ON THE GRAPHIC DESIGN OF PROGRAM TEXT

Aaron Marcus
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720 USA

Ronald Baecker
Human Computing Resources Corporation
10 St. Mary Street
Toronto, Ontario M4Y 1P9

ABSTRACT

Computer programs, like literature, deserve attention not only to conceptual and verbal (linguistic) structure but also to visual structure, i.e., the qualitities of alphanumeric text fonts and other graphic symbols, the spatial arrangement of isolated texts and symbols, the temporal sequencing of individual parts of the program, and the use of color (including gray values). With the increasing numbers of programs of ever greater complexity, and with the widespread availability of high resolution raster displays, both soft copy and hard copy, it is essential and possible to enhance significantly the graphic design of program text.

The paper summarizes relevant principles from information-oriented graphic design, especially book design, and shows how a standard C program might be translated into a well-designed typographic version. The paper's intention is to acquaint the computer graphics community with the available and relevant concepts, literature, and expertise, and to demonstrate the great potential for the graphic design of computer programs.

INTRODUCTION

After three decades of continuous and in some cases revolutionary development of computer hardware and software systems, one aspect of computer technology has shown resilience to change: the presentation of computer programs themselves. Even though computer graphics systems have achieved dynamic, color, and multifont display capabilities, the visual qualities of alphanumeric and graphic symbols of program code has remained relatively simple. There are limited extensions of symbols in some languages, e.g., in APL, and some attempts to visually structure the page with simple indentations for control statements and groups of code lines. There are also more elaborate schemes such as Nassi-Schneiderman diagrams [Nassi], Warnier-Orr diagrams [Higgins], contour diagrams [Organick], and SADT diagrams [Ross]; however, the typographic repertoire and appearance of programs often remains little changed from the manner in which teletypewriters first printed out programs.

Three other relevant developments should be noted. First, many programs have grown longer, more complicated, and impenetrable to even skilled programmers. Second, programmers themselves have continued to be nomadic, often shifting jobs, inheriting programs from others and passing on a legacy of programs to still others. Third, the number of non-professional programmers, i.e., hobbyists, occasional programmers, or those without formal degrees in computer science, has increased considerably. Thus there is an increasing need for more effective, more productive means to create and maintain programs. Software engineering has recognized in a number of ways that programming is a kind of literature which requires good writing. The most widespread development has been the concern with the logical structure and expressive style of programs. Out of this concern have emerged many of the modern software development techniques, including top-design design and stepwise refinement [Wirth], structured programming [Dahl], modularity [Parnas], and software tools [Kernighan]. Another kind of

development has occurred in the organization of the editoral and production team that produces the writing, for example, the concepts of Chief Programmer Teams [Baker] and Structured Walkthroughs [Yourdon]. A third more recent development is the increase of interest in enhancing the technology to support the writing and maintaining of good programs, by providing, for example, integrated software development environments [Wasserman] and high-performance personal workstations specialized to the task of program development [Teitelman, Deutsch, Gutz].

Unfortunately, there is yet another approach to improving program writing and maintenance which also recognizes programs as literature but which has been systematically ignored. This approach concerns the visible language in which textual information is embodied.

Books embody literature. Graphic designers of books are concerned with legibility and readability, i.e., functionality and appeal to the reader. It seems reasonable that computer programs viewed as literature can be reformulated in typographic and graphic formats to better convey the content of well-written programs. By adopting the model of book literature, computer science can adapt and apply the expertise available from graphic design.

GRAPHIC DESIGN AND COMPUTER GRAPHICS

Graphic design is a discipline concerned with portraying facts, concepts, and emotions, as well as the logic of space and time (structure and process) in effective visible language. Information-oriented graphic design in particular is familiar with the concept of algorithms, for much of its work involves specifying algorithms for visual parameters to convey complex information. Traditionally, graphic designers have worked in printed media such as books, charts, diagrams, and maps. There is considerable professional literature [Tinker, Zachrisson, Hartley, Williamson] dealing with the subjects of typographic legibility, spatial composition, sequencing of pages, organization of content, use of color, and use of illustrative material. This discipline and its literature have knowledge appropriate to and helpful for the task of designing verbal/visual displays for computer technology.

A potentially symbiotic relationship exists between graphic design and computer graphics in creating the "three faces" of computer systems: outer-faces, inter-faces, and inner-faces [Marcus 1981b]. Outer-faces are the end display products of data processing: images of information such as texts, tables, charts, maps, and diagrams. Inter-faces are the user-oriented human-machine interfaces used to create outer-faces. These inter-faces are comprised of frames (online) and pages (offline) of process control, data structures, and their documentation. Inner-faces are the frames and pages of source code programs and documentation which builders and maintainers of computer systems require in order to support the inter-faces and outer-faces. By combining intuitive, practical skills and scientific knowledge, graphic designers can help synthesize prototype solutions for any of the three faces and assist in analyzing and developing finished displays which are not merely 'pretty', but which communicate information better.

In other articles [Marcus 1980, Marcus 1981a], the primary author has explained the relationship of graphic design to outer-faces and inter-faces. This present article extends the relevance of graphic design to inner-faces, to the design of programs as texts, and suggests further directions for research in program visualization. In order to emphasize book-oriented graphic design principles that are most relevant to the graphic design of inner-faces, most of the discussion will focus on the display of a single frame or page rather than the viewer/reader's response to an interactive display.

VISUAL PARAMETERS FOR PROGRAM VISUALIZATION

The graphic design of programs requires the designer to select symbols and formats for the primary components of programs: variables, constants, logical structures, processes, commentary, and documentation aids. Numerous choices exist for layout grids, typographic styles, size and spacing of text lines, organization of lists, and other means of visual emphasis. Explicit decisions on these matters are usually unspecified in the original conception of programming languages. The graphic designer can now specify them using the principles of similarity, proximity, clarity, con-

sistency, and simplicity [Marcus 1980] as a guide to organizing all the visual parameters into effective and attractive frames or pages.

These graphic design specifications constitute a visual/verbal algorithm for the construction of frames and pages. Program visualization should facilitate learning of the program text, aid memorization of its features, encourage concentrated attention, and assist in revealing a clear conceptual structure, especially in situations where the viewer/reader may be distracted or poorly motivated.

Traditional literature on typographic legibility [Rehe] and book design [Hartley, Williamson] concern printed texts rather than computer graphics displays. However, many of the principles would appear to be transferable to the computer graphics environment. The following principles are based on this literature and the primary author's own professional experience as a graphic designer in both traditional and computer-based media.

BASIC PRINCIPLES

The Layout Grid

Programs often appear with no particular attention to position in the frame or page. However, horizontal and vertical axes in the composition can be specified to create limits for the columns of text, margins, and documentation apparatus such as page numbers, special headings, etc. The grid determines the extent of portions of the program text, tab stops, areas for other standard components of the display, and the space in between these areas.

Typography

While fixed-width characters are usually used for program depiction, current typesetting equipment and a growing number of high resolution hardcopy devices (e.g., Xerox 9700) and display terminals (e.g., Three Rivers Perq, Xerox Star) can display variable-width characters. This has a significant impact on typefont variation, size of characters and width of text lines. General practice suggests the following:

Font variation should usually be limited to a single typeface or at most two, except for special mathematical needs. Within a typeface family, regular (roman), italic, and bold roman are standard means of increasing emphasis. Unless a great amount of effort is expended on the design of characters, adaptions of well-established typefaces such as Times Roman, Helvetica, Garamond, Universe, etc. should be used. Frame presentations allow for reversed video characters, blinking, variable intensities, and other means of emphasis. These should be used sparingly because of their strength of differentiation from normal text. Type size variation should be limited to at most three sizes for text materials, and these sizes should be quickly and easily distinguished. Generally 9 or 10 point type for a 14 inch viewing distance is standard printed text size. Optimum size will vary with the detailed characteristics of a display device and the viewing situation. Column widths should be limited to allow 40-60 characters per line [Rehe]. In the interrupted texts of programs, lines are usually unjustified on the right and a 'ragged right' approach to overall page composition is appropriate, i.e., titling, headings, and other elements should usually be flush left and ragged right.

Line spacing of text lines varies with display devices, but should usually produce greater space between lines of text than between words. Spacing variations should be limited to a maximum of three variations and should be used consistently to signify changes in content. Likewise tab settings should be limited to a few regular horizontal positions.

With respect to capitalization, all-capital settings should be avoided for continuous text materials. All capital words are more uniform in the shape of their outline and may slow reading speed by as much as 13% [Rehe, 36]; however they may be used for isolated keywords and phrases.

Sequencing

Page sequencing and organization in book literature has evolved specific components of complicated text structures, e.g., title pages, tables of contents, abstracts, indices, and running heads which appear on every page. Programs often appear without this standard documentation apparatus, but large programs viewed as literature should contain these items. In book literature, their exact form varies dramaticly depending on con-

tent. While there are no universal standards, certain technical conventions arise, and it would seem reasonable to assume that conventions for programs could be established.

AN EXAMPLE: TYPESETTING A SHORT C PROGRAM

The above summary is not meant to be an operational specification, only a clarification of what more detailed principles might involve. Principles are often best seen in application. Those presented here are worked out in the accompanying illustrations. Figures 1 and 2 present a 'before and after' version of typographic program visualization.

Figure 1 presents a program in an elementary typographic form using fixed-width characters of a single font with limited horizontal spacing variation. There is little typographic hierarchy. The program is more readable than those presentations which use all-capital typography and multiple commands per line, but there are still ways in which it can be made more readable.

Figure 2, a protypical black-and-white visualization, requires a very high resolution bit map display terminal or a very high quality hardcopy device. The actual images of Figure 2 were generated in Times Roman type using a computer-controlled phototypesetter. Figures 1 and 2 are part of a series of experimental prototypes pages for online or offline documentation which illustrates the full potential of a graphic design approach to textual program visualization. Spatial location, typographic symbol heirarchies, figure-field enhancements, indexes, abstracts, etc. are combined to create a clear, consistent, explicitly structured page that is legible and appealing to the reader. The following paragraphs detail the features of this design and elaborate upon the basic principles suggested above.

Spatial Organization

The entire page/frame is a mosaic of content units with standrad locations but in some cases variable size. In an interactive environment, each of these areas could be a window to a higher or lower level of information.

The upper part of the first page (or frame of a high resolution terminal) is intended to be a standard documentation cluster of header units grouped in a natural order. These include a documentation source, a program title, program subtitle(s), revision or last update, unique code number, chapter reference and page number. The items are then repeated on every frame/page.

In a strong titling banner below the headers, the title is presented in a size larger than all of the others and on a field of 50% gray to distinguish it clearly but not in an overpowering way from the rest of the text. The version date and unique code numbers are intended to advise the reader of the particular version of this program. There may be others similar to it that must be distinguished. The abstract is intended to be a 100 word summary of the function and significance of the program. It appears in italic to set it off from other elements. The author/guide and location band are intended to identify specific persons at the installation site who can be contacted for assistance in interpreting or using the program. Note the use of a tab setting at approximately half the width of the main column of text for presenting two columns of information.

Modules of the program are indicated by unique 50% gray bands with bold roman module names. Their size is the largest of three standard sizes of type for the textual material of the program. Bold type is used to keep the type legible on a gray background.

Comments appear in 7 point type as separate marginalia to the left of the main column of text. These are intended to be single line phrases that can help the reader to understand individual code lines. The comments column is approximately 40 characters wide and appears in the smallest of the three text sizes. As phrases, the comments appear without initial capital or periods. In keeping with all clusters of text, they are flush left, ragged right.

Footnotes appear as 8 point type in a separate band of space at the bottom of the page/frame set off by a thin rule as wide as the main text. They are more detailed and complete explanations of the significance of code lines (or any other element such as a title). They appear as full sentences with initial capitals and closing periods.

Spatial Grid

Because of variable-width typesetting, a given phrase is approximately one third narrower than its typewritten equivalent. To usefully divide the page/frame, the main text column allows approximately 60 characters of 10 point type. The wide column permits code to be indented in 1/2 inch increments several hierarchical levels while still maintaining approximately 40 or more characters per line.

Code Conventions

Within the 10 point type of the primary column of code, the following typographic conventions are used. The first time that a function (e.g., "calc") is defined within the program, it is set off by repeating the name of the function in 12 point bold type followed by a thin rule. In order to call attention to local functions, these functions defined within the program are shown in bold while global functions appear in regular roman. Constants are often digits; therefore, to keep them all similar in appearance, named constants are shown in all-capitals. Variables appear as italic. The standard C symbol syntax has been altered slightly, e.g., "/*" and "*/" are not used to surround comments, and "{" and "}" as procedural symbols are not used because of explicit spatial structure which makes them redundant. These redundant symbols have been tentatively removed to reduce visual clutter.

CONCLUSIONS

The intention of the prototype in Figure 2 is not to establish standards but to demonstrate how explicit typographic specifications based on graphic design principles might affect the presentation of computer programs. The prototype is intended only to focus attention on this approach, to raise expectations of program readability, and to raise interesting questions which further research in the visible language of computer programs might explore.

These questions can be clustered into six categories.

The first research topic deals with the appropriate use of typography to reveal formal syntactic, semantic, and pragmatic properties of programs and program elements. For example, what is the appropriate use of boldface and italic? Should multiple fonts be used? How should color be employed?

A second concern is with the design and layout of program elements on the page using systems of grids, overlays, and windows. How important is redundancy, as for example in the use of brackets plus horizontal spacing? How should secondary text (comments and commentaries) be clustered around the primary text (code)?

A third area for research is the possibility of substituting a set of well-designed icons or symbols (pictograms or ideograms) for certain combinations of alphanumerics that occur repetitively in program code. What should these icons or symbols be? To what extent can program documentation become more diagrammatic, and rely less on the linear text forms of current programming languages?

A fourth set of questions arise out of the possibilities that interactive computer graphics offer in the inclusion of movement, blinking, and other kinds of change into program documentation. More fundamentally, we must explore the relationship between static paper and dynamic screen representations of computer programs.

A fifth problem area is in the depiction of large directed graphs of great complexity, networks in which nodes are not single points but entire frames (combinations of signs) and in which links are explicitly stated or implied connections between nodes. The spatial layout and user navigation problems that occur may be seen, for example, in the enhancement of program text into Nassi-Schneiderman diagrams, Warnier-Orr diagrams, contour diagrams, and SADT diagrams.

The final research topic concerns the ability of a program visualization to facilitate the integration of the various conceptual levels at which a program may be described. What relation should exist between high resolution detailed views and low resolution overview images of the same program? What is an optimal sequence for the basic units of a "program book"? What would its other parts such as tables of contents and indices look like? How are they to be used?

Finally, what is the relation between reading and writing such complex visual representations?

If Figure 2 has some merit as a workable format for C programs and other languages, it is the authors' hope that designing a visible language scheme will be recognized as a distinct and demanding task requiring the assistance of graphic design. Other researchers and design professionals may be moved to explore the subject further with the goal of turning computer graphics capabilities back on their sources in computer programming to develop more effective and humane programming literature.

ACKNOWLEDGEMENTS

The authors acknowledge the work of Mr. Richard Sniderman of Human Computing Resources Corporation in executing the typesetting. They would also like to thank John McCarthy, Mike O'Dell, and Dennis Hall of Lawrence Berkeley Laboratory for their helpful advice. This work was supported by Human Computing Resources Corporation, and by the Applied Mathematics Sciences Research Program of the Office of Energy Research of the Department of Energy under contract W-7405-ENG-48.

REFERENCES

Baker, F.T., Chief Programmer Team Management of Production Programming, IBM Systems Journal 11:1 (1972), 56-73.

Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A.R., Structured Programming, Academic Press, London, 1972.

Deutsch, L. Peter and Taft, Edward A., "Requirements for an Experimental Programming Environment," Xerox Palo Alto Research Center Report CSL-80-10, June 1980.

Gutz, S., Wasserman, A.I. and Spier, M.J., Personal Development Systems for the Professional Programmer, Computer, April 1981, 45-53.

Hartley, J., Designing Instructional Text, Nichols, New York, 1978.

Higgins, David A., Program Design and Construction, Prentice-Hall, Englewood Cliffs NJ, 1979.

Kernighan, Brian W. and P.J. Plauger, Software Tools, Addison-Wesley Publishing Company, Reading, 1976.

Marcus, Aaron, "Computer-Assisted Chart Making from the Graphic Designer's Perspective", Computer Graphics, 13:3, 1980, 247-254.

Marcus, Aaron, "Designing the Face of an Interface", Proceedings, NCGA-81, National Computer Graphics Association National Conference, 1981, 207-215.

Marcus, Aaron, "Graphic Design and Computer Design: Know Business is Show Business", Centerline, Center for Design, San Francisco, July 1981, 6-7.

Nassi, I. and Schneiderman, B., "Flowcharting Techniques for Structured Programing," ACM Sigpan Notices, August 1973.

Organick, E. and Thomas, J.W., Computer-generated Semantics Displays, Proc. IFIP Congress, Applications Volume, 1974, 898-902.

Parnas, D.L., On the Criteria to be Used in Decomposing Systems into Modules, Comm. of the ACM 15:12 (December 1972), 1053-1058.

Rehe, Rolf, Typography: How to Make it Most Legible, Design Reseach International, Carmel, Indiana, 1974.

Ross, Douglas T., Structured Analysis (SA): A Language for Communicating Ideas, IEEE Transactions on Software Engineering SE-3:1, January 1977, 16-34.

Teitelman, Warren, "A Display Oriented Programmer's Assitant," Int. Jour. Man-Machine Studies, 11, 1979, 157-187.

Tinker, M.A., Legibility of Print, Iowa State University Press, Ames, 1963.

Wasserman, A.I., Tutorial: Software Development Environments, IEEE Computer Society Press, Los Alamitos CA, 1981.

Williamson, Hugh, Methods of Book Design, Oxford University Press, New York, 1966.

Wirth, N., Program Development by Stepwise Refinement, Comm. of the ACM 14:4 (April 1971), 221-227.

Yourdon, E., Structured Walkthroughs, Prentice-Hall, Englewood Cliffs NJ, 1979.

Zachrisson, B. Legibility of Printed Text, Alqvist and Wiksell, Stockholm, 1965.

Figure 1A

```
#include <stdio.h>
#define MAXOP 20        /* max size of operand, operator */
#define NUMBER '0'      /* signal that number found */
#define TOOBIG '9'      /* signal that string is too big */

calc()     /* reverse Polish desk calculator */
{
        int type;
        char s[MAXOP];
        double op2, atof(), pop(), push();

        while ((type = getop(s, MAXOP)) != EOF)
                switch (type){
                case NUMBER:
                        push(atof(s));
                        break;
                case '+':
                        push(pop() + pop());
                        break;
                case '*':
                        push(pop() * pop());
                        break;
                case '-':
                        op2 = pop();
                        push(pop() - op2);
                        break;
                case '/':
                        op2 = pop();
                        if (op2 != 0.0)
                                push (pop() / op2);
                        else
                                printf("zero divisor popped\0");
                        break;
                case '=':
                        printf("%f\0, push(pop()));
                        break;
                case 'c':
                        clear();
                        break;
                case TOOBIG:
                        printf("%.20s ... is too long\0, s);
                        break;          default:
                        printf("unknown command %c\0, type);
                        break;
```

Figure 1B

```
#define MAXVAL 100      /* maximum depth of val stack */

int sp = 0;             /* stack pointer */
double val[MAXVAL];     /* value stack */

double push(f)          /* push f onto value stack */
double f;
{
        if (sp < MAXVAL)
                return (val[sp++] = f);
        else {
                printf("error: stack full\0);
                clear();
                return(0);
        }
}

double pop()            /* pop top value from stack */
{
        if (sp > 0)
                return(val[--sp]);
        else {
                printf("error: stack empty\0);
                clear();
                return(0);
        }
}

clear()                 /* clear stack */
{
        sp = 0;
}
```

Figure 1C

```
getop(s, lim)           /* get next operator or operand */
char s[];
int lim;
{
        int i, c;
        while ((c = getch()) == ' ' || c == '' || c == '0')
                ;
        if (c != '.' && (c < '0' || c > '9'))
                return(c);
        s[0] = c;
        for (i = 1; (c = getchar()) >= '0' && c <= '9'; i++)
                if (i < lim)
                        s[i] = c;
        if (c == '.') {         /* collect fraction */
                if (i < lim)
                        s[i] = c;
                for (i++; (c = getchar()) >= '0' && c <= '9'; i++)
                        if (i < lim)
                                s[i] = c;
        }
        if (i < lim) {          /* number is ok */
                ungetch(c);
                s[i] = '';
                return(NUMBER);
        } else {                /* it's too big; skip rest of line */
                while (c != '0 && c != EOF)
                        c = getchar();
                s[lim - 1] = '';
                return(TOOBIG);
        }
}


#define BUFSIZE 100

char buf[BUFSIZE];      /* buffer for ungetch */
int bufp = 0;           /* next free position in buf */

getch()                 /* get a (possibly pushed back) character */
{
        return((bufz > 0) ? buf[--bufp] : getchar());
}

ungetch(c)              /* push character back on input */
int c;
{
        if (bufp > BUFSIZE)
                printf("ungetch: too many characters0);
        else
                buf[bufp++] = c;
```

Figure 2A

Desk Calculator[1]

Version of 1 August 1981 Ref. No. 12.345.67

This program implements a simple desk calculator which uses reverse Polish notation. Operands are pushed onto a stack. When an operator arrives its operands are popped, the operator is applied, and the result is pushed onto the stack.

For Assistance Call:

Aaron Marcus
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720
415-486-5070

Ronald Baecker & Richard Sniderman
Human Computing Resources Corp.
10 St. Mary St.
Toronto Ont. M4Y 1P9
416-922-1937

Control Module

max size of operand, operator
signal that number found
signal that string is too big

```
#include <stdio.h>
#define MAXOP 20
#define NUMBER '0'
#define TOOBIG '9'
```

calc

reverse Polish desk calculator

```
calc()
int type;
char s [MAXOP];
double op2, atof(), pop(), push();

while ((type = getop (s, MAXOP)) != EOF)
        switch (type)
        case NUMBER:
                push (atof (s));
                break;
        case '+':
                push (pop() + pop());
                break;
        case '*':
                push (pop() * pop());
                break;
        case '-':
                op2 = pop();
                push (pop() - op2)[2];
                break;
```

1 This program was authored by Brian Kernighan and Dennis Ritchie of Bell Laboratories, Murray Hill, New Jersey. These prototype visual enhancements to the C program were designed by Aaron Marcus with the assistance of Ronald Baecker and Richard Sniderman.

2 Because + and * are commutative operators, the order in which the popped operands are combined is irrelevant. For the - and / operators, the left and right operands must be distinguished.

Figure 2B


```
        case '/':
                op2 = pop();
                if (op2 != 0.0)
                        push (pop() / op2);
                else
                        printf ("zero divisor popped\n");
                break;
        case '=':
                printf ("\t%f\n", push (pop()));[3]
                break;
        case 'c':
                clear();
                break;
        case TOOBIG:
                printf ("%.20s ... is too long\n", s);
                break;
        default:
                printf ("unknown command %c\n", type);
                break;
```

Stack Management Module

maximum depth of val stack
stack pointer
value stack

```
#define MAXVAL 100
int sp = 0;
double val [MAXVAL];
```

push

push f onto value stack

```
double push (f)
double f;

if (sp < MAXVAL)
        return (val [sp++] = f);
else
        printf ("error; stack full\n");
        clear();
        return (0);
```

pop

pop top value from stack

```
double pop()
if (sp > 0)
        return (val [--sp]);
else
        printf ("error: stack empty\n");
        clear();
        return (0);
```

clear

clear stack

```
clear()
sp = 0;
```

3 The stack and stack pointer which are shared by **push**, **pop**, and **clear** are defined in the **Stack Management Module** and are not referred to by **main**. Thus this piece of code examines the top of the stack without disturbing it.

Figure 2C

Input Module

getop

get next operator or operand

```
getop (s, lim)
char s[];
int lim;
int i, c;

while ((c = getch()) == ' ' | c == '\t' | c == '\n')
        ;
if (c != '.' && (c < '0' | c > '9'))
        return (c);
s [0] = c;
for (i = 1; (c = getchar()) >= '0' && c <= '9'; i++)
        if (i < lim)
                s [i] = c;
```

collect fraction

```
if (c == '.')
        if (i < lim)
                s [i] = c;
        for (i++; (c = getchar()) >= '0' && c <= '9'; i++)
                if (i < lim)
                        s [i] = c;
```

number is ok

```
if (i < lim)
        ungetch (c);
        s [i] = '\0';
        return (NUMBER);
```

it's too big; skip rest of line

```
else
        while (c != '\n' && c != EOF)
                c = getchar();
        s [lim - 1] = '\0';
        return (TOOBIG);
```

buffer for ungetch
next free position in buf

```
#define BUFSIZE 100
char buf [BUFSIZE][4];
int bufp = 0;
```

getch

get a character

```
getch()
return ((bufp > 0) ? buf [--bufp] : getchar());
```

ungetch

push character back on input

```
ungetch (c)
int c;

if (bufp > BUFSIZE)
        printf ("ungetch: too many characters\n");
else
        buf [bufp++] = c;
```

4 A single character rather than an array could have been used since in this program it is never the case that more than one extra character than necessary is read. This is a more general implementation.