++;
                           if (!a) k = parsed[k];
                           break;
                case  ':': k = parsed[k+1]; break;

                default  : error("run: unknown operator\n");
                }
        }
        CUROLD = CURNEW; CURNEW = 1-CUROLD;
}
```

It may look curious that two of the "instructions" are processed before the large case switch is entered. They catch the two most frequently executed operations and it makes the interpreter run faster to process them first. At the end of the run the two edit buffers, referred to by *old* and *new*, are swapped. Note that the procedure *Pow*() accepts and returns *long* values. To make use of the *C* library routine, you can define

```
long
Pow(a, b)
        long a, b;
{       double c = (double)a;
        double d = (double)b;
        return (long) pow(c, d);
}
```

The Complete Program

Here, finally, is the complete listing of the program, with all loose ends neatly tied up. On a UNIX system it is compiled with the command

```
cc -o popi main.c lex.c io.c expr.c run.c
```

If *display* is the name of a display routine that can show an unformatted image file on the specific monitor you use, you can test the working of the editor with the commands.

```
$ popi              invoke the editor
-> x^y              create a test pattern
-> w test           write it in a file
-> q                quit the editor
$ display test      display the result
```

Try it. You'll like it.

```
/***  popi.h (header file) **************************/

#define MANY    128
#define DEF_X   248     /* image width  */
#define DEF_Y   248     /* image height */

#define RVAL    257     /* larger than any char token */
#define LVAL    258
#define FNAME   259
#define VALUE   260
#define NAME    261
#define NEW     262
#define OLD     263
#define AND     264
#define OR      265
#define EQ      266
#define NE      267
#define GE      268
#define LE      269
#define UMIN    270
#define POW     271

struct SRC {
        unsigned char **pix;    /* pix[y][x] */
        char *str;
};

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

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

int     parsed[MANY];
struct  SRC     src[MANY];
short   CUROLD=0, CURNEW=1;
int     noerr, lexval, prs=0, nsrc=2;
char    text[256];

char *Emalloc();

main(argc, argv)
        char **argv;
{
        int i;

        src[CUROLD].pix = (unsigned char **)
                Emalloc(DEF_Y * sizeof(unsigned char *));
        src[CURNEW].pix = (unsigned char **)
                Emalloc(DEF_Y * sizeof(unsigned char *));
        for (i = 0; i < DEF_Y; i++)
        {       src[CUROLD].pix[i] = (unsigned char *)
                        Emalloc(DEF_X);
                src[CURNEW].pix[i] = (unsigned char *)
                        Emalloc(DEF_X);
        }

        for (i = 1; i < argc; i++)
                getpix(&src[nsrc], argv[i]);

        do noerr=1; while( parse() );
}

parse()
{       extern int lat;         /* look ahead token */

        printf("-> ");
        while (noerr)
        {       switch (lat = lex()) {
                case  ’q’: return 0;
                case ’\n’: return 1;
                case  ’;’: break;
                case  ’f’: showfiles();
                           break;
                case  ’r’: getname();
                           if (!noerr) continue;
                           getpix(&src[nsrc], text);
                           break;
                case  ’w’: getname();
                           if (!noerr) continue;
                           putpix(&src[CUROLD], text);
                           break;
                default  : transform();
                           if (noerr) run();
                           break;
        }       }
}

getname()
{       int t = lex();

        if (t != NAME && t != FNAME && !isalpha(t))
                error("expected name, bad token: %d\n", t);
}

emit(what)
{
        if (prs >= MANY)
                error("expression too long\n");
        parsed[prs++] = what;
}
```

```
error(s, d)
        char *s;
{
        extern int lat;

        fprintf(stderr, s, d);
        while (lat != '\n')
                lat = lex();
        noerr = 0;      /* noerr is now false */
}

char *
Emalloc(N)
{       char *try, *malloc();

        if ((try = malloc(N)) == NULL)
                error("out of memory\n");
        return try;
}

/***  lex.c  (lexical analyzer) *********************/

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

extern struct   SRC src[MANY];
extern short    CUROLD, CURNEW;
extern int      nsrc, lexval;
extern char     text[];

lex()
{       int c;

        do      /* ignore white space */
                c = getchar();
        while (c == ' ' || c == '\t');

        if (isdigit(c))
                c = getnumber(c);
        else if (isalpha(c) || c == '_')
                c = getstring(c);
        switch (c) {
        case EOF:  c = 'q'; break;
        case '*':  c = follow('*', POW, c); break;
        case '>':  c = follow('=', GE,  c); break;
        case '<':  c = follow('=', LE,  c); break;
        case '!':  c = follow('=', NE,  c); break;
        case '=':  c = follow('=', EQ,  c); break;
        case '|':  c = follow('|', OR,  c); break;
        case '&':  c = follow('&', AND, c); break;
        case 'Z':  c = VALUE; lexval = 255; break;
        case 'Y':  c = VALUE; lexval = DEF_Y-1; break;
        case 'X':  c = VALUE; lexval = DEF_X-1; break;
        default :  break;
        }
        return c;
}

getnumber(first)
{       int c;

        lexval = first - '0';
        while (isdigit(c = getchar()))
                lexval = 10*lexval + c - '0';
        pushback(c);
        return VALUE;
}

getstring(first)
{       int c = first;
        char *str = text;

        do {
                *str++ = c;
                c = getchar();
        } while (isalpha(c) || c == '_' || isdigit(c));
        *str = '\0';
        pushback(c);

        if (strcmp(text, "new") == 0) return NEW;
        if (strcmp(text, "old") == 0) return OLD;

        for (c = 2; c < nsrc; c++)
                if (strcmp(src[c].str, text) == 0)
                {       lexval = c-1;
                        return FNAME;
                }
        if (strlen(text) > 1)
                return NAME;
        return first;
}
```

```
follow(tok, ifyes, ifno)
{       int c;

        if ((c = getchar()) == tok)
                return ifyes;
        pushback(c);

        return ifno;
}

pushback(c)
{
        ungetc(c, stdin);
}

/***  io.c   (file handler)  ************************/

#include        <stdio.h>
#include        "popi.h"

extern struct SRC src[MANY];
extern int nsrc, noerr;
extern char *Emalloc();

getpix(into, str)
        struct SRC *into;       /* work buffer */
        char *str;              /* file name   */
{
        FILE *fd;
        int i;

        if ((fd = fopen(str, "r")) == NULL)
        {       fprintf(stderr, "no file %s\n", str);
                return;
        }

        if (into->pix == (unsigned char **) 0)
        {       into->pix = (unsigned char **)
                        Emalloc(DEF_Y * sizeof(unsigned char *));
                for (i = 0; i < DEF_Y; i++)
                        into->pix[i] = (unsigned char *)
                                Emalloc(DEF_X);
        }
        into->str = (char *) Emalloc(strlen(str)+1);
        if (!noerr) return;     /* set by Emalloc */

        for (i = 0; i < DEF_Y; i++)
                fread(into->pix[i], 1, DEF_X, fd);
        strcpy(into->str, str);

        fclose(fd);
        nsrc++;
}
```

```
putpix(into, str)
        struct SRC *into;       /* work buffer */
        char *str;              /* file name   */
{
        FILE *fd;
        int i;

        if ((fd = fopen(str, "w")) == NULL)
        {       fprintf(stderr, "cannot create %s\n", str);
                return;
        }
        for (i = 0; i < DEF_Y; i++)
                fwrite(into->pix[i], 1, DEF_X, fd);
        fclose(fd);
}

showfiles()
{       int n;

        if (nsrc == 2)
                printf("no files open\n");
        else
                for (n = 2; n < nsrc; n++)
                        printf("$%d = %s\n", n-1, src[n].str);
}

/***  expr.c (parser)  ******************************/

#include "popi.h"

extern int      lexval, nsrc;
extern struct   SRC     src[MANY];
extern short    CUROLD, CURNEW;
int             lat;    /* look ahead token */

int op[4][7] = {
        { '*', '/', '%', 0, 0, 0, 0, },
        { '+', '-',   0, 0, 0, 0, 0, },
        { '>', '<',  GE, LE, EQ, NE, 0, },
        { '^', AND,  OR, 0, 0, 0, 0, },
};
```

```
expr()
{       extern int prs;
        extern int parsed[MANY];
        int remem1, remem2;

        level(3);
        if (lat == '?')
        {       lat = lex();
                emit('?');
                remem1 = prs; emit(0);
                expr();
                expect(':'); emit(':');
                remem2 = prs; emit(0);
                parsed[remem1] = prs-1;
                expr();
                parsed[remem2] = prs-1;
        }
}

level(nr)
{       int i;
        extern int noerr;

        if (nr < 0)
        {       factor();
                return;
        }
        level(nr-1);
        for (i = 0; op[nr][i] != 0 && noerr; i++)
                if (lat == op[nr][i])
                {       lat = lex();
                        level(nr);
                        emit(op[nr][i]);
                        break;
                }
}

transform()
{       extern int prs;

        prs = 0; /* initial length of parse string */
        if (lat != NEW)
        {       expr();
                emit('@');
                pushback(lat);
                return;
        }
        lat = lex();
        if (lat == '[')
        {       fileref(CURNEW, LVAL);
                expect('='); expr(); emit('=');
        } else
        {       expect('='); expr(); emit('@');
        }
        if (lat != '\n' && lat != ';')
                error("syntax error, separator\n");
        pushback(lat);
}

factor()
{       int n;

        switch (lat) {
        case   '(':     lat = lex();
                        expr();
                        expect(')');
                        break;
        case   '-':     lat = lex();
                        factor();
                        emit(UMIN);
                        break;
        case   '!':     lat = lex();
                        factor();
                        emit('!');
                        break;
        case   OLD:     lat = lex();
                        fileref(CUROLD, RVAL);
                        break;
        case FNAME:     n = lexval;
                        lat = lex();
                        fileref(n+1, RVAL);
                        break;
        case   '$':     lat = lex();
                        expect(VALUE);
                        fileref(lexval+1, RVAL);
                        break;
        case VALUE:     emit(VALUE);
                        emit(lexval);
                        lat = lex();
                        break;
        case 'y':
        case 'x':       emit(lat);
                        lat = lex();
                        break;
        default :       error("expr: syntax error\n");
        }
        if (lat == POW)
        {       lat = lex();
                factor();
                emit(POW);
        }
}

fileref(val, tok)
{
        if (val < 0 || val >= nsrc)
                error("bad file number: %d\n", val);

        emit(VALUE);
        emit(val);
        if (lat == '[')
        {       lat = lex();
                expr(); expect(',');
                expr(); expect(']');    /* [x,y] */
        } else
        {       emit('x');
                emit('y');
        }
        emit(tok);
}

expect(t)
{
        if (lat == t)
                lat = lex();
        else
                error("error: expected token %d\n",t);
}

/***  run.c  (interpreter) **************************/

#include        "popi.h"

extern int      prs, parsed[MANY];
extern struct   SRC     src[MANY];
extern short    CUROLD, CURNEW;

#define dop(OP) a = *--rr; tr = rr-1; *tr = (*tr OP (long)a)

long
Pow(a, b)
        long a, b;
{
        double c = (double)a;
        double d = (double)b;
        double pow();

        return (long) pow(c, d);
}
```

```
run()
{       long R[MANY];               /* the stack    */
        register long *rr, *tr;     /* top of stack */
        register unsigned char *u;  /* explicit destination */
        register unsigned char *p;  /* default  destination */
        register int k;             /* indexes parse string */
        int a, b, c;                /* scratch     */
        int x, y;                   /* coordinates */

        p = src[CURNEW].pix[0];
        for (y = 0; y < DEF_Y; y++, p = src[CURNEW].pix[y])
        for (x = 0; x < DEF_X; x++, p++)
        for (k = 0, rr = R; k < prs; k++)
        {       if (parsed[k] == VALUE)
                {       *rr++ = (long)parsed[++k];
                        continue;
                }
                if (parsed[k] == ’@’)
                {       *p = (unsigned char) (*--rr);
                        continue;
                }
                switch (parsed[k]) {
                case  ’+’: dop(+);  break;
                case  ’-’: dop(-);  break;
                case  ’*’: dop(*);  break;
                case  ’/’: dop(/);  break;
                case  ’%’: dop(%);  break;
                case  ’>’: dop(>);  break;
                case  ’<’: dop(<);  break;
                case   GE: dop(>=); break;
                case   LE: dop(<=); break;
                case   EQ: dop(==); break;
                case   NE: dop(!=); break;
                case  AND: dop(&&); break;
                case   OR: dop(||); break;
                case  ’ˆ’: dop(|);  break;
                case  ’x’: *rr++ = (long)x; break;
                case  ’y’: *rr++ = (long)y; break;
                case UMIN: tr = rr-1; *tr = -(*tr); break;
                case  ’!’: tr = rr-1; *tr = !(*tr); break;
                case  ’=’: a = *--rr;
                           u = (unsigned char *) *--rr;
                           *u = (unsigned char) a;
                           break;
                case RVAL: a = *--rr;
                           b = *--rr;
                           tr = rr-1;
                           c = *tr;
                           *tr = (long) src[c].pix[a][b];
                           break;
                case LVAL: a = *--rr;
                           b = *--rr;
                           tr = rr-1;
                           c = *tr;
                           *tr = (long) &(src[c].pix[a][b]);
                           break;
                case  POW: a = *--rr;
                           *(rr-1) = Pow(*(rr-1),(long)a);
                           break;
                case  '?': a = *--rr; k++;
                           if (!a) k = parsed[k];
                           break;
                case  ':': k = parsed[k+1]; break;

                default  : error("run: unknown operator\n");
                }
        }
        CUROLD = CURNEW; CURNEW = 1-CUROLD;
}
```

Library Routines

Here is an overview of the library routines that were used in the program. It should not be hard to find equivalents for them if you run on a system other than UNIX. They are all considered standard routines in the *C* programming language.

fclose	fread	isalpha	pow	strcpy
fopen	fwrite	isdigit	printf	strlen
fprintf	getchar	malloc	strcmp	ungetc

Efficiency Considerations

The complete program listed above is only about 500 lines of *C* text. Yet it can be an amazingly powerful package once you get the hang of the notation for transformation expressions. I have run it under a UNIX operating system on an AT&T PC6300+, a DEC VAX/750, a DEC VAX/785, and a CRAY/XMP computer. Not to worry, the difference in performance for the software is not nearly as spectacular as the difference in price of this hardware. Here is a comparison of runtimes for a few transformations. All times given are in seconds.

Runtimes (frame size: 248×248 pixels)

Transformation	PC6300+	VAX/750	VAX/785	CRAY/XMP
new=128	7.2	4.1	1.6	0.1
new=pjw	24.2	10.5	3.7	0.4
new=Z-old	32.4	15.1	5.2	0.6
new=(pjw+rob)/2	59.2	25.9	9.7	1.0
new=(x<X/2)?pjw:rob	63.1	33.2	13.9	1.4
new=(pjw<128)?Z-pjw:pjw[X-x,y]	79.3	38.6	14.8	1.6

The times quoted are for the program as listed, with interpreted code. Since the interpreter runs the same code many thousands of times, even for a single image transformation, every improvement in that portion of the code pays off immediately. One way to achieve a substantial speedup is to replace the interpreter with an on-the-fly compiler that translates the transformation program into machine code and runs it. It will go too far to get into the details of that extension, but suffice it to say that it is worth the effort. The speed difference between interpreted code and compiled code can be as high as 1:100 for nontrivial transformations. The program *popi* you see above is a predecessor of the image editor called *pico*. *Pico* has the same basic structure as *popi*, but has the built-in compiler to make it faster. The compiler extension more than doubles the size of the program, but in a way it can achieve the same as the purchase of a Cray supercomputer. (The performance of running *pico* with compiled code on a Cray is of course utterly decadent.)

Adding a Display Routine

To display an image on your terminal screen requires code that is hardware dependent, so I have not included it in *popi*, but you can easily extend the software in this way. If you have a one-byte-per-pixel monitor, you can write the pixel values to the screen without processing.

```
display(pix)
        unsigned char **pix;
{
        register int x, y;

        for (y = 0; y < DEF_Y; y++)
        for (x = 0; x < DEF_X; x++)
                putdot(pix[y][x], x, y);
}
```

The device-dependent routine *putdot*(*val*, *x*, *y*) should write a pixel with brightness *val* at location *x*, *y* on the screen. Note, however, that most display monitors will work substantially faster if you can write one scanline at a time, using, for example:

```
        for (y = 0; y < DEF_Y; y++)
                putline(pix[y], y);
```

Some display monitors use a different type of coordinate system, e.g., with *Y* at the top of the screen and 0 at the bottom. You can use the same routine, and simply subtract the *y* variable from *Y* to produce a picture that is right side up:

```
        for (y = 0; y < DEF_Y; y++)
                putline(pix[y], DEF_Y-1-y);
```

If you have a one-bit-per-pixel dot-mapped monitor, the image has to be halftoned before it can be displayed. Here is a routine that you can use. It uses a standard halftoning method, like the ones used for printing photos in newspapers, to map gray values onto pure black-and-white values. (For more information on halftoning methods, refer to the bibliography at the end of this chapter.)

```
#define RES     8

thresh[RES][RES] = {
        {  0, 128,  32, 160,   8, 136,  40, 168, },
        {192,  64, 224,  96, 200,  72, 232, 104, },
        { 48, 176,  16, 144,  56, 184,  24, 152, },
        {240, 112, 208,  80, 248, 120, 216,  88, },
        { 12, 140,  44, 172,   4, 132,  36, 164, },
        {204,  76, 236, 108, 196,  68, 228, 100, },
        { 60, 188,  28, 156,  52, 180,  20, 148, },
        {252, 124, 220,  92, 244, 116, 212,  84, },
};      /* an array with threshold values */

display(pix)
        unsigned char **pix;
{
        register int x, y;

        for (y = 0; y < DEF_Y; y++)
        for (x = 0; x < DEF_X; x++)
        {       if (pix[y][x] >= thresh[y%RES][x%RES])
                        putdot(1, x, y);
                else
                        putdot(0, x, y);
        }
}
```

Note again that it may be faster first to assemble a whole scanline of one bit dots in memory and write it to the screen with a single procedure call *putline*().

Including the display routine in the editor is done in three steps. First, extend the lexical analyzer to recognize a new keyword, such as *display*, and return a new token, e.g., *DISP*. Then add a value for the new token to the header file “popi.h,” e.g.,

```
#define DISP    272
```

Finally, extend the routine *parse*() to respond to the new command, by adding another case to the switch statement. For instance,

```
case DISP: display(src[CUROLD].pix); break;
```

If you are more daring, you want to consider including routine *putline*() directly inside *run*(). The best place to do this is at the end of the loop on variable *y*. The extra procedure call makes the program run a little slower. Being able to see the effect of a transformation happen in real-time, however, will more than make up for it.

In a similar way *popi* can be extended with any number of user-defined routines, either standard transformations that you would like to have predefined, or transformations that may be hard to express in the language of the transformation expressions. In Chapter **6** we include some examples of routines you may want to add in this manner.

Hints for Other Extensions

One of the first things you may want to do is to extend *popi* to handle both color and black-and-white images. One simple way to do this is to store pixel values not in 8-bit bytes but in 32-bit words (for instance, a long integer). You will need 8 bits each for the red, green, and blue color components, and use the remaining 8 bits to separate the color components within the pixel word. You will have to be careful to get the image arithmetic to work right, especially to avoid overflow between neighboring color components. But it is not too hard to do. Another method is, of course, to store the red, green, and blue components in three separate image files and use the editor as is.

The extension for polar coordinates (Chapter **3**) is relatively easy. The simplest method is to prepare two files with precomputed values of all *r* and *a* values. The editor loads the two files in memory, as if they were image files, and simply looks up each *r* and *a* value as needed, instead of computing them over and over on-the-fly. You will need to add two lines to the parsing routine *factor*(), to catch *r* and *a* in the same case statement that processes *x* and *y*. Then you will have to add a modest amount of code to the interpreter routine *run*() to look up the precomputed value for either *r* or *a*, given *x* and *y*. The code for *RVAL* in *run*() can serve as an example.

Trigonometric functions are also relatively easy to add. The lexical analyzer will need to recognize a few more character sequences, such as *sin*, *cos*, and *atan*. They can be translated into three new tokens and interpreted by *run*() similar to the token *POW*. Note, however, that the *sin* and *cos* functions return values between +1.0 and −1.0, while the editor works only with integers. A simple way around this is to have these functions return 1000 times their value, and to renormalize the results in the transformation expressions.

Adding an *undo* operation takes only one line of code, to be added to the case switch in routine *parse*() in main.c.

```
    case 'u': CUROLD = CURNEW; CURNEW = 1-CUROLD; break;
```

Those who really want to experiment will also want to consider the extension of *popi* with variables, interactively defined functions, and explicit control flow statements. Each such extension will increase both the power and the size of the editor. But be warned: If you are not a skilled programmer when you begin with these extensions, you probably will be when you complete them.

Books

A couple of books may be helpful if you would like to work with the software discussed above. The best reference to the *C* programming language is still Kernighan and Ritchie's manual from 1978. A discussion of the draft ANSI standard *C* language can be found in Harbison and Steele's book. Much more about the design of parsers and lexical analyzer can be found in the famous *dragon* books by Aho and others. A wealth of information on *C* programming can also be found in Kernighan and Pike's book on UNIX. Read especially Chapter **8** on program development if you consider extending *popi*.

An excellent introduction to digital halftoning methods can be found in Robert Ulichney's book.

For completeness the list below also includes a reference to a paper published in the *AT&T Technical Journal* with more information on the structure of the picture editor *pico*, *popi*'s bigger brother.

The C Programming Language, Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall, 1978, 2nd revised edition 1988, ISBN 0-13-110163-3.

C − A Reference Manual, Samual P. Harbison and Guy L. Steele Jr., Prentice-Hall, 2nd edition, 1987, ISBN 0-13-109802-0.

Compilers − Principles, Techniques and Tools, Al Aho, Ravi Sethi, and Jeff Ullman. Addison-Wesley, 1986, ISBN 0-201-10088-6.

The UNIX Programming Environment, Brian W. Kernighan and Rob Pike, Prentice-Hall, 1984, ISBN 0-13-937699-2.

Digital Halftoning, Robert Ulichney, The MIT Press, 1987, ISBN 0-262-21009-6.

"Pico − a Picture Editor," Gerard J. Holzmann, *AT&T Technical Journal*, Vol. 66, No. 2, 1987, pp. 2–13.